LinuxFocus article number 272
http://linuxfocus.org

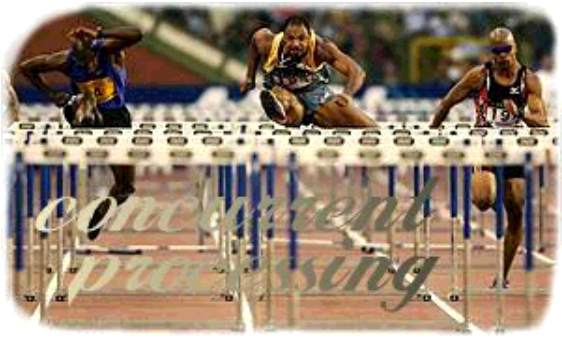
Evenwijdig programmeren - Principes van en kennismaking met processen

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 geÃ¯nteresseerd in programmeren (voornamelijk assembleertaal en c/c++). Sinds 1999 werkt hij bijna uitsluitend nog met Linux/Unix.

Vertaald naar het Nederlands door:
CyberProphet
<cyberprophet/at/linux.be>



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)

Alle referenties naar man pagina's staan tussen accolades na de naam van het commando. Alle glibc functies zijn gedocumenteerd in de infopagina's van gnu (info Libc, of typ info:/libc/top in Konqueror).

inleiding

Eén van de belangrijkste keerpunten in de geschiedenis van besturingssystemen is het concept van mutiprogrammeren, een techniek om de uitvoering van verschillende programma's door erlkaar te weven om een constant gebruik van de bronnen van het systeem te bekomen. Denk eens aan een simpel werkstationnetje, waar een gebruiker terzelfdertijd een tekstverwerker, een geluidsspeler, een printopdracht, een webbrowser en meer kan uitvoeren. Het is een belangrijk concept voor de besturingssystemen van vandaag. Zoals we zullen ontdenken zijn deze lijst van programma's, al zijn ze

dan de meest zichtbare, nog maar een klein deeltje van de programma's die momenteel tegelijkertijd worden uitgevoerd op ons systeem.

Het concept van processen

Om programma's met elkaar te kunnen verweven, wordt een besturingssyteem opmerkelijk gecompliceerder; om conflicten tussen lopende programma's te vermijden, is een onvermijdelijke keuze het resumeren van elk van hen met alle informatie die nodig is voor hun uitvoering.

Voordat we op ontdekking gaan om te zien wat er in onze linuxdoos gebeurt, geef ik je graag eerst wat technische woordenschat: Als we een lopend programma aannemen, op een gegeven tijdstip, dan is de code de reeks instructies waaruit het opgemaakt is, de geheugenruimte is het gedeelte van het machinegeheugen opgenomen door zijn data en de processorstatus is de waarde van de microprocessor's parameters, zoals de vlaggen en de Program Counter (het adres om steeds naar de volgende uit te voeren instructie, te gaan)(Ned.: programma teller).

We definiëren de term LOPEND PROGRAMMA als een serie objecten die gemaakt zijn in code, geheugenruimte en processorstatus. Als gedurende een zeker tijdstip tijdens de uitvoering van de machine, we deze informatie opslaan en vervangen met dezelfde reeks informatie genomen van een ander lopend programma, zal de uitvoering van deze laatste verder gezet worden op het punt waar het was gestopt: Als we dit doen met het eerste programma en daarna met het tweede, voorziet dit in de verweving zoals we hierboven al besproken hebben. De term PROCES (of TAAK) wordt gebruikt om zo'n lopend programma te beschrijven.

Laten we eens bekijken wat er gebeurde met het werkstation waar we in de inleiding over spraken: op elk moment is er enkel Één taak die uitgevoerd wordt (er is enkel maar een microprocessor en het kan geen twee dingen tegelijk doen), en de machine voert delen van zijn code uit; Na een zekere tijd, Quantum genaamd, wordt het lopend proces opgeschort, zijn informatie wordt opgeslagen en vervangen door een ander wachtende taak, wiens code nu voor een quantum van tijd uitgevoerd zal worden, enz. Dit is wat we multitasking noemen.

