Systémové programovanie / ÚINF/SPR1b

SPR1a

Róbert Novotný
robert.novotny@upjs.sk

1. 3. 2011

Rúry (pipes)

- zariadenie pre **jednosmernú** komunikáciu
  - do jedného konca zapisujeme
  - z druhého čítame
- **sériové** zariadenie
  - dáta sú čítané vždy v rovnakom poradí, ako boli do rúry zapisované
- používané ako mechanizmus **IPC**
  - medziprocesorová komunikácia

Rúry v shelli


```
ls | less
```

- shell vytvorí dva procesy, spojí ich **rúrou**
  - stdout pre **ls** sa spojí so **stdin** lessu
- rúra má obmedzenú **kapacitu**
  - analógia skladu s obmedzenou kapacitou
  - ak sa sklad naplní, zapisujúci proces blokuje
  - ak je sklad prázdny, čítajúci proces blokuje

mechanizmus synchronizácie procesov!

Vytváranie rúr

pipe()

- systémové volanie
- očakáva pole dĺžky 2
- na index 0 vloží deskriptor súboru pre čítanie
- na index 1 vloží deskriptor súboru pre zápis

všetko je súbor!

Odovzdávanie hodnôt


```
int pipe_fds[2]
int fd_read;
int fd_write;

pipe(pipe_fds);
fd_read = pipe_fds[0];
fd_write = pipe_fds[1];
```

- dáta zapísané do **fd_write** môžu byť načítané z **fd_read**

Idióm pre komunikáciu medzi rodičom a potomkom

- rúra sa notoricky používa pre komunikáciu medzi rodičom a potomkom
  - potomok vznikne forknutím rodiča
- rodič zapisuje do rúry a dieťa číta
- idióm:
  1. pomocou **pipe()** vytvoríme rúru
  2. **fork**neme sa
  3. rodič a dieťa zatvoria tie konce rúr, ktoré **nepoužívajú**
     - **fork**() duplikuje pre potomka všetky otvorené súborové deskriptory

Rúry: ukážka


```
void writer(const char* sprava, int pocet, FILE * f) {
  for (; pocet > 0; --pocet) {
    fprintf(f, “%s\n”, sprava); fflush(f);
    sleep(1);
  }
}
```

kvôli demonštrácii na chvíľu zaspíme

```
void reader(FILE * f) {
  char buffer[1024];
  while (!feof(f) && !ferror(f)
     && fgets(buffer, sizeof(buffer), f) != NULL)
  {
    fputs (buffer, stdout);
  }
}
```

Fork/pipe


```
int fds[2];
pid_t pid;
pipe(fds);
pid = fork();
if (pid == (pid_t) 0) {
  FILE * stream;
  close(fds[1]);
  stream = fdopen(fds[0], "r");
  reader(stream);
  close(fds[0]);
} else {
  /* ... */
}
return 0;
```

sme u potomka

1. zatvoríme koniec rúry na zápis
2. prevedieme deskriptor na súbor
3. načítame dáta
4. zavrieme rúru

Fork/pipe


```
int fds[2];
pid_t pid;
pipe(fds);
pid = fork();
if (pid == (pid_t) 0) {
  /* ... */
} else {
    FILE* stream;
    close(fds[0]);
    stream = fdopen(fds[1], "w");
    writer("Hello, world.", 5, stream);
    close(fds[1]);
}
return 0;
```

1. zatvoríme koniec rúry na čítanie
2. prevedieme deskriptor na súbor
3. zapíšeme dáta
4. zavrieme rúru

sme u rodiča

Zatváranie nepotrebných koncov

- potomkovský proces zdedí všetky otvorené deskriptory súborov
- **zdedí** deskriptor pre čítanie i zápis
- potomok však **nepotrebuje zapisovať**
  - koniec pre zápis uňho zavrieme
  - predchádza sa mrhaniu zdrojmi
  - predchádza sa bezpečnostným chybám
- rodič **nepotrebuje čítať**

Presmerovanie stdin, stdout a stderr

- často potrebujeme presmerovať jeden koniec rúry do niektorého zo štandardných prúdov
- príklad: chceme stotožniť koniec rúry určený pre čítanie so **stdin**

dup2()

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

zatvorí stdin a znovuotvorí ho ako duplikát **fd**

Fork/pipe


```
int fds[2];
pid_t pid;
pipe(fds);
pid = fork();
if (pid == (pid_t) 0) {
  FILE * stream;
  close(fds[1]);
  dup2(fds[0], STDIN_FILENO);
  execlp("sort", "sort", 0);
} else {
  /* ... */
}
return 0;
```

sme u potomka

1. zatvoríme koniec rúry na zápis
2. stotožníme čítací koniec so **stdin**
3. nahradíme potomkovský proces programom **sort**

