LinuxFocus article number 281
http://linuxfocus.org

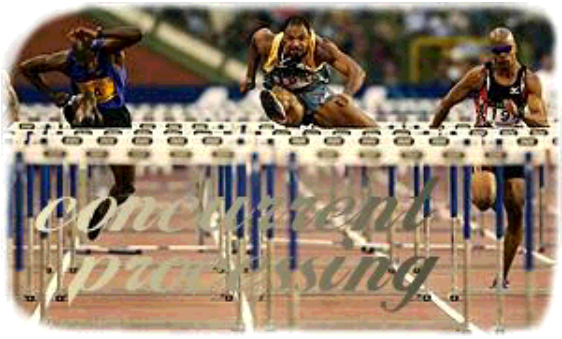
Evenwijdig programmeren - communicatie tussen processen

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

Over de auteur:

Is student aan de faculteit van Telecommunicatie Engineering in Politnecnico in Milaan, werkt als netwerkbeheerder en is geïnteresseerd in programmeren (voornamelijk Assembly en C/C++).

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



Kort:

Deze serie van artikelen heeft tot doel om de lezer te informeren over het concept van multitasking en de implementatie ervan in Linux. Te beginnen bij de theoretische concepten over de basis van multitasken, zullen we eindigen met het schrijven van een complete applicatie om de communicatie tussen processen te demonstreren, met behulp van een eenvoudig maar efficiënt communicatie protocol.

Vereisten voor het begrijpen van het artikel zijn:

- Enige kennis van de shell
- Basis kennis van C (syntax, loops, bibliotheken)

Het valt aan te raden ook het eerste artikel uit deze serie te lezen, daar het een basis is voor deze: November 2002, article 272 .

Introductie

Daar zijn we weer, worstelend met Linux multitasking. Zoals we zagen in het voorgaande artikel, zijn voor het forken van de uitvoering van een programma, een paar regels code voldoende, omdat het besturingssyteem de zorgt voor de initialisatie, beheer en timing van de processen die we creëeren.

Deze service, geleverd door het besturingssysteem is fundamenteel, het is ’de supervisie van proces’ uitvoering; processen worden dus uitgevoerd in een toegewijde omgeving. Verlies van de controle over een proces levert de programmeur een synchronisatie probleem op, samengevat in de vraag: hoe is het

mogelijk om twee onafhankelijke processen samen te laten werken?

Het probleem is complexer dan het lijkt: het is niet alleen een vraag over synchronisatie van de uitvoering van de processen, maar ook over het delen van data, zowel in lees- als in schrijfmodus.

Laten we het eens hebben over het klassieke probleem van concurrente toegang tot data: als twee processen dezelfde dataset lezen, is dit uiteraard geen probleem, en de uitvoering is CONSISTENT. Laat een van de twee processen de dataset aanpassen: de andere zal verschillende resultaten opleveren, afhankelijk van de tijd waarop de dataset wordt gelezen, voor of na het schrijven door het eerste proces. Voorbeeld: we hebben twee processen, "A" en "B", en een integer "d". Proces A verhoogt d met 1, proces B print deze. Geschreven in een meta taal kunnen we het zo weergeven

A { d->d+1 } & B { d->output }

waarbij de "&" de concurrente uitvoering aangeeft. Een eerste mogelijke uitvoering is

(-) d = 5 (A) d = 6 (B) output = 6

maar als proces B eerst wordt uitgevoerd, krijgen we

(-) d = 5 (B) output = 5 (A) d = 6

Je zult begrijpen hoe belangrijk het is om deze situaties correct te behandelen: het risico van INCONSISTENTE data is groot en onacceptabel. Probeer je eens voor te stellen dat de datasets je bankrekening representeren en je zult dit probleem nooit meer onderschatten.

In het voorgaande artikel hebben we reeds gesproken over een eerste vorm van synchronisatie door het gebruik van de waitpid(2) functie, welke een proces laat wachten op het stoppen van een andere, alvorens door te gaan. In feite staat dit ons toe om sommige van de conflicten met betrekking tot het lezen en schrijven van data op te lossen: als de dataset van proces P1 is gedefiniëerd, zal een proces P2, welke met dezelfde dataset werkt of een subset ervan, wachten op het stoppen van P1 alvorens door te gaan met zijn eigen uitvoering.

