Concurrent programming – Message queues (3)

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

About the author:

Vor kurzem erhielt ich mein Diplom von der Fakultät für Telekommunikations – Engineering am Politecnico in Milano. Mein Hauptinteresse liegt im Programmieren (Assembler und C/C++). Seit 1999 arbeite ich fast ausschliesslich mit Linux/Unix.

Abstract:

Das ist der letzte Artikel in der Serie über concurrent programming: wir werden den zweiten und letzten Layer unseres Protokolls implementieren, damit erzeugen wir Funktionen, die das Benutzerverhalten auf der Basis des ersten Layers, den wir im vorangegangenen Artikel entwickelten, realisiert.

Es ist sicher auch eine gute Idee, einige der früher erschienenen Artikel dieser Serie gelesen zu haben:

- Concurrent programming – Message Queues (2)
- Concurrent programming – Prinzipien und Einführung in Prozesse
- Concurrent programming – Kommunikation zwischen Prozessen
- Concurrent programming – Message Queues (1)

Protokollausführung – Layer 2 – Allgemeines

Das Programm ipcdemo wurde als einfache Ausführung einer Vermittlungstelle zwischen Benutzern, die versuchen einander Messages zu senden, entwickelt. Um diese Simulation etwas realistischer zu machen, fügte ich das Konzept "Service" ein: die Hauptaufgabe einer Servicemessage (Signalisierung) ist nicht die Datenübermittlung von Benutzer zu Benutzer, sondern die Information über deren Steuerung. Die Service–Message wird von den Benutzern an die Vermittlungstelle geschickt, um mitzuteilen, dass sie noch vorhanden sind, wie sie erreicht werden können (übermitteln der IPC– Queue –ID) und dass sie abbrechen. Zwei weitere Service wurden eingeführt: Abbrechen und Timing. Der erste wird von der Vermittlungstelle benutzt, um dem Benutzer mitzuteilen, dass er abbrechen sollte. Der zweite versucht die Reaktionszeit des Benutzers zu messen. Mehr darüber später in den Abschnitten über Benutzer und Switch.

Layer 2 enthält High−Level−Funktionen zum Senden und Empfangen von Meldungen, um Services anzufordern oder auf Anforderungen anzusprechen, sowie Initialisierungsinformationen: diese Funktionen wurden aus Layer 1−Funktionen aufgebaut, sie sind daher recht einfach zu verstehen. Bitte beachte, dass ich in layer2.h einige Aliase für Meldungstypen (User−Message oder Service−Message) und verschiedene Services (darunter zwei vom Benutzer festzulegende Services zum Experimentieren) deklariere.

Die ipcdemo ist nur ein Beispielcode: er ist nicht optimiert, und wie wir sehen, benutzte ich viele globale Variablen, der Leser sollte sich aber nur auf die IPC−Details und nicht auf den Code konzentrieren. Falls jemand jedoch auf etwas echt Seltsames stößt, bitte an mich schreiben und wir können das diskutieren.

Implementation der User−Prozesse

Der User−Prozess ist schlicht ein Childprozess der Vermittlungstelle (oder besser, des Parentprozesses, der als Vermittlungstelle handelt). Das bedeutet, der User−Prozess hat alle Variablen initialisiert, genau wie der Switch: er kennt z.B. die id der Switch−Queue, weil sie vom switch selber in einer lokale Variablen, vor der Verzweigung (forking operation), gespeichert wurde.

Wenn der User−Prozess seine Arbeit beginnt, sollte er als erstes eine Queue erstellen und der Vermittlungstelle mitteilen, wie diese zu erreichen ist. Um das zu bewerkstelligen, schickt der User−Prozess zwei Servicemeldungen: SERV_BIRTH und SERV_QID.

```
/* Initialize queue  */
qid = init_queue(i);

/* Let the switch know we are alive */
child_send_birth(i, sw);

/* Let the switch know how to reach us */
child_send_qid(i, qid, sw);
```

Dann wird er in der Hauptschleife wirksam: hier schickt der User−Prozess Meldungen, prüft auf eintreffende Meldungen von anderen Usern und ob die Vermittlungstelle einen Service anforderte.

