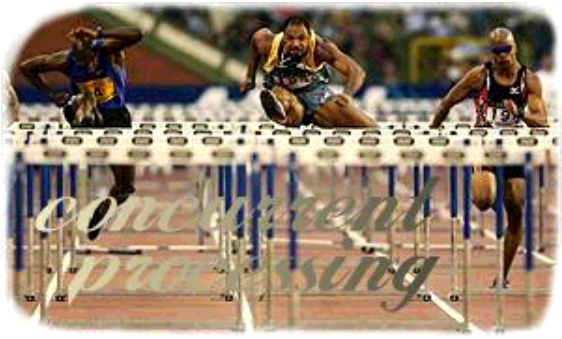
LinuxFocus article number 296
http://linuxfocus.org

Evenwijdig programmeren - Bericht wachtrijen (2)



door Leonardo Giordani
<leo.giordani(at)libero.it>

Over de auteur:

Ik heb juist mijn diploma ontvangen van de Faculteit van Telecommunicatie Engineering in Politecnico in Milaan, ben geïnteresseerd in programmeren (voornamelijk Assembly en C/C++). En sinds 1999 werk ik bijna alleen maar met Linux/Unix.

Vertaald naar het Nederlands door:
Guus Snijders
<ghs(at)linuxfocus.org>

Kort:

Deze serie artikelen heeft als doel de lezer in het concept van multitasking, en de implementatie ervan in een Linux OS, te introduceren. We beginnen met de theoretische concepten die aan de basis liggen van multitasking, en zullen eindigen met het schrijven van een volledige toepassing om de communicatie tussen processen te demonstreren met een simpel maar krachtig communicatieprotocol.
Vereisten om dit artikel te begrijpen zijn:

- minimale kennis van de shell
- Basiskennis van de C taal (syntax, lussen, bibliotheken)

Alle referenties naar manual pages zijn geplaatst tussen accolades na de commando naam. Alle glibc functies zijn gedocumenteerd met "info Libc".

Het is misschien een goed idee om ook enkele van de voorgaande artikelen in deze serie eerst te lezen:

- Evenwijdig programmeren - Principes van en kennismaking met processen
- Evenwijdig programmeren - communicatie tussen processen
- Evenwijdig programmeren - Bericht wachtrijen (1)

Introductie

In het vorige artikel van deze kleine serie zagen we hoe we twee (of meer) processen kunnen laten synchroniseren en samenwerken door het gebruik van berichtwachtrijen. In deze zullen we verder gaan en beginnen met het maken van simpel protocol voor onze bericht uitwisseling.

We hadden al gezegd dat een protocol een set regels is die mensen of machines in staat stellen te praten, zelfs als ze anders zijn. Zo is Engels bijvoorbeeld een protocol, omdat het me in staat stelt te spreken tot mijn Indische lezers (die altijd erg geïnteresseerd zijn in wat ik schrijf). Een iets meer Linux-gerelateerd voorbeeld, is het hercompileren van je kernel (wees niet bang, het is niet zo moeilijk), waarbij je zeker de Networking sectie zult opmerken, waar je je kernel verschillende netwerk protocollen kunt laten ondersteunen, zoals TCP/IP.

Om een protocol te creëren, zullen we moeten beslissen wat voor soort applicatie we zullen ontwikkelen. Deze keer zullen we een eenvoudige telefoon switch simulator bouwen. Het main proces zal de telefoon switch zijn, en de ’zoon’ processen zullen zich gedragen als gebruikers: we zullen gebruikers berichten naar elkaar laten sturen door de switch.

Het protocol zal drie verschillende situaties behelsen: de geboorte van een gebruiker (dwz de gebruiker bestaat en is verbonden), het normale werk van de gebruiker, en de dood van een gebruiker (hij is niet langer verbonden). Laten we spreken over deze drie situaties:

Als een gebruiker verbindt met het systeem, creëert hij zijn eigen berichtwachtrij (vergeet niet dat we het hebben over processen), de identifiers moeten naar de switch worden gestuurd om deze te laten weten hoe te communiceren met deze gebruiker. Hier heeft het de tijd om een aantal data structuren te creëren, indien nodig. Het ontvangt van de switch de indentifier van de wachtrij waar hij de berichten heen kan schrijven die door de switch verstuurd moeten worden naar andere gebruikers.