Fork/pipe


```
/*...*/
if (pid == (pid_t) 0) {
  /* ... */
} else {
    FILE* stream;
    close(fds[0]);
    stream = fdopen(fds[1], “w”);
    fprintf(stream, "Raz\n");
    fprintf(stream, "Dva\n");
    fflush(stream);
    close(fds[1]);
    waitpid(pid, NULL, 0);
}
return 0;
```

1. zatvoríme koniec rúry na čítanie
2. prevedieme deskriptor na súbor
3. zapíšeme dáta
4. zavrieme rúru
5. čakáme, kým neskončí potomok

sme u rodiča

Uľahčenie práce s rúrami

- komunikácia cez rúru s programom bežiacim v podprocese je veľmi častá
- namiesto komba **pipe, fork, dup2, exec, fdopen**

popen()

pclose()

```
popen("sort", "w");
```

```
pclose(stream);
```

Fork/pipe

1. v **popen()** uvedieme program
2. "w" = chceme zapisovať do potomka

```
#include <stdio.h>
#include <unistd.h>
int main() {
  FILE* stream = popen(“sort”, “w”);
  fprintf(stream, “Raz.\n”);
  fprintf(stream, “Dva.\n”);
  fprintf(stream, “One fish, two fish.\n”);
  return pclose(stream);
}
```

stream = koniec rúry (zapisovací)

opačný koniec = stdin potomka

FIFO: rúry v súborovom systéme

- FIFO = klasický front
- v UNIXe = rúra, ktorá má meno v súborovom systéme
  - a.k.a. **named pipe** (pomenovaná rúra)

```
mkfifo /tmp/mojefifo
```

príkaz v shelli

FIFO: rúry v súborovom systéme

- vieme sa veseliť
- vytvoríme FIFO
- v jednom okne čítame dáta

príkaz v shelli

```
cat < /tmp/mojefifo
```

- v druhom zapisujeme

```
cat > /tmp/mojefifo
```

Práca s FIFO z kódu

- vytvorenie:

mkfifo()

  - zadáme cestu a práva (analogicky ako v shelli)
- používanie:
  - FIFO je **súbor**
  - fičíme na klasických funkciách (fopen(), fscanf()...)

Vlastnosti fíf

- FIFO môže mať viacero zapisovateľov a čitateľov
- dáta zapisované atomicky
  - do veľkosti 4KB na Linuxe
- dáta od viacerých zapisovateľov/čitateľov môžu byť prekladané
- Windows Named Pipes vs Unix Pipes
  - rúra vo Win32 môže byť sieťová
    - v Unixe: to isté sa dosiahne socketmi
  - viacero zapisovateľov/čitateľov bez prekladania
  - rúru možno použiť na obojsmernú komunikáciu

Vlákna

- ako umožniť beh viacerých vecí naraz?
- popri procesoch ďalší mechanizmus paralelného vykonávania úloh
- kernel ich vykonáva striedavo a budí dojem paralelného behu

OS > Proces > Vlákna

- vlákna existujú v rámci procesu
- každý proces má aspoň jedno vlákno, v ktorom sa vykonáva program
- vlákno môže spustiť ďalšie vlákna
- každé z vlákien môže vykonávať rozličnú časť programu

Vlákna vs. forknuté procesy

- forknuté procesy
  - z rodiča sa skopíruje virtuálna pamäť, deskriptory súborov...
- vlákna
  - nič sa nekopíruje
  - všetky vlákna procesu zdieľajú premenné, deskriptory a ostatné systémové prostriedky

Vlákna v operačných systémoch

- Linux:
  - implementuje štandard POSIX
  - pthreads <pthread.h>
  - pri kompilácii treba dodať –pthread
- Windows:
  - vlastné riešenie v rámci Win32 API

http://software.intel.com/en-us/blogs/2006/10/19/why-windows-threads-are-better-than-posix-threads/

Vytvorenie vlákna

- thread ID: identifikátor vlákna
  - analogické k PID
  - typ **pthread_t**
- thread function: kód, ktorý má bežať v rámci vlákna
  - 1 parameter void *, vracia void *
- thread argument: parameter pre funkciu

Ukážkový kód


```
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;
}
```

thread function

thread ID

Ukážkový kód


```
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;
}
```

- *thread ID (vyplní ho)*
- *atribúty vlákna*
- *pointer na thread function*
- *thread arguments*

Parametre pre pthread_create()

- pointer na **pthread_id**
  - vráti sa v ňom thread ID
- pointer na **atribúty vlákna**
  - možnosť nastaviť spôsob, akým vlákno interaguje so zvyškom programu
  - defaultne NULL
- pointer na **funkciu vlákna**
  - deklarácia: void * (*) (void *)
- **argument** funkcie vlákna
  - preposlaný bez zmeny do funkcie vlákna

Po zavolaní pthread_create() sa začne paralelne vykonávať funkcia vlákna.

