Programování v OS Linux

- Argumenty programu
- Synchronizace vláken
  1. Mutexy
  2. Semafory vláken
  3. Signály ve vláknech
- Komunikace mezi procesy (**IPC**)
  0. Pomocí argumentů příkazového řádku
  1. Signály
  2. Semafory procesů
  3. Sdílená paměť
  4. Fronty zpráv
  5. Roury
  6. Síťová rozhraní (*sockets*)

Přejít na první stránku

Argumenty programu

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

Argumenty příkazového řádku:

argc - počet řetězců v příkazové řádce

argv[ ] - pole řetězců reprezentujících jednotlivé argumenty programu

envp[ ] - pole řetězců reprezentujících jednotlivé proměnné prostředí

!!!POZOR !!! některé implementace jazyka C argument *envp[ ] nepodporují

V Linuxu se k proměnným prostředím přistupuje pomocí funkcí

char *getenv (const char *name);

int putenv(const char *string);

Přejít na první stránku

```
/* Program args.c */

#include <stdio.h>
#include <process.h>

    void main( int argc, char *argv[])
    {
        int count;

        /* Zobrazení jednotlivých argumentů příkazové řádky */
        printf( "\nCommand-line arguments:\n" );

        for ( count = 0; count < argc; count++ ) {
            if (argv[count][0] == '-' )
                printf( "option:  %s\n", argv[count]+1);

            else
                printf( "argument %d:  %s\n", count, argv[count]);
        }
        exit(0);
    }
```

Přejít na první stránku

```
$ ./args -i -lr hello -f “fred.c“

argument 0: args
option: i
option: lr
argument 3: hello
option: f
argument 5: fred.c
```

Přejít na první stránku

Procesy

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

int main ()
{
  printf (“The process ID is %d\n”, (int) getpid ());
  printf (“The parent process ID is %d\n”, (int) getppid ());
  return 0;
}
```

Přejít na první stránku

Procesy – system()

```
#include <stdlib.h>
int main ()
{
  int return_value;
  return_value = system (“ls -l /”);
  return return_value;
}
```

Přejít na první stránku

Procesy – fork, exec

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
  pid_t child_pid;
  printf (“the main program process ID is %d\n”, (int) getpid ());
  child_pid = fork ();

  if (child_pid != 0) {
  printf (“this is the parent process, with id %d\n”, (int) getpid ());
  printf (“the child’s process ID is %d\n”, (int) child_pid);
 }
 else
 printf (“this is the child process, with id %d\n”, (int) getpid ());
 return 0;
}
```

Přejít na první stránku

Vlákna - vytvoření

```
#include <pthread.h>
#include <stdio.h>
void * print_xs (void* unused)
{
  while (1)
  fputc (‘x’, stderr);
  return NULL;
}

int main ()
{
  pthread_t thread_id;
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  while (1)
    fputc (‘o’, stderr);
  return 0;
}
```

Přejít na první stránku

Mutexy

„MUTual EXclusion locks“

MUTEX

- zámek, který může uzamknout (zablokovat) v daném časovém okamžiku pouze jediné vlákno;
- další vlákna musí čekat dokud nebude mutex odblokovaný;
- odblokování musí provést stejné vlákno, které mutex uzamklo.

Přejít na první stránku

Mutexy - pokračování

```
#include <pthread.h>
```

ZÁKLADNÍ FUNKCE PRO PRÁCI S MUTEXY:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        pthread_mutex_ attr);    //může být NULL

int pthread_mutex_lock (pthread_mutex_t * mutex);
int pthread_mutex_unlock (pthread_mutex_t * mutex);
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

NÁVRATOVÁ HODNOTA

Úspěšné volání = 0

Jinak - kód chyby

Přejít na první stránku

Semafory vláken

#include <semaphore.h>

ZÁKLADNÍ FUNKCE PRO PRÁCI SE SEMAFORY:

int sem_init (sem_t *sem, int pshared, unsigned int value);

int sem_wait (sem_t *sem);
int sem_post (sem_t *sem);

int sem_destroy (sem_t *sem);

Přejít na první stránku

Semafory vláken - pokračování

sem_wait (&sem);

a) funkce v případě nenulové hodnoty semaforu ji dekrementuje o 1;
b) pokud je hodnota semaforu nulová, čeká dokud se nezmění; pak se provede bod a)

sem_post (&sem);

Funkce automaticky inkrementuje hodnotu semaforu o 1.

Přejít na první stránku

Signály ve vláknech

Problematika signálů je obecně vysvětlena v části vysvětlující komunikaci mezi procesy.