De gebruiker kan berichten versturen en ontvangen. Als hij een bericht ontvangt van een andere gebruiker, kunnen er twee situaties onstaan: de ontvanger is verbonden, of niet. We besluiten dat in beide gevallen een bevestiging moet worden verstuurd naar de zender, om deze te laten weten wat er gebeurt met zijn bericht. Dit vereist geen acties van de ontvanger zelf, de switch zou dit moeten doen.

Als een gebruiker de verbinding met het systeem verbreekt, zou hij de switch moeten informeren, maar verder zijn er geen acties nodig. De metacode om deze manier van werken te beschrijven, is als volgt:

```
/* Birth */
create_queue
init
send_alive
send_queue_id
get_switch_queue_id

/* Work */
while(!leaving){
 receive_all
 if(<send condition>){
  send_message
 }
 if(<leave condition>){
  leaving = 1
```

```
 }
}

/* Death */
send_dead
```

Nu dienen we het gedrag van onze telefoon switch te bepalen: als een gebruiker verbindt, stuurt deze ons een bericht met de identifier van zijn bericht wachtrij; dus, dienen we deze op te slaan om berichten voor deze gebruiker af te leveren en te antwoorden met de identifier van een wachtrij waar hij zijn bericht kan laten die we naar andere gebruikers moeten sturen. Dan moeten we alle ontvangen berichten analyseren en controleren of de ontvangers aanwezig zijn: als de ontvanger verbonden is, moeten we het bericht versturen, als de ontvanger niet verbonden is, moeten we het bericht verwijderen; in beide gevallen moeten we de zender bevestigen. Als een gebruiker verdwijnt verwijderen we simpelweg de identifier van zijn wachtrij, zodat deze onbereikbaar wordt.

Weer, onze metacode implementatie is

```
while(1){
 /* New user */
 if (<birth of a user>){
  get_queue_id
  send switch_queue_id
 }

 /* User dies */
 if (<death of a user>){
  remove_user
 }

 /* Messages delivering */
 check_message
 if (<user alive>){
  send_message
  ack_sender_ok
 }
 else{
  ack_sender_error
 }
}
```

Fout afhandeling

Het afhandelen van fout condities is een van de moeilijkste en belangrijkste dingen om te doen in een projekt. Een goed en compleet subsysteem om op fouten te controleren kan tot 50% van de code die we schrijven in beslag nemen. In dit artikel zal ik niet uitleggen hoe goede fout controle routines kunnen worden geschreven, omdat het onderwerp te complex is, maar vanaf nu zal ik altijd controleren en reageren op fout condities. Een goede introductie in fout controle kan gevonden worden in de glibc manual (www.gnu.org) maar, indien geïnteresseerd, zal ik hier later een artikel aan wijden.

Protocol implementatie - Laag 1

Ons kleine protocol bestaat uit twee lagen: de eerste (de laagste) bestaat uit functies om wachtrijen te

beheren en berichten klaar te maken en te versturen, terwijl de hogere laag het protocol implementeert als functies die verglijkbaar zijn met de metacode die we gebruikten om het gedrag van de switch en de gebruikers te beschrijven.

Het eerste ding om te doen is een structuur te creëren voor ons bericht met het kernel prototype van msgbuf

```
typedef struct
{
  int service;
  int sender;
  int receiver;
  int data;
} messg_t;

typedef struct
{
  long mtype; /* Tipo del messaggio */
  messg_t messaggio;
} mymsgbuf_t;
```

Dit is iets algemeens dat we later kunnen uitbreiden: de zender en ontvanger velden bevatten een gebruikers id en het data veld bevat de eigenlijke data, terwijl het service veld wordt gebruikt om een service van de switch aan te vragen. We zouden ons bijvoorbeeld kunnen voorstellen twee services te hebben: een voor onmiddelijke en een voor vertraagde aflevering, in welk geval het data veld het aantal seconden vertraging zou kunnen transporteren. Dit is slechts een voorbeeld, maar laat ons zien dat het service veld ons vele mogelijkheden oplevert.