Zoals al voorheen vermeld: de introductie van multitaksing zorgt vor een reeks problemen, meeste van deze problemen zijn niet triviaal, zoals het de wachtrijindeling van het wachtend proces (schematisering); niettemin dienen ze het te doen met de architectuur van elk besturingssysteem: misschien wordt dit wel het hoofdonderwerp van een volgend artikel, waarin ik misschien enkele delen van de linux kernel code voorstel.

Taken in Linux en Unix

Laten we eens het een en ander ontdekken van taken die op jouw machine lopen. Het commando die ons die informatie levert is ps(1), wat een acroniem is voor "process status". Als je een gewone tekstshell opent en het ps commando invoert, zal je een uitvoer krijgen die op het volgende lijkt:

```
PID TTY          TIME CMD
2241 ttyp4    00:00:00 bash
```

```
 2346 ttyp4    00:00:00 ps
```

Ik heb al eerder vermeld dat deze lijst niet compleet is, maar laten we ons momenteel hierop concentreren: ps heeft ons een lijst gegeven van elke taak die momenteel op de huidige terminal loopt. In de laatste kolom herkennen we de naam welke de taak start (zoals "mozilla" voor de mozilla Web Browser en "gcc" voor de GNU Compiler Collection). Uiteraard verschijnt ook "ps" in de lijst omdat het uitgevoerd werd wanneer de lijst van lopende taken op het scherm werd gedrukt.

Laten we (momenteel) ons niets aantrekken van TIME en TTY en laten we eens kijken naar PID, de Process IDentifier. De PID is een uniek positief nummer (geen nul) die aan elke lopende taak wordt toegewezen; Wanneer de taak afgelopen is kan de PID hergebruikt worden, maar we hebben de garantie dat tijdens de uitvoering de PID niet verandert. Al dit maakt duidelijk dat de afdruk die elk van jullie bekomt door het ps commando in te voeren, verschillend zal zijn van het voorbeeld hierboven. Laten we, om te toetsen of ik de waarheid spreek, nog een shell openen zonder de voorgaande te sluiten en laten we terug het ps-commando invoeren: Deze keer krijgen we dezelfde lijst van taken maar met verschillende PID-nummers, wat getuigt van het feit dat het over twee verschillende taken gaat, al is het programma dan hetzelfde.

We kunnen ook een lijst verkrijgen van alle taken die op onze Linux doos lopen: de man pagina van het ps-commando zegt dat de aanvulling -e betekent dat hij alle taken moet selecteren. Laten we eens "ps -e" in een terminal typen; ps zal een lange lijst zoals hierboven afgebeeld afdrukken. Om deze lijst op een eenvoudige manier te kunnen analiseren; kunnen we het overbrengen naar een logbestandje.

```
ps -e > ps.log
```

Nu kunnen we dit bestand lezen of bewerken met de editor van onze keuze (of simpelweg met het less commando); Zoals in het begin van dit artikel vermeld is het aantal lopende processen hoger dan we zouden verwachten. We merken trouwens op dat de lijst niet alleen taken bevat die wij hebben gestart (door de prompt of de grafische omgeving), maar ook een reeks taken, waarvan sommige met hele vreemde namen: het nummer en de identiteit van de verschenen taken hangen af van de configuratie van je systeem, maar er zijn enkele algemeenheden. Eerst en vooral, het maakt niet uit hoe je je systeem geconfigureerd hebt, de taak met een PID die gelijk is aan 1 is altijd "Init", de vader van alle taken. Het heeft het nummer 1 omdat het altijd de eerste taak is die door het besturingssysteem wordt uitgevoerd. Wat we ook nog kunnen opmerken is het feit dat er heel wat taken aanwezig zijn, wiens naam begint met een "d": dit zijn de zogenaamde "deamons", en zijn sommige van de belangrijkste taken op het systeem. In een volgend artikel zullen we init en de daemons in detail bestuderen.

