LinuxFocus article number 287
http://linuxfocus.org

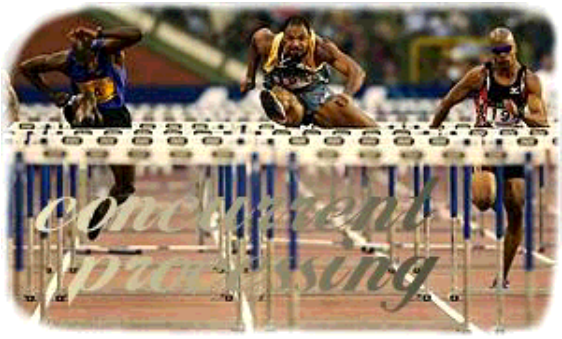
Evenwijdig programmeren - Bericht wachtrijen (1)

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

Over de auteur:

Hij is een student aan de faculteit voor telecommunicatieingenieurs in Politecnico van Milaan, hij werkt als netwerkbeheerder en is geïnteresseerd in programmeren (voornamelijk assembler en c/c++). Sinds 1999 werkt hij bijna uitsluitend nog met Linux/Unix.

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



Kort:

Deze serie artikelen heeft het doel om de lezer 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 we 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)

Het is een goed idee om ook de andere artikelen in deze serie te lezen, welke verschenen in de laatste twee uitgaven van LinuxFocus (November2002 en January2002).

Introductie

In de vorige artikelen hebben we het concept van evenwijdig programmeren geïntroduceert en een eerste oplossing bestudeerd voor het probleem van interproces communicatie: de semaforen. Zoals we zagen, staat het gebruik van semaforen ons toe om de toegang tot gedeelde bronnen te regelen, om zo twee of meer processen ruwweg te synchroniseren.

Het synchroniseren van processen betekend het klokken van hun werk, niet in een absoluut referentie

systeem (het geven van een exacte tijd waarop een proces z'n werk zou moeten beginnen) maar in een relatieve, waar we kunnen opgeven welk proces het eerste zou moeten werken en welke daarna.

Als je hiervoor semaforen gebruikt, blijkt al snel dat dit erg complex en beperkt kan worden: complex omdat ieder proces een semafoor dient te beheren voor ieder ander proces waarmee het zich synchroniseert. Beperkt omdat het ons niet toestaat om parameters tussen de processen uit te wisselen. Stel je bijvoorbeeld het creëren van een nieuw proces voor: deze gebeurtenis zou bekend moeten worden gemaakt bij alle werkende processen, maar semaforen staan een proces niet toe om zulke informatie te versturen.

De controle op de toegang tot gedeelde bronnen door semaforen kan bovendien leiden tot het continue blokkeren van van een proces, als een van de andere betrokken processen een bron vrijgeeft en vervolgens weer reserveerd voordat anderen er gebruik van kunnen maken: zoals we al zagen is het in de wereld van evenwijdig programmeren niet mogelijk om van te voren te weten welk proces wanneer wordt uitgevoerd.

Deze korte notities maken ons duidelijk dat semaforen een inadequate manier zijn om complexe synchronisatie problemen op te lossen. Een elegante oplossing voor dit probleem is het gebruik van bericht rijen (message queues): in dit artikel zullen we de theory van deze interproces communicatie voorziening bestuderen en een klein programma schrijven met behulp van SysV primitieven.

De theory van Bericht Rijen

Ieder proces kan een of meer structuren creëren, genaamd queues (wachtrijen). Iedere structuur kan een of meer berichten bevatten van verschillende typen, welke van verschillende bronnen kan komen en informatie bevatten van iedere natuur; iedereen kan een beicht naar de rij sturen, er van uitgaande dat hij zijn identifier kent. Het proces kan de rij sequentieel benaderen, de berichten in chronologische volgorde lezen (van de oudste, de eerste naar de meest recente, de laatst gearriveerde), maar selectief, dat wil zeggen, alleen berichten van een bepaald type: deze laatste feature geeft ons een soort van controle over de prioriteit van de berichten die we lezen.

Het gebruik van rijen is dus een simpele implementatie van een mail systeem tussen processen: ieder proces heeft een adres waarmee het onderscheiden kan worden van andere processen. Het proces kan dan de berichten die in zijn box zijn afgeleverd lezen in een volgorde naar voorkeur, en aansluitend reageren op wat er werd verteld.

De synchronisatie van twee processen kan dus gebeuren door simpelweg berichten te gebruiken tussen de twee: de gedeelde bronnen zullen nog steeds semaforen bevatten om de processen hun status te laten weten, maar de timing tussen processen zal direct plaatsvinden. We kunnen onmiddelijk zien dat het gebruik van bericht wachtrijen zo het probleem sterk vereenvoudigd, waar het eerst een extreem complex probleem was.

Voordat we de bericht wachtrijen in C kunnen implementeren, moeten we het eerst hebben over een ander probleem, gerelateerd aan synchronisatie: de behoefte aan een communicatie protocol.

