Programación concurrente – Colas de mensajes (3)

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

About the author:

Acabo de recibir mi título de la Facultad de Ingeniería de Telecomunicaciones de la Politécnica de Milan. Interesado en la programación (principalmente en Ensamblador y C/C++). Desde 1999 trabajo casi exclusivamente con Linux/Unix.

Abstract:

Este es el último artículo de la serie sobre programación concurrente: en él implementaremos la segunda y última capa de nuestro protocolo, creando funciones que simularán el comportamiento del usuario basándose en la primera capa desarrollada en el pasado artículo.

Podría ser una buenda idea leerse primero alguno de los artículos anteriores de la serie:

- Programación concurrente – Colas de mensajes (2)
- Programación concurrente – Principios e introducción a los procesos
- Programación concurrente – Comunicación entre procesos
- Programación concurrente – Colas de mensajes (1)

Implementación del protocolo – Capa 2 – General

El programa ipcdemo ha sido desarrollado con el fin de ser una implementación simple de un conmutador entre usuarios que intentan enviarse mensajes entre ellos. Para añadirle diversión a la simulación añadí el concepto de "servicio": un mensaje de servicio (señalización) es un mensaje cuyo propósito principal no consiste en transportar información entre usuarios, sino información de control. Los mensajes de servicio serán enviados al conmutados por los usuarios para hacerle saber que están vivos, cómo alcanzarlos (enviando un identificador de cola IPC) y que están cerrándose. Se han definido dos servicios más: Finalización y temporización; el primero lo utiliza el conmutador para decirle al usuario que debe finalizar, el segundo intenta medir el tiempo de respuesta del usuario. Hablaremos más sobre este tema luego, en la sección de usuario y en la del conmutador.

La capa 2 contiene funciones de alto nivel para enviar y recibir mensajes, para pedir y responder a servicios, y algún material de inicialización: esas funciones se construyen utilizando las de la capa 1 y, por tanto, son realmente fáciles de entender. Nótese que en layer2.h declaré algunos alias para representar tipos de mensajes (mensajes de usuario y mensajes de servicio) y diferentes servicios (entre ellos dos servicios definidos por el usuario para experimentos).

El ipcdemo es sólamente un código de demostración: no está optimizado, y salta a la vista que he empleado muchas variables globales, pero esto es sóolo para permitir que el lector se concentre en la parte de IPC y no en los detalles del código. De cualquier forma, si encuentras algo realmente extraño, simplemente escríbeme y lo comentaremos.

Implementación del proceso de usuario

El usuario es simplemente un proceso hijo del conmutador (o, mejor, del proceso padre, que actúa como un conmutador). Esto significa que el usuario tiene todas las variables inicializadas justamente como el conmutador: por ejemplo, conoce el identificador de la cola del conmutador, porque se guarda en una variable local por parte del propio conmutador antes de la operación fork.

Cuando el usuario comienza su vida, lo primero que debería hacer es crear una cola y permitir que el conmutador conozca cómo llegar a ella; para hacer esto el usuario envía dos mensajes de servicio, SERV_BIRTH y 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);
```

Entonces, entra en el bucle principal: en él, el usuario envía un mensaje, compruba si hay mensajes entrantes de otros usuarios y comprueba si el conmutador solicitó un servicio.

La decisión de enviar mensajes se toma en base a una probabilidad: la función myrand() devuelve un número aleatorio normalizado según el argumento pasado, en este caso 100, y se envía el mensaje sólo si este número es menor que la probabilidad especificada; ya que el usuario se duerme durante 1 segundo entre dos ejecuciones del bucle, más o menos enviará tantos mensajes como la probabilidad de envío cada 100 segundos, asumiendo que 100 extracciones son suficientes para transformar la probabilidad en realidad, las cuales son insuficientes... Simplemente tendremos que poner atención en no utilizar probabilidades demasiado bajas o nuestra simulación se ejecutará durante mucho tiempo.

```
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);
```

```
}
```

Los mensajes de otros usuarios son, de hecho, mensajes que los otros usuarios envían al conmutador y que el conmutador nos envía a nosotros, y están marcados con el tipo TYPE_CONN (como 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);
}
```

Si el conmutador solicitó un servicio utilizaremos un mensaje marcado con el tipo TYPE_SERV, y tenemos que responder; en caso de finalización del servicio le enviaremos al conmutador un mensaje de asentimiento, de forma que nos pueda marcar como inalcanzables y pare de enviarnos mensajes; entonces tenemos que leer todos los mensajes restantes (para guardar las formas, podríamos saltarnos este paso), eliminar la cola y decir adiós a la simulación. La petición del servicio de temporización que enviamos al conmutador es un mensaje que contiene la fecha actual: el conmutador la resta del instante de tiempo en que el mensaje fue enviado para registrar cuánto tiempo ha estado el mensaje en las colas. Como vemos, también estamos haciendo QoS (Calidad de Servicio), de forma que la simulación ya es posiblemente mejor que el actual sistema telefónico...

```
/* 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;
  }
}
```