Het is duidelijk dat deze methode een eerste oplossing biedt, deze is echter nog ver van de beste, daar proces P2 idle dient te blijven, gedurende een periode die erg lang kan zijn, wachtend totdat P1 zijn uitvoering stopt, zelfs als deze geen bewerking meer uitvoert op de data. We moeten onze controle dus verfijnen, oftewel de de toegang tot enkele data of dataset regelen. De oplossing voor dit probleem ligt in een set primitieven van de standaard bibliotheek, bekend onder de naam SysV IPC (System V InterProcess Communicatie, de System V manier van communicatie tussen processen).

SysV keys

Voordat we verder ingaan op de argumenten met betrekking op de concurrenty theory en diens implementaties, zullen we eerst een typische SysV structuur introduceren: IPC keys. Een IPC key (sleutel) is nummer dat gebruikt wordt om een IPC controle structuur (verderop beschreven) te identificeren, maar kan ook gebruikt worden om generieke identifiers te genereren, bijvoorbeeld om

niet-IPC stucturen te organiseren. Een sleutel kan gegenereerd worden met de ftok(3) functie:

key_t ftok(const char *pathname, int proj_id);

welke gebruik maakt van de naam van een bestaand bestand (pathname) en een integer. Het is niet gegarandeerd dat de sleutel uniek is, daar de gebruikte parameters van het bestand (i-node nummer en device (apparaat) nummer) identieke combinaties kunnen opleveren. Een goede oplossing is om een kleine library te creëeren die de gebruikte sleutels bijhoudt en duplicaten voorkomt. which uses the name of an existing file (pathname)

Semaforen

Het idee van een semafoor voor verkeers controle kan gebruikt worden zonder grote modificaties voor data-toegangs-controle. Een semafoor is een bepaalde structuur die een waarde groter of gelijk aan nul bevat en die het beheer voert over een wachtrij of processen die wachten op een bepaalde conditie van de semafoor zelf. Zelfs eenvoudig lijkende semaforen zijn erg krachtig en vergroten consequent de complicaties. Laten we beginnen (als altijd) zonder fout controle: we zullen het opnemen in onze code zodra we bezig gaan met een meer complex programma.

Semaforen kunnen gebruikt worden om de toegang tot bronnen te regelen: de waarde van de semafoor representeerd het aantal processen dat de bron kunnen benaderen; telkens als een proces de bron benaderd zal de waarde van de semafoor afnemen en weer toenemen als de bron wordt losgelaten. Als de bron exclusief is (als slechts een proces toegang kan krijgen) zal de initiële waarde van de semafoor 1 zijn.

Een andere taak kan ook volbracht worden door de semafoor, de bronteller: de waarde representeerd, in dit geval, de hoeveelheid beschikbare bronnen (bijvoorbeeld het aantal vrije geheugen cellen).

Laten we een praktisch voorbeeld nemen, waarbij de semafoor types gebruikt zullen worden: stel dat we een buffer hebben, waarin verchillende S1,...,Sn kunnen schrijven, maar welke alleen een proces L kan lezen; verdere operaties kunnen niet worden gelijktijdig worden uitgevoerd (oftewel, op ieder moment kan slechts een proces bewerkingen uitvoeren op de buffer). Vanzelfsprekend kunnen S processen altijd schrijven, tenzij de buffer vol is, terwijl alleen proces L kan lezen als de buffer niet leeg is. Dus hebben we 3 semaforen nodig: de eerste houdt de toegang tot de buffer bij, de tweede en derde houden bij hoeveel elementen zich in de buffer bevinden (we zullen later zien waarom twee semaforen niet voldoen).

Houd in gedachten dat daar de toegang tot de buffer exclusief is, de eerste semafoor een binaire zal zijn (waarde 1 of 0), terwijl de tweede en derde waardes zullen aannemen die gerelateerd zijn aan de dimensie van de buffer.