Vyslání signálu danému procesu :

int kill (int pid, int signal);

V případě vláken je lepší používat funkci:

pthread_kill(pthread_t thread, int signal);

Přejít na první stránku

Signály ve vláknech - pokračování

Stejné jako v případě signálů mezi procesy

Zpracování signálů:

```
# include <signal.h>

signal(int sig, void (*func)(int));
```

nebo lépe

```
int sigaction (int sig, const struct sigaction *act,
               struct sigaction *oact);
```

Přejít na první stránku

Komunikace mezi procesy

Signály

- oznamují procesu, že došlo k nějaké události;
- chovají se jako „programová přerušení“ - objevují se asynchronně vůči běhu procesu;
- signály může odesílat:

  *** jádro operačního systému**

  *** jeden proces druhému (nebo sám sobě)**

Přejít na první stránku

Signály - generování

Způsoby generování

* jádro generuje některé signály na základě chybových stavů procesu (např. SIGSEGV chyba adresace mimo adresový prostor procesu, SIGFPE neplatná matematická operace při výpočtu v pohyblivé řádové čárce)
* z příkazového řádku příkazem **kill** - např. ukončení programu

  s PID 12587pomocí signálu SIGKILL **$ kill -KILL 12587**
* pomocí vybraných terminálových znaků - např. stisk Ctrl C generuje signál SIGINT (ukončení probíhajícího procesu), Ctrl \ - SIGQUIT (programové přerušení procesu)

Přejít na první stránku

Signály - generování

*** z procesu -** voláním funkce:

int kill (int pid, int signal);

kde **pid > 0** **PID** konkrétního procesu, kterému bude poslán signál

0 signál poslán všem procesům skupiny (s výjimkou vybraných systémových), do níž proces patří

-1 signál poslán všem procesům (až na vybrané systémové - dle implementace); standard POSIX chování funkce pro tuto hodnotu nespecifikuje

pid < -1 signál poslán všem procesům jejichž identifikace skupiny je shodná s absolutní hodnotou pid

V případě vláken je lepší používat funkci:

pthread_kill(pthread_t thread, int signal);

Přejít na první stránku

Signály - zpracování

Způsoby zpracování v procesu

1. Implicitní zpracování;

2. Ignorování signálu;

3. Zpracování pomocí určené funkce - specifikované pomocí volání jádra
   `signal()`- NENÍ ve standardu POSIX !

```
void (*signal(int sig, void (*func)(int)))(int);
```

kde sig - specifikace signálu, který má být odchycen

func - funkce, která bude volána po obdržení signálu

Přejít na první stránku

Signály - zpracování

```
# include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

kde

sig - specifikace signálu, který má být odchycen

func – adresa funkce, která bude volána po obdržení signálu **sig** nebo jedna z hodnot:

SIG_IGN signál bude ignorován (nefunguje v případě SIGKILL)

SIG_DFL signál bude zpracován implicitním způsobem

Např. signál SIGINT má být ignorován:

```
signal (SIGINT, SIG_IGN);
```

Přejít na první stránku

Signály - příklad zpracování signálu SIGINT = Ctrl C

```
/* Stones, Matthew str. 360 */

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig) {
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
    (void) signal(SIGINT, ouch);
    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Přejít na první stránku

Signály - použití volání jádra **sigaction** (POSIX)

```
# include <signal.h>

int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

Struktura sigaction definuje akce, které se provedou po obdržení určeného signálu.

SLOŽKY struktury sigaction:

```
void (*) (int) sa_handler; /* funkce, SIG_DFL nebo
                              SIG_IGN */

sigset_t sa_mask;     /* maska signálů - tyto
                         signály budou blokovány */

int sa_flags;         /* modifikátory */
```

Např.:

```
sigaction (SIGUSR1, &sa, NULL);
```

Přejít na první stránku

Signály - použití funkce sigaction

Maska `sa_mask se nastavuje pomocí funkcí:`

```
int sigemptyset(sigset_t *set); /*prázdná množina sig.*/
int sigfillset(sigset_t *set);  /* všechny signály */
int sigaddset(sigset_t *set,int signo); /*přidání sig.*/
int sigdellset(sigset_t *set,int signo); /*odebrání sig.*/
```

Další funkce pro práci se signály:

```
/* Zablokování procesu do příchodu signálu */
int pause(void);
/* Vygenerování signálu SIGALRM po uplynutí určeného
   časového intervalu seconds */
unsigned int alarm(unsigned int seconds);
```

Přejít na první stránku

Signály - příklad použití funkce `sigaction`

```
/* Příklad - Stones, Matthew str. 365 */

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
  printf("OUCH! - I got signal %d\n", sig);
}
```

Přejít na první stránku

```
/* Příklad - pokračování */

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, NULL);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Přejít na první stránku

Semafory

Semafory jsou základním mechanismem synchronizace procesů. Ve srovnání se semafory vláken jsou semafory procesů implementačně výrazně náročnější.