Die Entscheidung, Meldungen zu senden, wird auf Basis der Wahrscheinlichkeit getroffen: die Funktion myrand() liefert eine Zufallszahl, die auf das auszuführende Argument eingestellt ist, in diesem Fall 100. Eine Meldung wird nur gesandt, wenn diese Zahl kleiner als die spezifizierte Wahrscheinlichkeitszahl ist. Da der User−Prozess 1 Sekunde Pause zwischen zwei Schleifendurchläufen macht, folgt daraus, dass der User−Prozess so viele Meldungen schickt, wie die wahrscheinliche Anzahl der Send−Meldungen jede 100 Sekunden, das ist wirklich zu wenig... Wir müssen also darauf achten, die Wahrscheinlichkeitszahl nicht zu niedrig zu setzen oder unsere Simulation wird ewig dauern.

```
if(myrand(100) < send_prob){
  dest = 0;

  /* Do not send messages to the switch, to you, */
  /* and to the same receiver of the previous message */
  while((dest == 0) || (dest == i) || (dest == olddest)){
    dest = myrand(childs + 1);
  }
  olddest = dest;

  printf("%d -- U %d -- Message to user %d\n", (int) time(NULL), i, dest);
  child_send_msg(i, dest, 0, sw);
}
```

Die Meldungen der anderen User sind tatsächlich zur Vermittlungstelle und von dieser an uns gesandt worden; sie sind mit Typ TYPE_CONN (as CONNECTION) gekennzeichnet.

```
/* Check the incoming box for simple messages */
if(child_get_msg(TYPE_CONN, &in)){
  msg_sender = get_sender(&in);
  msg_data = get_data(&in);
  printf("%d -- U %d -- Message from user %d: %d\n",
         (int) time(NULL), i, msg_sender, msg_data);
}
```

Falls die Vermittlungstelle einen Service anfordert, benutzen wir eine Meldung vom Typ TYPE_SERV. Die Servicemeldung müssen wir beantworten. Im Fall der Anschlussbeendigung senden wir eine Bestätigung und die Vermittlungsstelle markiert uns als nicht erreichbar. Danach müssen wir alle übriggebliebenen Meldungen lesen (aus Höflichkeit, wir könnten auch das überspringen), die Queue entfernen und uns von der Simulation verabschieden. Die Anforderung des Zeit−Service, die wir zur Vermittlungstelle schicken, ist eine Meldung mit der aktuellen Zeit: die Vermittlungstelle subtrahiert diese von der Zeit, die sie für das Verbleiben der Meldung in der Queue protokolliert hat. Wie wir feststellen, führen wir auch QoS (Quality of Service) durch − die Simulation ist wahrscheinlich besser als das wirkliche Telefonsystem....

```
/* Check if the switch requested a service */
if(child_get_msg(TYPE_SERV, &in)){
  msg_service = get_service(&in);

  switch(msg_service){
  case SERV_TERM:
    /* Sorry, we have to terminate */
    /* Send an acknowledgement to the switch */
    child_send_death(i, getpid(), sw);

    /* Read the last messages we have in the queue */
    while(child_get_msg(TYPE_CONN, &in)){
      msg_sender = get_sender(&in);
      msg_data = get_data(&in);
      printf("%d -- U %d -- Message from user %d: %d\n",
             (int) time(NULL), i, msg_sender, msg_data);
    }

    /* Remove the queue */
    close_queue(qid);
    printf("%d -- U %d -- Termination\n", (int) time(NULL), i);
    exit(0);
    break;
  case SERV_TIME:
    /* We have to time our work */
    child_send_time(i, sw);
    printf("%d -- U %d -- Timing\n", (int) time(NULL), i);
    break;
  }
}
```

Implementation des Vermittlungstellen−Prozesses

Der Parent−Prozess besteht aus zwei Teilen: vor und nach der Erstellung der Child−Prozesse. Im ersten Teil muss er ein Array initiieren, um die Queue−.I.D. seiner Child−Prozesse zu speichern und seine eigene Queue aufzubauen: das ist sicherlich nicht der wahre Weg, um etwas dieser Art einzuführen, jedoch würde die Einführung von dynamischen Listen in diesem Zusammenhang ausserhalb des Rahmens diese Artikels und

damit nicht nützlich sein. Wünschten wir eine Entwicklung für eine Vielzahl von Verbindungen, sollten wir nicht vergessen, dynamische Strukturen und Speicherallozierung einzusetzen. Die Queue−Identifikatoren werden am Anfang mit dem Wert der Vermittlungstellen−Queue−I.D. initialisiert, d.h. der User ist noch nicht voll da: beendet ein User den Anschluss, wird die Queue−I.D. wieder auf den ursprünglichen Wert gesetzt.