Ako ukončiť vlákna?

- kód vlákna dobehne po vrátení hodnoty z funkcie vlákna
  - hodnota sa objaví ako návratová hodnota z **pthread_create()**
- alebo po zavolaní **pthread_exit()**
  - parameter pre túto funkciu sa objaví ako návratová hodnota

Pozor! Ak púšťame pthread_create() z hlavného vlákna, v štandardnom chovaní dobehne hlavne vlákno skôr ako novovytvorené! Potenciálny zdroj chýb!

Odovzdávanie hodnôt


```
void * print_xs(void * parameter) {
  int i;
  int * pocet_znakov = (int *) parameter;
  for(i = 0; i < pocet_znakov; i++) {
    putc('X');
  }
}
int main () {
  pthread_t thread_id;
  pthread_create(&thread_id, NULL, &print_xs, NULL);
  while (1) fputc(‘o’, stderr);
  return 0;
}
```

Odovzdávanie hodnôt

```
void * print_xs(void * parameter) {
  int i;
  int * pocet_znakov = (int *) parameter;
  for(i = 0; i < pocet_znakov; i++) {
    putc('X');
  }
}
int main() {
  pthread_t thread1_id, thread2_id;
  int pocet1, pocet2;
  pocet1 = 5; pocet2 = 10;
  pthread_create(&thread1_id, NULL, &print_xs, &pocet1);
  pthread_create(&thread2_id, NULL, &print_xs, &pocet2);
  return 0;
}
```

Hlavné vlákno dobehne skôr ako dve nové vlákna!
Pamäť pre lokálne premenné sa automaticky dealokuje!

SPR1a

Memory leak!

Čo ak hlavné vlákno dobehne skôr?

- jedno z riešení problému?
- čakajme v main(), kým nedobehnú naše dve vlákna

```
pthread_join(pthread_t, void *)
```

- uvedieme **ID vlákna**, na ktoré máme čakať
- a uvedieme pointer, do ktorého sa vloží **návratová hodnota** funkcie vlákna

Odovzdávanie hodnôt


```
int main() {
  pthread_t thread1_id, thread2_id;
  int pocet1, pocet2;
  pocet1 = 5; pocet2 = 10;
  pthread_create(&thread1_id, NULL, &print_xs, &pocet1);
  pthread_create(&thread2_id, NULL, &print_xs, &pocet2);
  pthread_join(thread1_id, NULL);
  pthread_join(thread2_id, NULL);
  return 0;
}
```

čakáme na dobehnutie oboch vlákien

NULL = nepotrebujeme návratovú hodnotu funkcie vlákna

Morálne ponaučenie

- nezabúdajte na to, že dáta odovzdané vláknu treba dealokovať!
- to platí pre lokálne premenné
  - zlé použitie sme videli
- i pre dynamicky alokované premenné!
  - pamäť treba uvoľniť cez **free()**

Návratové hodnoty z vlákna

- druhým parametrom pre pthread_join() je pointer na návratovú hodnotu z funkcie vlákna
- keďže je typu void *, môžeme tam hodiť ľubovoľný pointer
- pri vyzdvihovaní hodnoty ho treba pretypovať

Odovzdávanie hodnôt


```
void * hladaj_prvocislo(void * poradie_prvocisla) {
  /* … zlozite vypocty …*/
  return (void *) vysledok;
}
int main() {
  pthread_t vlakno;
  int ktore_prvocislo = 4000;
  int prvocislo;
  pthread_create(&vlakno, NULL,
                 &hladaj_prvocislo, &ktore_prvocislo);
  pthread_join(vlakno, (void *) &prvocislo);
  printf("Vysledok: %d", prvocislo);
}
```

Ako zistiť aktuálne vlákno?

- **pthread_self()**
  - vracia ID aktuálneho vlákna, v ktorom beží kód
- **pthread_equal()**
  - zistí, či ID vlákna je rovnaké ako ID iného vlákna
  - porovnávajú sa hodnoty typu **pthread_t**