Een Protocol Creëren

Een protocol is een set regels die bepalen hoe de interactie tussen elementen in een set plaatsvindt; in het vorige artikel implementeerden we een van de eenvoudigste protocollen door een semafoor te creëren en twee processen te dwingen zich te gedragen naar de status ervan.
Het gebruik van bericht rijen staat ons toe complexere protocollen implementeren: het is voldoende om je voor te stellen dat ieder netwerk protocol (TCP/IP, DNS, SMTP, ...) is gebouwd op een bericht uitwisselings architectuur, zelfs al vindt de communicatie plaats tussen computers en niet intern tussen hen. De verglijking ligt voor de hand: er bestaat geen echt verschil tussen interproces communicatie op dezelfde machine of tussen machines. Zoals we zullen zien in een toekomstig artikel waarin we de concepten waar we over spreken zullen uitbreiden, is een gedistibueerde omgeving (meerdere verbonden computers) een erg eenvoudig onderwerp.

Dit is een erg simpel voorbeeld van een protocol dat gebaseerd is op bericht uitwisseling: twee processen A en B worden gelijktijdig uitgevoerd en verwerken verschillende data; als ze klaar zijn met het verwerken van de data, moeten ze de resultaten samenvoegen. Een eenvoudig protocol om hun interactie te beheersen zou het volgende kunnen zijn

PROCES B:

- Werk met je data
- Zodra je klaar bent, stuur een bericht naar A
- Zodra A antwoord, begin met het sturen van resultaten

PROCES A:

- Werk met je data
- wacht op een bericht van B
- Beantwoord het bericht
- Ontvang data en meng ze met de jouwe

Het bepalen van welk proces de data dient te samenvoegen is in dit geval onbelangrijk; meestal gebeurd dit op basis van de natuur van het betrokken proces (client/server) maar deze discussie verdiend een toegewijd artikel.

Dit protocol is eenvoudig uit te breiden met mogelijkheid van n processen: ieder proces, behalve A, werkt met zijn eigen data en stuurt dan een bericht naar A. Als A antwoord, stuurt ieder proces zijn resultaten: de structuur van de individuele processen (behalve A) is niet gewijzigd.

System V Bericht Wachtrijen

Nu is het tijd om te spreken over de imlementatie van deze concepten in het Linux besturingssysteem. Zoals reeds gezegd, hebben we een set primitieven die ons toestaan om de structuren gerelateerd aan

bericht rijen te beheren. Deze werken als die gegeven om de semaforen te beheren: ik zal dus aannemen dat de lezer bekend is met de basis concepten van proces creatie, het gebruik van system calls en IPC keys.

De structuur aan de basis van het systeem om een bericht te beschrijven is genaamd `msgbuf` ;en wordt gedeclareerd in `linux/msg.h`

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
        long mtype;         /* type of message */
        char mtext[1];      /* message text */
};
```

Het veld mtype representeert het type bericht en is een strikt positief nummer: de overeenkomst tussen nummers en berichten moet van te voren worden aangegeven, en is deel van de protocol definitie. Het tweede veld representeert de inhoud van het bericht maar hoeft niet overwogen te worden in de declaratie. De structuur `msgbuf` kan zo worden geherdefiniëerd dat het complexe data kan bevatten; het bijvoorbeeld mogelijk zo te schrijven

```
struct message {
        long mtype;         /* message type */
        long sender;        /* sender id */
        long receiver;      /* receiver id */
        struct info data;   /* message content */
        ...
};
```

Voordat we de argumenten bekijken die strikt gerelateerd zijn aan de evenwijdigheids theorie moeten we het creëren van het prototype van een bericht met de maximale grootte van 4056 bytes bekijken. Uiteraard is het mogelijk om de kernel te hercompileren om deze dimensie te vergroten, maar dit maakt de applicatie niet-overdraagbaar (niet portable). (deze grootte is overigens vastgesteld om goede performance te garanderen en deze veel groter maken is zeker niet goed).

Om een nieuwe wachtrij te creëren, moet een proces de `msgget()` functie aanroepen:

```
int msgget(key_t key, int msgflg)
```

welke als argumenten de IPC key en wat flags neemt, die gezet kunnen worden met

```
IPC_CREAT | 0660
```

(creëer de wachtrij als hij niet bestaat en geeft toegang aan de eigenaar en de group users), en dit retourneert de wachtrij identifier.

Net als in de vorige artikelen zullen we aannemen dat er geen fouten zullen optreden, zodat we de code eenvoudig kunnen houden, zelfs als we het in een toekomstig artikel zullen hebben over veilige IPC code.

Om een bericht naar een rij te sturen van welke we de identifier kennen, dienen we gebruik te maken van de `msgsnd()` primitieve

```
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg)
```

waarbij geldt dat `msqid` de identifier is van de wachtrij, `msgp` een pointer is naar het bericht dat we hebben verstuurd (het type wordt hier aangeven als `struct msgbuf` maar welke het type is dat we hebben geherdefiniëerd), `msgsz` de dimensie van het bericht (behalve de lengte van het `mtype` dat de lengte is van een long, welke meestal 4 bytes is) en `msgflg` ,een flag die gerelateerd is aan het wacht beleid. De lengte van het bericht kan eenvoudig gevonden worden als