Laten we eens kijken hoe semaforen zijn geïmplenteerd in C met gebruik van SysV primitieven. De functie om een semafoor te creëeren is semget(2);

int semget(key_t key, int nsems, int semflg);

waarbij key een IPC sleutel is, nsems het aantal semaforen dat we willen aanmaken en semflg is de toegangscontrole geïmplementeerd met 12 bits, de eerste 3 gerelateerd aan creatie policies (beleiden) en

de andere 9 aan lees en schrijf toegang door de user, groep en other (merk de gelijkenis met het Unix bestandssysteem op); voor een complete beschrijving, zie de man page van ipc(5). Zoals je kunt zien, gebruikt SysV een set van semaforen in plaats van enkelen, resulterend in een compactere code.

Laten we onze eerste semafoor creëeren

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

int main(void)
{
  key_t key;
  int semid;

  key = ftok("/etc/fstab", getpid());

  /* create a semaphore set with only 1 semaphore: */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  return 0;
}
```

Om verder te gaan zullen we moeten leren hoe we semaforen kunnen beheren en verwijderen; het beheer van de semafoor geschied door de primitief semctl(2)

int semctl(int semid, int semnum, int cmd, ...)

welke opereert aansluitend op de actie die wordt aangegeven door cmd op de set semid en (indien gevraagd door de actie) om de enkele semafoor semnum. We zullen enkele opties introduceren zodra deze deze nodig zijn, maar een complete lijst kan gevonden worden in de man page. Afhankelijk van de cmd actie kan het nodig zijn om een ander argument te specificeren voor de actie, van het type

```
union semun {
 int val;                  /* value for SETVAL */
 struct semid_ds *buf;     /* buffer for IPC_STAT, IPC_SET */
 unsigned short *array;    /* array for GETALL, SETALL */
                           /* Linux specific part: */
 struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

Om de waarde van een semafoor te zetten, zou de SETVAL directief gebruikt moeten worden en de waarde moet gespecificeerd worden in de union semun; laten we het voorgaande programma aanpassen om de waarde van de semafoor 1 te maken

```
[...]

  /* create a semaphore set with only 1 semaphore */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);
```

```
[...]
```

Dan moeten we de semafoor loslaten en de structuren die gebruikt werden voor het beheer dealloceren; deze taak gebeurt door de directief IPC_RMID van semctl. Deze directief verwijderd de semafoor en stuurt een bericht naar alle processen die wachten op toegang tot de bron. Een laatste modificatie van het programma is

```
[...]

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* deallocate semaphore */
  semctl(semid, 0, IPC_RMID);

[...]
```

Zoals we reeds eerder hebben gezien is het creëeren en beheren van een structuur voor het beheren van gelijktijdige uitvoering niet moeilijk; zodra we foutbeheer introduceren worden de dingen complexer, maar alleen vanuit het oogpunt van complexiteit van de code.

De semafoor kan nu gebruikt worden door de functie semop(2)

int semop(int semid, struct sembuf *sops, unsigned nsops);

waar semid de set identifier is, sops een array met operaties om uit te voeren en nsops het aantal van deze operaties. Iedere operatie wordt gepresenteerd door een sembuf struct.

unsigned short sem_num; short sem_op; short sem_flg;

Dat wil zeggen, het semafoor nummer in set (sem_num), de operatie (sem_op) en een flag om de wacht policy in te stellen: voor nu laten we sem_flg 0. De operaties die we kunnen specificeren zijn integer nummers en volgen deze regels:

1. sem_op < 0
   Als de absolute waarde van de semafoor groter of gelijk aan die van sem_op is, gaat de operatie en sem_op wordt toegevoegd aan de waarde van de semafoor (eigenlijk is het een afgetrokken, negatief nummer). Als de absolute waarde van sem_op groter is dan de waarde van de semafoor, valt het proces in een slaap staat tot het aantal bronnen beschikbaar is.
2. sem_op = 0
   Het proces slaapt tot de waarde van de semafoor 0 bereikt.
3. sem_op > 0
   De waarde van sem_op wordt toegevoegd aan de waarde van de semafoor, de vorig gebruikte bronnen loslatend.