Nu kunnen we een aantal functies implementeren om onze data structuren te beheren, vooral om de velden van de berichten te zetten en te krijgen. Deze functies zijn allemaal min of meer gelijk, dus geef ik er hier maar twee, de andere zijn te vinden in de .h files

```
void set_sender(mymsgbuf_t * buf, int sender)
{
  buf->message.sender = sender;
}

int get_sender(mymsgbuf_t * buf)
{
  return(buf->message.sender);
}
```

Het doel van deze functies is niet om de code te beperken (ze bestaan uit slechts 1 regel code): ze zijn er om ons hun bedoeling te herinneren en laten het protocol dichter bij menselijke taal komen, en dus eenvoudiger in gebruik.

Nu moeten we de functies schrijven om IPC keys te genereren, bericht wachtrijen te creëren en te verwijderen, berichten te vesturen en te ontvangen: het bouwen van een IPC key is simpel

```
key_t build_key(char c)
{
  key_t key;
  key = ftok(".", c);
  return(key);
}
```

Then the function to create a queue

```
int create_queue(key_t key)
{
  int qid;

  if((qid = msgget(key, IPC_CREAT | 0660)) == -1){
    perror("msgget");
    exit(1);
  }

  return(qid);
}
```

Zoals je kunt zien is fout beheer in dit geval erg simpel. De volgende code vernietigt een wachtrij

```
int remove_queue(int qid)
{
  if(msgctl(qid, IPC_RMID, 0) == -1)
  {
    perror("msgctl");
    exit(1);
  }
  return(0);
}
```

En tenslotte de functies om berichten te versturen en te ontvangen: een bericht sturen betekend voor ons het schrijven ervan naar een bepaalde wachtrij, bijvoorbeeld degene die ons gegeven is door de switch.

```
int send_message(int qid, mymsgbuf_t *qbuf)
{
  int result, lenght;
  lenght = sizeof(mymsgbuf_t) - sizeof(long);
  if ((result = msgsnd(qid, qbuf, lenght, 0)) == -1){
    perror("msgsnd");
    exit(1);
  }

  return(result);
}

int receive_message(int qid, long type, mymsgbuf_t *qbuf)
{
  int result, length;
  length = sizeof(mymsgbuf_t) - sizeof(long);

  if((result = msgrcv(qid, (struct msgbuf *)qbuf, length, type, IPC_NOWAIT)) == -1){
    if(errno == ENOMSG){
      return(0);
    }
    else{
      perror("msgrcv");
      exit(1);
    }
  }

  return(result);
}
```

Dat is alles. Je kunt de functies vinden in het bestand layer1.h: probeer eens een programma (bijvoorbeeld dat van het vorige artikel) te schrijven met behulp hiervan. In het volgende artikel zullen het hebben over laag 2 van het protocol en deze implementeren.

Aangeraden leesstof

- Silberschatz, Galvin, Gagne, **Operating System Concepts - Sixth Edition**, Wiley&Sons, 2001
- Tanenbaum, WoodHull, **Operating Systems: Design and Implementation - Second Edition**, Prentice Hall, 2000
- Stallings, **Operating Systems - Fourth Edition**, Prentice Hall, 2002
- Bovet, Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000
- The Linux Programmer's Guide: **http://www.tldp.org/LDP/lpg/index.html**
- Linux Kernel 2.4 Internals **http://www.tldp.org/LDP/lki/lki-5.html**
- Webpagina van het #kernelnewbies IRC kanaal **http://www.kernelnewbies.org/**
- De linux-kernel mailing list FAQ **http://www.tux.org/lkml/**

Zoals altijd kun je me commentaar, correcties en vragen sturen op mijn mail adres (leo.giordiani(at)libero.it) of via de Talkback pagina. Schrijf me alsjeblieft in Engels, Duits of Italiaans.

Site onderhouden door het LinuxFocus editors team © Leonardo Giordani "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Vertaling info: en --> -- : Leonardo Giordani <leo.giordani(at)libero.it> en --> nl: Guus Snijders <ghs(at)linuxfocus.org>

2005-01-14, generated by lfparser_pdf version 2.51