Multitasking in libc

We begrijpen nu het concept van processen en hoe belangrijk ze zijn voor ons besturingssysteem: nu gaan we verder, en zullen beginnen met het schrijven van multitasking code; van het triviale simultaan uitvoeren van taken gaan we nu verder voorwaarts naar een nieuw probleem: de communicatie tussen gelijktijdig werkende taken en hun synchronisatie; We zullen twee elegante oplossingen voor dit probleem bespreken, boodschappen en semaforen, maar deze laatste zullen in een volgend artikel over de threads in detail uitgelegd worden. Na de boodschappen wordt het tijd om ons programma gebaseerd op deze concepten te schrijven.

De standaard C bibliotheek (libc, geÃ¯mplementeerd in linux met de glibc) gebruikt de V multitasking tools van het unix systeem; het Unix System V (vanaf nu SysV genaamd) is een commerciÃ«le unix implementatie, en is de grondlegger van twee van de belangrijkste unix-families, de andere zijnde BSD unix.

In de libc is het pid_t type gedefiniÃ«erd als een integer die de mogelijkheid bezit om een pid te bevatten. Vanaf nu zullen we het gebruiken om de waarde van een pid te dragen, maar enkel voor de klaarheid van het programma; hetzelfde met het gebruik van integers.

Laten we eens de functie ontdekken die ons de kennis verschaft over het pid die de taak bevat die ons programma doet lopen.

```
pid_t getpid (void)
```

(welke wordt gedefiniÃ«erd met de bibliotheken unistd.h en sys/types.h) en schrijf een programma met als doel het afdrukken van zijn pid. Voer met een editor naar keuze volgende code in

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{
  pid_t pid;

  pid = getpid();
  printf("De pid die aan de taak is toegewezen bedraagt %d\n", pid);
  return 0;
}
```

Sla het programma op als print_pid.c en compileer het

```
gcc -Wall -o print_pid print_pid.c
```

Dit zal een uitvoerbaar bestand aanmaken print_pid genaamd. Ik wil je eraan herinneren dat als de huidige directory niet in het pad staat het noodzakelijk is om het programma te runnen met "./print_pid". Als we het programma uitvoeren zullen we niet verrast worden: Het drukt een positief nummer af, en als het meer dan eens uitgevoerd wordt zal je het nummer zien vermeerderen met een factor 1; Dit is niet standaard zo, omdat een ander taak gecreÃ«erd kan worden van een ander programma tussen twee uitvoeringen van print_pid door. Probeer bijvoorbeeld is ps uit te voeren tussen twee uitvoeringen van print_pid....

Nu wordt het tijd om te leren hoe je taken kan creÃ«ren, maar eerst dien ik nog het een en ander te vertellen over wat er werkelijk gebeurt tijdens deze actie. Wanneer een programma (opgevat in een proces A) een ander proces (B) creÃ«ert, zijn de twee identiek, dat wil zeggen dat ze dezelfde code hebben, het geheugen vol van dezelfde data (maar niet hetzelfde geheugen) en dezelfde processor status. Vanaf dit punt kunnen de twee taken uitgevoerd worden in verschillende richtingen, bijvoorbeeld afhankelijk van de invoer van een gebruik of een of andere data. Taak A is het "Vader proces" terwil B het "zoon-proces" is; Nu kunnen we beter de naam van "vader van alle taken" begrijpen die aan init is gegeven. De functie die een nieuwe taak creÃ«ert is

```
pid_t fork(void)
```

En zijn naam komt van de eigenschap om de uitvoering van het proces vast te nemen. Het nummer die geretourneerd wordt is een pid, maar verdient speciale aandacht. We hebben gezegd dat de huidige taak zichzelf vermenigvuldigt in een vader en een zoon, die zichzelf zullen uitvoeren, verweven met het ander lopend programma, terwijl ze beide verschillend werk doen, maar welk proces zal uitgevoerd worden direct na de vermenigvuldiging, de vader of de zoon? Het antwoord is simpel: Ã©Ã©n van de twee. De beslissing ligt bij een deel van het besturingssysteem die de scheduler wordt genoemd, en die geen aandacht schenkt of de taak nu de vader of de zoon is, het volgt enkel een algoritme gebaseerd op andere parameters.

Hoe dan ook, het is belangrijk te weten welke taak in uitvoering is daar ze beide dezelfde code hebben. Beide taken zullen de code van de vader en de code van de zoon bevatten, maar elk van hen dient maar Ã©Ã©n van deze codes uit te voeren. Kijk even, om een beter zicht op dit concept te hebben, naar volgend algoritme:

```
- FORK
- IF YOU ARE THE SON EXECUTE (...)
- IF YOU ARE THE FATHER EXECUTE (...)
```

wat in een soort van meta taal de code van ons programma voorstelt. Laten we het misterie uit de doeken doen: de fork function retourneert een ’0’ tot naar taak van de zoon en de pid van de zoon naar de vader. Dus is het voldoende om te testen of het pid die geretourneerd wordt een nul is om te weten welk proces die code doet uitvoeren. Als we hetin c-taal vertalen bekomen we

```
int main()
{
  pid_t pid;
  pid = fork();
  if (pid == 0)
  {
    CODE OF THE SON PROCESS
  }
  CODE OF THE FATHER PROCESS
}
```

Het is tijd om het eerste echte voorbeeld van multitasking code te schrijven: je kan het opslaan als een fork_demo.c bestand en het compileren zoals voorheen getoond. De regelnummers zijn er enkel ter verduidelijking. Het programma zal zichzelf forken en zowel de vader als de zoon zullen iets op het scherm afdrukken; De laatste afdruk zal (als alles goed gaat) de verweving zijn van de twee afdrukken.

```
(01) #include <unistd.h>
(02) #include <sys/types.h>
(03) #include <stdio.h>
(04) int main()
(05) {
(05)   pid_t pid;
(06)   int i;

(07)   pid = fork();

(08)   if (pid == 0){
(09)     for (i = 0; i  < 8; i++){
(10)       printf("-SON-\n");
(11)     }
(12)     return(0);
(13)   }
```

```
(14)   for (i = 0; i < 8; i++){
(15)     printf("+FATHER+\n");
(16)   }
(17)   return(0);
(18) }
```

regelnummers (01)-(03) bevatten de includes voor de nodige bibliotheken (standaard I/O), multitasking). De main (zoals altijd in GNU), retourneert een integer, in normale omstandigheden, als het programma het einde zonder fouten bereikt, is dit nul, of een foutmelding als iets misgaat; laten we ervan uit dat alles zonder fouten verloopt (we zullen laten foutbeveiligingen inlassen wanneer de basisconcepten duidelijk zijn). Daarna definieren we het datatype dat een pid bevat (05) en een integer die als teller fungeert in de lussen (06). Deze twee types zijn, zoals voorheen al vermeld, identiek, maar zijn er voor alle duidelijkheid.

Op regel (07) roepen we de fork function aan die een nul zal retourneren naar het programma die in het proces van de zoon wordt uitgevoerd, en het pid van de zoon in het proces van de vader zal retourneren; De test staat op regel (08). Nu zal de code op regels (09)-(13) in het proces van de zoon worden uitgevoerd, terwijl de rest (14)-(16) uitgevoerd wordt in het proces van de vader.
De twee gedeelten drukken simpelweg acht keer het woordje "-son-" of "+father+", wat afhankelijk is van het proces die het uitvoert, en retourneert een nul op het einde. Dit is heel belangrijk, omdat zonder deze laatste "return" het proces van de zoon, wanneer de lus ten einde is, ook de code van de vader zou gaan uitvoeren (probeer het maar, het zal geen schade aan je machine aanbrengen, maar het doet simpelweg niet wat we willen dat het doet). Zo'n fout is heel moeilijk te vinden, daar een uitvoering van een multitasking programma (zeker een complexe) bij elke nieuwe uitvoering verschillende resultaten oplevert, wat het onmogelijk maakt het programma te debuggen op de resultaten.

Misschien zal je geen voldoening hebben in het uitvoeren van dit programma; ik kan je niet verzekeren dat het resultaat een mix tussen de twee strings zal zijn, en dat door de snelheid van de uitvoering van zo'n korte lus. Waarschijnlijk zal de output een reeks geven van "+FATHER+" strings, gevolgd door "-son-"strings of het tegenovergestelde. Probeer dit programma echter meerdere malen uit te voeren en misschien zal het resultaat wel veranderen.

Door een willekeurige vertraging voor elke printf aan te roepen, zouden we misschien een visueler effect van multitasking kunnen krijgen: dit doen we door de sleep en de rand function:

```
sleep(rand()%4)
```

Dit zorgt ervoor dat het programma voor een willekeurig aantal seconden tussen 0 en 3 (het %-teken retourneert het overblijvende gedeelte van de integer deling) slaapt. Nu is de code het volgende:

```
(09)  for (i = 0; i < 8; i++){
(->)    sleep (rand()%4);
(10)    printf("-FIGLIO-\n");
(11)  }
```

en hetzelfde doen we voor de code van de vader. Sla het op als fork_demo2.c, compileer het en voer het uit. Het is nu trager, maar we merken een verschil in de volgorde van afdrukken:

```
[leo@mobile ipc2]$ ./fork_demo2
-SON-
```

```
+FATHER+
+FATHER+
-SON-
-SON-
+FATHER+
+FATHER+
-SON-
-FIGLIO-
+FATHER+
+FATHER+
-SON-
-SON-
-SON-
+FATHER+
+FATHER+
[leo@mobile ipc2]$
```

Laten we nu eens een kijkje nemen naar nieuwe problemen waar we ons voor gesteld zien: We kunnen een zeker aantal taken van de zoon creÃ«ren wanneer we een vaderproces krijgen, zodat de operaties verschillend worden uitgevoerd van die die door het vaderproces worden uitgevoerd in een evenwijdige uitvoeringsomgeving; Vaak dient de vader te communiceren met de zonen of toch tenminste zich synchroniseren met hen, om operaties op het goede moment te kunnen uitvoeren; The wait functie is zo'n manier om tussen taken te kunnen syncrhoniseren.

```
pid_t waitpid (pid_t PID, int *STATUS_PTR, int OPTIONS)
```

De PID is hier de PID van de taak wiens einde wij verwachtende zijn, STATUS_PTR is een pointer naar een integer die de status van de de taak van de zoon zal bevatten (NULL als deze informatie niet nodig is) wat OPTIONS betreft, gaat het om een reeks opties waar we ons op het ogenblik niets van hoeven aan te trekken. Dit is een voorbeeld van een programma waar de vader een zoonproces creÃ«ert en wacht totdat die afgelopen is:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
    for (i = 0; i < 14; i++){
      sleep (rand()%4);
      printf("-SON-\n");
    }
    return 0;
  }
  sleep (rand()%4);
  printf("+FATHER+ Waiting for son's termination...\n");
  waitpid (pid, NULL, 0);
  printf("+FATHER+ ...ended\n");
  return 0;
}
```

De sleep function in de code van de vader is ingelast om uitvoeringen te laten verschillen. Laten we de

code opslaan als fork_demo3.c, het compileren en het uitvoeren. We hebben nu onze eerste multitasking gesynchroniseerde toepassing geschreven.

In het volgende artikel zullen we meer leren over synchronisatie en communicatie tussen taken; schrijf nu programma's die de beschreven functies gebruiken en zend ze naar me op zodat ik enkele van hen kan gebruiken om goede oplossingen of slechte fouten te tonen. Zend me zowel het .c bestand met gecommentarieerde code en een klein tekstbestandje met een beschrijving van het programma, je naam en je e-mail adres.

Recommended readings

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

Vertaling info:
it --> -- : Leonardo Giordani <leo.giordani(at)libero.it>
it --> en: Leonardo Giordani <leo.giordani%28at%29libero.it>
en --> nl: CyberProphet <cyberprophet/at/linux.be>

2005-01-14, generated by lfparser_pdf version 2.51