Im zweiten Teil wirkt der Parentprozess als Vermittlungstelle, der wie der User−Prozess eine Schleife durchläuft, bis alle User−Anschlüsse beendet sind. Die Vermittlungstelle prüft auf eintreffende Meldungen von den Usern und leitet sie zu ihrem Ziel.

```
/* Check if some user has connected */
if(switch_get_msg(TYPE_CONN, &in)){

  msg_receiver = get_receiver(&in);
  msg_sender = get_sender(&in);
  msg_data = get_data(&in);

  /* If the destination is alive */
  if(queues[msg_receiver] != sw){

    /* Send a messge to the destination (follow-up the received message) */
    switch_send_msg(msg_sender, msg_data, queues[msg_receiver]);

    printf("%d -- S -- Sender: %d -- Destination: %d\n",
           (int) time(NULL), msg_sender, msg_receiver);
  }
  else{
    /* The destination is not alive */
    printf("%d -- S -- Unreachable destination (Sender: %d - Destination: %d)\n",
          (int) time(NULL), msg_sender, msg_receiver);
  }
```

Sendet ein User jedoch eine Meldung durch die Vermittlungsstelle, kann das Gegenstand einer Serviceanforderung auf Wahrscheinlichkeitsbasis sein (Vorgang wie vorgehend beschrieben); im ersten Fall zwingen wir den User abzubrechen, in zweiten Fall beginnen wir einen Timingablauf: Wir protokollieren die aktuelle Zeit und kennzeichnen den User, um zu verhindern, dass wir einen User behandeln, der sich schon in diesem Prozessablauf befindet. Erhalten wir keine Meldungen, ist es möglich, dass alle User abgehängt sind: in diesem Fall warten wir, bis alle Child−Prozesse wirklich beendet sind (der letzte User könnte noch nach verbleibenden Meldungen in seiner Queue prüfen), entfernen unsere Queue und brechen ab.

```
  /* Randomly request a service to the sender of the last message */
  if((myrand(100)  < death_prob) && (queues[msg_sender] != sw)){
    switch(myrand(2))
      {
      case 0:
    /* The user must terminate */
    printf("%d -- S -- User %d chosen for termination\n",
          (int) time(NULL), msg_sender);
    switch_send_term(i, queues[msg_sender]);
    break;
      case 1:
    /* Check if we are already timing that user */
    if(!timing[msg_sender][0]){
      timing[msg_sender][0] = 1;
      timing[msg_sender][1] = (int) time(NULL);
      printf("%d -- S -- User %d chosen for timing...\n",
            timing[msg_sender][1], msg_sender);
      switch_send_time(queues[msg_sender]);
    }
    break;
      }
```

```
  }
}
else{
  if(deadproc == childs){
    /* All childs have been terminated, just wait for the last to complete its last jobs */
    waitpid(pid, &status, 0);

    /* Remove the switch queue */
    remove_queue(sw);

    /* Terminate the program */
    exit(0);
  }
}
```

Dann schauen wir nach Service−Meldungen: wir können Meldungen über unseren Start, unsere Beendigung, unsere User−I.D. und Auskunft über unseren Time−Service erhalten.

```
if(switch_get_msg(TYPE_SERV, &in)){
  msg_service = get_service(&in);
  msg_sender = get_sender(&in);

  switch(msg_service)
    {
    case SERV_BIRTH:
      /* A new user has connected */
      printf("%d -- S -- Activation of user %d\n", (int) time(NULL), msg_sender);
      break;

    case SERV_DEATH:
      /* The user is terminating */
      printf("%d -- S -- User %d is terminating\n", (int) time(NULL), msg_sender);

      /* Remove its queue from the list */
      queues[msg_sender] = sw;

      /* Remember how many users are dead */
      deadproc++;
    break;

    case SERV_QID:
      /* The user is sending us its queue id */
      msg_data = get_data(&in);
      printf("%d -- S -- Got queue id of user %d: %d\n",
            (int) time(NULL), msg_sender, msg_data);
      queues[msg_sender] = msg_data;
      break;

    case SERV_TIME:
      msg_data = get_data(&in);

      /* Timing informations */
      timing[msg_sender][1] = msg_data - timing[msg_sender][1];

      printf("%d -- S -- Timing of user %d: %d seconds\n",
            (int) time(NULL), msg_sender, timing[msg_sender][1]);
      /* The user is no more under time control */
      timing[msg_sender][0] = 0;
      break;
    }
}
```