Het volgende programma probeert weer te geven hoe semaforen gebruikt kunnen worden om het vorige buffer voorbeeld te implementeren: we zullen 5 processen creëeren, genaamd W (writers, schrijfers) en een proces R (reader, lezer). Ieder W proces probeert om de controle over de bron (de buffer) te krijgen, deze te locken door een semafoor en, als de buffer niet vol is, een element er in te plaatsen en de bron weer vrij te geven. Het R proces probeert om de bron te locken, een element uit de buffer te nemen als

de buffer niet leeg is, en de bron weer vrij te geven.

Lezen en schrijven van de buffer zijn slechts virtueel: dit gebeurt omdat, zoals gezien in het vorige artikel, ieder proces zijn eigen geheugen ruimte heeft en niet dat van een ander proces kan benaderen. Dit maakt correct beheer van de buffer met 5 processen onmogelijk, omdat ieder zijn eigen kopie van de buffer ziet. Dit zal veranderen als we het over gedeeld geheugen (shared memory) hebben, maar laten we de dingen stap voor stap leren.

Waarom hebben we 3 semaforen nodig? De eerste (nummer 0) gedraagt zich als een buffer toegangs slot (lock) en heeft een maximale waarde van 1, terwijl de andere 2 de overflow (vollopen) en underflow (leegraken) condities beheren, omdat semop een kant op functioneerd.

Laten we dit toelichten met een semafoor (genaamd 0), waarvan de waarde het aantal lege ruimtes in de buffer weergeeft. Ieder keer als een S proces iets in de buffer plaatst, neemt de waarde van de semafoor met 1 af, totdat de waarde 0 bedraagt, of te wel, totdat de buffer vol is. Deze semafoor kan geen underflow conditie aan: het R proces kan zijn waarde verhogen zonder limiet. We hebben dus een speciale semafoor (genaamd U) nodig, waarvan de waarde het aantal elementen in de buffer weergeeft. Iedere keer dat een W proces een een element in de buffer plaatst, zal de waarde van de semafoor U afnemen en de waarde van de O semafoor toenemen.

De overflow conditie wordt dus geïdentificeerd door de onmogelijkheid van het verlagen van een O semafoor en de underflow conditie door de onmogelijkheid van het verlagen van de U semafoor.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(int argc, char *argv[])
{
  /* IPC */
  pid_t pid;
  key_t key;
  int semid;
  union semun arg;
  struct sembuf lock_res = {0, -1, 0};
  struct sembuf rel_res = {0, 1, 0};
  struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
  struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};

  /* Other */
  int i;

  if(argc < 2){
    printf("Usage: bufdemo [dimensione]\n");
    exit(0);
  }

  /* Semaphores */
  key = ftok("/etc/fstab", getpid());

  /* Create a semaphore set with 3 semaphore */
  semid = semget(key, 3, 0666 | IPC_CREAT);
```

```
  /* Initialize semaphore #0 to 1 - Resource controller */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* Initialize semaphore #1 to buf_length - Overflow controller */
  /* Sem value is 'free space in buffer' */
  arg.val = atol(argv[1]);
  semctl(semid, 1, SETVAL, arg);

  /* Initialize semaphore #2 to buf_length - Underflow controller */
  /* Sem value is 'elements in buffer' */
  arg.val = 0;
  semctl(semid, 2, SETVAL, arg);