```
length = sizeof(struct message) - sizeof(long);
```

terwijl het wacht beleid refereert aan het geval van een volle rij: als `msgflg` gezet op IPC_NOWAIT zal het zendende proces niet wachten totdat er wat ruimte beschikbaar is en stoppen met een error code; we zullen spreken over zo'n geval als we het hebben over foutbeheer.

Om de berichten te lezen die zich in een rij bevinden, gebruiken we de `msgrcv()` system call

int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg)

waar de `msgp` pointer de buffer aangeeft waar we het gelezen bericht van de rij kopiëren van de rij en `mtype` geeft de subset van berichten aan die we willen behandelen.

Het verwijderen van een wachtrij kan gedaan worden met de `msgctl()` primitieve met de flag IPC_RMID

```
msgctl(qid, IPC_RMID, 0)
```

Laten we eens testen wat we zeiden met een eenvoudig programma dat een berichten wachtrij creëert, een bericht stuurt en deze leest; we zullen de correcte werking van het systeem controleren.

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

/* Redefines the struct msgbuf */
typedef struct mymsgbuf
{
  long mtype;
  int int_num;
  float float_num;
  char ch;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;

  mess_t sent;
```

```
  mess_t received;

  int length;

  /* Initializes the seed of the pseudo-random number generator */
  srand (time (0));

  /* Length of the message */
  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  /* Creates the queue*/
  qid = msgget(msgkey, IPC_CREAT | 0660);

  printf("QID = %d\n", qid);

  /* Builds a message */
  sent.mtype = 1;
  sent.int_num = rand();
  sent.float_num = (float)(rand())/3;
  sent.ch = 'f';

  /* Sends the message */
  msgsnd(qid, &sent, length, 0);
  printf("MESSAGE SENT...\n");

  /* Receives the message */
  msgrcv(qid, &received, length, sent.mtype, 0);
  printf("MESSAGE RECEIVED...\n");

  /* Controls that received and sent messages are equal */
  printf("Integer number = %d (sent %d) -- ", received.int_num,
         sent.int_num);
  if(received.int_num == sent.int_num) printf(" OK\n");
  else printf("ERROR\n");

  printf("Float numero = %f (sent %f) -- ", received.float_num,
         sent.float_num);
  if(received.float_num == sent.float_num) printf(" OK\n");
  else printf("ERROR\n");

  printf("Char = %c (sent %c) -- ", received.ch, sent.ch);
  if(received.ch == sent.ch) printf(" OK\n");
  else printf("ERROR\n");

  /* Destroys the queue */
  msgctl(qid, IPC_RMID, 0);
}
```

Nu kunnen we twee processen creëren en deze laten communiceren door een bericht wachtrij; denk even aan proces forking concepten: de waarde van alle variabelen van het vader proces wordt meegenomen naar het zoon proces (geheugen kopie). Dit betekend dat de wachtrij moeten creëren voordat de fork plaatsvindt, zodat de zoon de wachtrij identifier zal kennen en dus kunnen benaderen.

De code die ik heb geschreven creëert een rij die door het zoon proces wordt gebruikt om zijn data naar de vader te sturen: de zoon genereert random (willekeurige) nummers, stuurt deze naar de vader en beide printen ze uit op de standaard output.

```c
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>
#include <sys/types.h>

/* Redefines the message structure */
typedef struct mymsgbuf
{
  long mtype;
  int num;
} mess_t;

int main()
{
  int qid;
  key_t msgkey;
  pid_t pid;

  mess_t buf;

  int length;
  int cont;

  length = sizeof(mess_t) - sizeof(long);

  msgkey = ftok(".",'m');

  qid = msgget(msgkey, IPC_CREAT | 0660);

  if(!(pid = fork())){
    printf("SON - QID = %d\n", qid);

    srand (time (0));

    for(cont = 0; cont < 10; cont++){
      sleep (rand()%4);
      buf.mtype = 1;
      buf.num = rand()%100;
      msgsnd(qid, &buf, length, 0);
      printf("SON - MESSAGE NUMBER %d: %d\n", cont+1, buf.num);
    }

    return 0;
  }

  printf("FATHER - QID = %d\n", qid);

  for(cont = 0; cont < 10; cont++){
    sleep (rand()%4);
    msgrcv(qid, &buf, length, 1, 0);
    printf("FATHER - MESSAGE NUMBER %d: %d\n", cont+1, buf.num);
  }

  msgctl(qid, IPC_RMID, 0);

  return 0;
}
```

We creëerden dus twee processen die kunnen samenwerken op een elementaire manier, door een systeem van bericht uitwisseling. We hadden geen (formeel) protocol nodig omdat de uitgevoerde operaties erg simpel waren; in het volgende artikel zullen we het weer hebben over bericht wachtrijen en over het beheren van verschillende bericht types. We zullen ook werken aan het communicatie protocol om te beginnen met het bouwen van ons grote IPC project (een telefoon switch simulator).

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
- Web page of the #kernelnewbies IRC channel http://www.kernelnewbies.org/
- The linux-kernel mailing list FAQ http://www.tux.org/lkml/

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