LinuxFocus article number 316
http://linuxfocus.org

Concurrent programming - Message queues (3)

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

About the author:

I just received my diploma from the Faculty of Telecommunication Engineering in Politecnico of Milan. Interested in programming (mostly in Assembly and C/C++). Since 1999 works almost only with Linux/Unix.

Abstract:

This is the last article in the series about concurrent programming: here we will implement the second and last layer of our protocol, creating functions that will implement the user's behaviour on the basis of the first layer developed in the past article.

It might be also a good idea to read some of the previous article in this series first:

- Concurrent programming - Message queues (2)
- Concurrent programming - Principles and introduction to processes
- Concurrent programming - Communications between processes
- Concurrent programming - Message queues (1)

Protocol implementation - Layer 2 - General

The ipcdemo program has been developed to be a simple implementation of a switch between users which try to send messages to one another. To add some fun to the simulation I added the concept of "service": a service message (signalling) is a message which main purpose is not to carry user-to-user data, but control information. The service messages will be sent by users to the switch to let it know that they are alive, how to reach them (sending IPC queue id) and that they are terminating. Two more services have been defined: Termination and Timing; the first one is used by the switch to tell the user that it should terminate, the second tries to measure the user's response time. More on this will be discussed later in the user and in the switch section.

Layer number 2 contains high-level functions to send and receive messages, to request and answer services and some initializazion stuff: those functions are built using Layer 1 functions, and thus are really simple to understand. Just notice that I declared in layer2.h some alias to represent message types (user message or service message) and different services (among them two user-defined services for experiments).

The ipcdemo is only a demonstration code: it is not optimized, and you will notice I used many global variables, but this is only to let the reader focus on the IPC stuff and not on code details. Anyway, if you find something really weird, just write me and we will discuss it.

Implementation of the user process

The user is simply a child process of the switch (or, better, of the parent process, which acts as a switch). This means that the user has all variables initialized just like the switch: for example it knows the switch queue id, because it is saved in a local variable by the switch itself before the forking operation.

When the user begins its life, the fist thing it should do is to create a queue and let the switch know how to reach it; to do this the user sends two service messages, SERV_BIRTH and 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);
```

Then it enters the main loop: here the user sends a message, checks for incoming messages from other users and checks if the switch requested a service.

The decision about message sending is taken on a probability basis: the function myrand() returns a random number normalized to the argument passed, in this case 100, and we send a message only if this number is less than the specified probability; since the user sleeps 1 second between two loop executions this means that more or less the user will send as many messages as the send probability every 100 seconds, assuming that 100 extractions are enough to transform probability into reality, which is really too few... Just pay attention not to use too low probabilities or your simulation will run for ages.

```
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);
}
```

The messages from other users are indeed messages that the other users sent to the switch and that the switch sent us, and are marked with the type TYPE_CONN (as CONNECTION).

```
/* 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);
}
```

If the switch requested a service we will use a message marked with the type TYPE_SERV, and we have to answer; in case of termination service we send the switch and acknowledgement message, so it can mark us as unreachable and stop sending us messages; then we have to read all remaining messages (just to be polite, we could also skip this step), remove the queue and say goodbye to the simulation. The time service request that we send to the switch is a message containing the current time: the switch will subtract this from the time it registered when the message was sent to log how many time the message spent in the queues. As you see we are also doing QoS (Quality os Service), so the simulation is probably already better than the actual telephone system...

```
/* 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 of the switch process

The parent process is split into two parts, before and after the creation of the childs. During the first part it has to initialize an array to save the queue id of its childs and to create its own queue: this is surely not the correct way to implement something of this type, but introducing dynamic lists in this context would be out of the scope of the article and after all not useful; anyway, if you plan to develop something accepting any number of connections remember that you have to use dynamic structures and memory allocation. The queue identifiers are at the beginning initialized with the value of the switch's queue id, meaning that the user is not yet alive: when a user terminates the queue id is again set to the original value.

In the second part the parent process acts as a switch, running through a loop just as the user does, until all users are terminated. The switch checks for incoming messages from users and routes them to their destination.

```
/* 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);
  }
```

But if a user sent a message through the switch it can be the object of a service request on a probability basis (working as before); in the first case we force the user to terminate, in the second one we begin a timing operation: we register the current time and mark the user so that we do not try to time a user which is already doing this operation. If we do not receive message it is possible that all users are terminated: in this case we wait that the child processes are really ended (the last user could be checking the remaining messages in its queue), remove our queue and exit.

```
  /* 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
    waitpid(pid, &status, 0);

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

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

Then we check for service messages: we can receive messages about user birth, user termination, user queue id and answers to the time service.

```
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;
        }
}
```

Final considerations

We are at the end of this little series of articles about concurrent programming: not all possibilities have been reviewed but you have now a good idea of what is behind the word IPC and of the problems that it can solve. I recommend to modify the simple program I developed for this article and to extend it; as already said it is difficult to debug multiprocess programs, but this can be a nice occasion to improve your knowledge of debuggers (remember that gdb is your best friend during programming): check the lists at the end of the article to find some interesting programs to use during programming in general.

Just a little advice about IPC experiments. Many times you will run programs that will not work as you wanted (the program above ran many many times...), but when you fork processes just hitting Ctrl-C does not kill all them. I did not mentioned the kill program before, but at this point you know much things about processes and you will understand the man page. But there is another thing your processes will leave behind them after being killed: IPC structures. In the example above if you kill the running processes they will surely not deallocate message queues; in order to clean up the whole kernel memory allocated for our experiments we can use the programs ipcs and ipcrm: ipcs shows a list of the current allocated IPC resources (not only by you but also by other programs, so be careful), while ipcrm lets you remove some of them; if you run ipcrm without arguments you will get all the information you need: suggested numbers for first experiments are "5 70 70".

To extract the project execute "tar xvzf ipcdemo-0.1.tar.gz". To build the ipcdemo program just execute "make" in the project directory; "make clean" clears backup files and "make cleanall" clears object files too.

Conclusion

I want to beg your pardon for the late publication of this article, software development is luckily not the only thing in life... As always I wait for your comments about the article and for suggestions about future topics: what about threads?

Suggested programs, sites and readings

Take a look at the past articles for the reccomended books, this time I will give you some addresses on the Internet about programming, debugging and nice readings.

Debuggers (as already said) are the best friend of a developer, at least during development: learn how to use gdb before ddd, because graphical stuff is nice but not essential.

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

Did you receive the mighty "Segmentation fault" message and are wondering where you wrote wrong code? In addition to reading the core dumped file with gdb you can run the program with valgrind and take advantages from its memory simulation framework.

- Valgrind An open-source memory debugger for x86-linux: developer.kde.org/~sewardj

As you noticed writing IPC in C language is funny but complex. Python is the solution: it has complete support to forking and other stuff plus it is exensible in C. Take a look, it is worth it.

- Python: www.python.org

Download

- Click here to get to the download page for this article