Schlussfolgerungen

Wir sind am Ende unserer kleinen Artikelserie über concurring programming: nicht alle Möglichkeiten wurden ausgeschöpft, aber wir haben jetzt einen guten Einblick, was hinter dem Begriff IPC steckt und welche Aufgaben man damit lösen kann. Ich empfehle das kleine Programm, das von mir für diesen Artikel entwickelt wurde, zu erweitern. Wie schon erwähnt, es ist schwierig, Multi−Prozess−Programme zu debuggen, dieses könnte jedoch eine schöne Möglichkeit sein, um unsere Kenntnisse über Debug−Programme zu erweitern (vergessen wir nicht, dass gdb unser bester Freund beim Programmieren ist): in der Liste am Ende dieses Artikels finden wir einige nützliche Programme zum Gebrauch beim Programmieren allgemein.

Ein kurzer Hinweis zu Experimenten mit IPC. Programme funktionieren nicht immer wie gewünscht (das hier beschriebene Programm wurde vielfach ausgeführt...), aber wenn wir Prozesse verzweigen, werden nicht alle durch Anwendung von Ctrl−C gekillt. Ich habe das kill−Programm bisher nicht erwähnt, aber ich vermute, wir verstehen einiges über Prozesse und die Man Pages. Unsere Prozesse lassen jedoch noch etwas anderes nach dem kill−Kommando zurück: IPC−Strukturen. In unserem obigen Beispiel wird das Killen der laufenden Prozesse nicht die Meldungs−Queue deallozieren. Um den gesamten, für unser Experiment allozierten Kernelspeicher zu entleeren, können wir die Programme ipcs und ipcrm benutzen: ipcs gibt eine Liste aller aktiven allozierten IPC−Resourrcen (nicht nur unsere, sondern alle Programme, Vorsicht ist also geboten), während ipcrm uns erlaubt, einige zu entfernen. Führen wir ipcrm ohne Argumente aus, erhalten wir alle benötigten Informationen: die empfohlenen Werte für die ersten Experimente sind "5 70 70".

Um das Projekt zu extrahieren, führen wir "tar xvzf ipcdemo−0.1.tar.gz" aus. Zum Installieren des ipcdemo−Programms führen wir "make" im aktuellen Verzeichnis des Projekts aus. "make clean" beseitigt die Sicherungsdateien und "make cleanall" entfernt Objektdateien.

Schlussbemerkung

Ich möchte mich für die Verzögerung bei der Veröffentlichung diese Artikels entschuldigen, Softwareentwicklung ist glücklicherweise nicht unser einziger Lebensinhalt... Wie stets warte ich auf eure Kommentare zu diesem Artikel und auf Vorschläge für zukünftige Themen: wie wär's denn mit Threads?

Empfohlene Programme, Sites und Lesbares

Empfohlene Bücher finden wir in den vorangegangenen Artikeln, dieses Mal gebe ich euch einige Internetadressen über Programmieren, Debuggen und nützlichen Lesestoff.

Debugger (wie schon erwähnt) sind der beste Freund des Developers, mindestens während der Entwicklung: lernt den Gebrauch von gdb ehe ihr euch mit ddd abgebt, graphischer Kram ist schön, aber nicht unbedingt notwendig.

- GDB Der GNU Project Debugger: www.gnu.org/directory/gdb.html
- DDD Data Display Debugger: www.gnu.org/software/ddd

Habt ihr die aufdringliche "Segmentation fault"− Meldung erhalten und ihr wundert euch, wo ihr fehlerhaften Code geschrieben habt? Zusätzlich zum Lesen der Speicherauszugdatei mit gdb kann das Programm mit valgrind ausgeführt werden, wir sollten auch sein Speichersimulations−Framework nutzen.

- Valgrind Ein Open−Source Memory Debugger für x86−linux: developer.kde.org/~sewardj

Wie wir bemerkten, macht das Schreiben von IPC in C−Sprache Spass, ist aber komplex. Python ist die Lösung: es bietet vollständige Unterstützung für Verzweigung (forking) und anderes, ausserdem ist es erweiterbar in C. Schaut's euch mal an, es lohnt sich.

- Python: http://www.python.org/

Download

- Hier klicken, um zur Download−Seite für diesen Artikel zu gelangen.

Webpages maintained by the LinuxFocus Editor team © Leonardo Giordani "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: en −−> −− : Leonardo Giordani <leo.giordani(at)libero.it> en −−> de: Jürgen Pohl <sept.sapins/at/verizon.net>

2005−01−11, generated by lfparser_pdf version 2.51