  /* Fork */
  for (i = 0; i < 5; i++){
    pid = fork();
    if (!pid){
      for (i = 0; i < 20; i++){
        sleep(rand()%6);
        /* Try to lock resource - sem #0 */
        if (semop(semid, &lock_res, 1) == -1){
          perror("semop:lock_res");
        }
        /* Lock a free space - sem #1 / Put an element - sem #2*/
        if (semop(semid, &push, 2) != -1){
          printf("---> Process:%d\n", getpid());
        }
        else{
          printf("---> Process:%d  BUFFER FULL\n", getpid());
        }
        /* Release resource */
        semop(semid, &rel_res, 1);
      }
      exit(0);
    }
  }

  for (i = 0;i < 100; i++){
    sleep(rand()%3);
    /* Try to lock resource - sem #0 */
    if (semop(semid, &lock_res, 1) == -1){
      perror("semop:lock_res");
    }
    /* Unlock a free space - sem #1 / Get an element - sem #2 */
    if (semop(semid, &pop, 2) != -1){
      printf("<--- Process:%d\n", getpid());
    }
    else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
    /* Release resource */
    semop(semid, &rel_res, 1);
  }

  /* Destroy semaphores */
  semctl(semid, 0, IPC_RMID);

  return 0;
}
```

Laten we de interessantste delen van de code bespreken:

```
struct sembuf lock_res = {0, -1, 0};
struct sembuf rel_res = {0, 1, 0};
struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};
```

Deze 4 regels zijn de acties die we kunnen uitvoeren op onze semafoor set: de eerste 2 zijn enkele acties, terwijl de andere dubbele zijn. De eerste actie, lock_res, probeert de bron te locken: het verlaagt de waarde van de eerste semafoor (nummer 0) met 1 (als de waarde niet nul is) en voert het gegeven beleid uit als de bron bezig is (dat wil zeggen, het proces wacht). De rel_res actie is identiek aan lock_res maar de bron wordt vrijgegeven (de waarde is positief).

De push en pop acties zijn een beetje bijzonder. Het zijn arrays van twee acties, de eerste op semafoor nummer 1 en de tweede op semafoor nummer 2; terwijl de eerste wordt verhoogd, wordt de tweede verlaagd en viceversa, maar het beleid is niet meer om te wachten: IPC_NOWAIT forceert het proces om door te gaan met de uitvoering als de bron bezig is.

```
/* Initialize semaphore #0 to 1 - Resource controller */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);

/* Initialize semaphore #1 to buf_length - Overflow controller */
/* Sem value is 'free space in buffer' */
arg.val = atol(argv[1]);
semctl(semid, 1, SETVAL, arg);

/* Initialize semaphore #2 to buf_length - Underflow controller */
/* Sem value is 'elements in buffer' */
arg.val = 0;
semctl(semid, 2, SETVAL, arg);
```

Hier initializeren we de waarde van de semaforen: de eerste wordt 1 omdat deze de toegang tot een exclusieve bron regelt: de tweede naar de lengte van de buffer (opgegeven op de opdrachtregel) en de derde op 0, zoals gezegd voor het stuk over- en underflow.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
/* Lock a free space - sem #1 / Put an element - sem #2*/
if (semop(semid, &push, 2) != -1){
  printf("---> Process:%d\n", getpid());
}
else{
  printf("---> Process:%d  BUFFER FULL\n", getpid());
}
/* Release resource */
semop(semid, &rel_res, 1);
```

Het W proces probeert de bron te reserveren door de lock_res actie; als dit is gebeurd, voert het een push uit en verteld het op standard output: als de operatie niet kan worden uitgevoerd, drukt het af dat de buffer vol is. Daarna wordt de bron weer vrijgegeven.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
```

```
/* Unlock a free space - sem #1 / Get an element - sem #2 */
if (semop(semid, &pop, 2) != -1){
  printf("<--- Process:%d\n", getpid());
}
else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
/* Release resource */
semop(semid, &rel_res, 1);
```

Het R proces gedraagt zich min of meer als de W processen: reserveerd de bron, voert een pop uit en geeft de bron vrij.

In het volgende artikel zullen we spreken over message queues (wachtrijen voor berichten), een andere structuur voor InterProcess Commnunicatie en synchronisatie. Zoals altijd, als je iets eenvoudigs schijft naar aanleiding van je geleerd hebt van dit artikel, stuur het naar me, met je naar en e-mail adres, ik zal het graag bekijken. Goed werk!

Aanbevolen lezingen

- 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

---

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