Funkce pro práci se semafory jsou obsaženy v knihovně s hlavičkovým souborem **<sys/sem.h>**

Vytvoření semaforu:

int semget();

Provádění operací wait a post se semaforem:

int semop();

Řízení semaforu:

int semctl();

Přejít na první stránku

Příklad - binární semafor

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* Definice unie semun */
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
```

Přejít na první stránku

Binární semafor - alokace a dealokace

```
int binary_semaphore_allocation (key_t key,
                 int sem_flags)
{
  return semget (key, 1, sem_flags);
}

int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1,
   IPC_RMID,ignored_argument);
}
```

Přejít na první stránku

Binární semafor - inicializace

```
/* Inicializace binárního semaforu s hodnotou 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);
}
```

Přejít na první stránku

Binární semafor - operace *wait*

```
/* Čeká, pokud je hodnota rovna nule.
   Pokud je hodnota > 0, dekrementuje ji o 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);
}
```

Přejít na první stránku

binární semafor - operace ***post***

```
/* Okamžitá inkrementace hodnoty semaforu o 1. */

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);
}
```

Přejít na první stránku

Práce se soubory a zařízeními

Vysokoúrovňový přístup:

- funkce obsaženy ve standardní knihovně (hlavičkový soubor `stdio.h`);
- pracuje se se souborovými proudy (*streams*);
- *stream* je implementován jako pointer na strukturu FILE;
- při spuštění programu se automaticky otevírají tři proudy stdin, stdout a stderr reprezentující standardní vstup, výstup a chybový výstup;

Přejít na první stránku

Práce se soubory a zařízeními

- přístup k souboru se získá pomocí funkce:

  FILE *fopen(const char *filename, const char *mode);

- další funkce, např.:
  fread(), fwrite(), fseek(), fgets(), fprintf(), fscanf()

- ukončení přístupu k souboru (=uzavření proudu) pomocí funkce:
  int fclose(FILE *stream);

Přejít na první stránku

Práce se soubory a zařízeními

Nízkoúrovňový přístup k souborům a zařízením:

- každý proces má k dispozici několik deskriptorů souborů (reprezentovány čísly typu **`int`**), které slouží k přístupu k otevřeným souborům nebo zařízením;

- při spuštění programu se automaticky otevírají tři zařízení: standardní vstup (0), výstup (1) a chybový výstup (2);

- funkce resp. jejich parametry jsou obsaženy v knihovnách s hlavičkovými soubory uni**`std.h`**, **`sys/types.h`** a **`sys/stat.h`**;

Přejít na první stránku

Práce se soubory a zařízeními - pokračování

- nový deskriptor soubor se získá pomocí funkce:

int open(const char *path, int oflags[,int mode]);

Např.: **fd = open(“/dev/tty”,O_RDW);**

nebo **in = open(“file.out”,O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);**

- další funkce, např.: **read(), write(), ioctl(), lseek(), fstat();**

- ukončení přístupu k souboru (=uvolnění deskriptoru):

int close(int filedesc);

Přejít na první stránku

Listing 4.15 (*thread-pid*) Print Process IDs for Threads

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

void* thread_function (void* arg)
{
  fprintf (stderr, “child thread pid is %d\n”, (int) getpid ());

  /* Spin forever. */
  while (1);
  return NULL;
}
```


Listing 4.15 (*thread-pid*) Print Process IDs for Threads

```
int main ()
{
  pthread_t thread;

  fprintf (stderr, “main thread pid is %d\n”, (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);

  /* Spin forever. */
  while (1);
  return 0;
}
```


Job Queue Thread Function, Protected by a Mutex

```
#include <malloc.h>
#include <pthread.h>

struct job {
/* Link field for linked list. */
struct job* next;

/* Other fields describing work to be done... */
};

/* A linked list of pending jobs. */
struct job* job_queue;

/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Přejít na první stránku

```
/* Process queued jobs until the queue is empty. */
void* thread_function (void* arg)
{
while (1) {
struct job* next_job;
/* Lock the mutex on the job queue. */
pthread_mutex_lock (&job_queue_mutex);

/* Now it's safe to check if the queue is empty. */
if (job_queue == NULL)
next_job = NULL;
else {
/* Get the next available job. */
next_job = job_queue;
/* Remove this job from the list. */
job_queue = job_queue->next;
}
```

Přejít na první stránku

```
/* Unlock the mutex on the job queue because we're done with
the queue for now. */
pthread_mutex_unlock (&job_queue_mutex);

/* Was the queue empty? If so, end the thread. */
if (next_job == NULL)
break;

/* Carry out the work. */
process_job (next_job);

/* Clean up. */
free (next_job);
}
return NULL;
}
```


Přejít na první stránku

Synchronizace vláken pomocí MUTEXů (příklad)

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

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
```

Přejít na první stránku

```
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
```

Přejít na první stránku

```
res = pthread_create(&a_thread, NULL, thread_function, NULL);
if (res != 0) {
    perror ("Thread creation failed");
    exit (EXIT_FAILURE);
}
```

Přejít na první stránku

```
pthread_mutex_lock(&work_mutex);
    printf("Input some text. Enter 'end' to finish\n");
    while(!time_to_exit) {
        fgets(work_area, WORK_SIZE, stdin);
        pthread_mutex_unlock(&work_mutex);
        while(1) {
            pthread_mutex_lock(&work_mutex);
            if (work_area[0] != '\0') {
                pthread_mutex_unlock(&work_mutex);
                sleep(1);
            }
            else {  break; }
        }
    }
```

Přejít na první stránku

```
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}
```

Přejít na první stránku

```
void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);

    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);

        sleep(1);

        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);

            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
```

Přejít na první stránku

```
    time_to_exit = 1;
    work_area[0] = '\0';

    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

Přejít na první stránku