Implementación del proceso conmutador

El proceso padre se divide en dos partes, antes y después de la creación de los hijos. Durante la primera parte se tiene que inicializar el array para salvar los identificadores de cola de sus hijos y se tiene que crear su

propia cola; seguramente esta no es la forma correcta de implementar algo de este tipo, pero introducir listas dinámicas en este contexto quedaría fuera del alcance de este artículo y, despues de todo, no sería útil; de todas formas, si se planea desarrollar algo que acepte cualquier número de conexiones hay que acordarse de utilizar estructuras dinámicas y reserva de memoria. Los identificadores de las colas son inicializados al principio con el valor del identificador de la cola del conmutador, simbolizando que el usuario no está vivo todavía: cuando un usuario termina se vuelve a establecer el identificador de la cola a su valor original.

En la segunda parte, el proceso padre actúa como un conmutador, ejecutando un bucle tal y como hace el usuario, hasta que todos los usuarios han finalizado. El conmutador comprueba si hay mensajes entrantes procedentes de usuarios y los enruta hacia sus destinos.

```
/* 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);
  }
```

Pero si un usuario envió un mensaje a través del conmutador, puede ser el objeto de una petición de servicio de acuerdo a una probabilidad (funcionando igual que antes); en el primer caso forzamos al usuario a terminar, en el segundo comenzamos una operación de temporización: registramos el tiempo actual y marcamos al usuario de forma que no intentemos temporizar a un usuario que ya esté realizando esta operación. Si no recibimos un mensaje es posible que todos los usuarios hayan terminado: en este caso esperamos que los procesos hijos finalicen realmente (el último usuario podría estar comprobando los mensajes restantes en su cola), eliminamos nuestra cola y salimos.

```
  /* 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);
  }
}
```

Entonces comprobamos la recepción de mensajes de servicio: podemos recibir mensajes sobre el nacimiento de un usuario, la finalización del usuario, el identificador de la cola del usuario y respuestas al servicio de temporización.

```
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;
    }
```

```
}
```

Consideraciones finales

Estamos en el final de esta pequeña serie de artículos sobre la programación concurrente: no se han revisado todas las posibilidades, pero ahora tenemos una buena idea de lo que hay detrás de la palabra IPC y de los problemas que puede solucionar. Recomiendo modificar y ampliar el sencillo programa que he desarrollado en este artículo; como ya hemos dicho, es difícil depurar programas multiproceso, pero esta puede ser una buena ocasión para mejorar nuestros conocimientos sobre depuradores (recordemos que el gdb es nuestro mejor amigo durante la fase de programación): revise las listas al final del artículo para encontrar algún programa interesante para utilizar durante la programación en general.

Sólamente un pequeño consejo sobre los experimentos con IPC. Muchas veces ejecutará programas que no funcionarán como queremos (el programa de antes se ejecutó muchas muchas veces...), pero cuando hacemos forks de procesos, simplemente con apretar Ctrl−C no los matamos todos. Antes no he mencionado nada del programa kill, pero llegados a este punto sabe muchas cosas sobre procesos y entenderá la página del man. Pero hay otra cosa que sus procesos dejarán detrás después de haber sido matados: las estructuras IPC. En el ejemplo de arriba, si mata los procesos que se están ejecutando, seguramente no liberarán las colas de mensajes; para limpiar toda la memoria del núcleo reservada por nuestros experimentos podemos utilizar los programas ipcs e ipcrm: ipcs muestra una lista de los recursos IPC reservados (no sólo por nosotros sino también por otros programas, así que cuidado), mientras que ipcrm permite eliminar alguno de ellos; si ejecutamos ipcrm sin argumentos obtendremos toda la información que necesitamos: números adecuados para los primeros experimentos son "5 70 70".

Para extraer el proyecto ejecutamos "tar xvzf ipcdemo−0.1.tar.gz". Para compilar el programa ipcdemo simplemente tendremos que ejecutar "make" en el directorio del proyecto; "make clean" elimina los ficheros de backup y "make cleanall" elimina también los ficheros objeto.

Conclusión

Quiero pedir perdón por la tardía publicación de este artículo, el desarrollo de software no es, afortunadamente, la única cosa en mi vida... Como siempre espero comentarios sobre el artículo y sugerencias sobre futuros temas: ¿qué tal hilos?

Programas, páginas web y lecturas recomendadas

Para los libros recomendados, mejor échele un vistazo a los artículos anteriores, esta vez daré algunas direcciones de Internet sobre programación, depuración y lecturas interesantes.

Los depuradores (como ya hemos dicho) son los mejores amigos de un desarrollador, al menos durante el desarrollo: aprenda cómo utilizar el gdb antes del ddd, porque el tema gráfico está bien pero no es esencial.

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

¿Ha recibido el poderoso mensaje "Segmentation fault" y se está preguntando dónde escribió el código erróneo? Además de leer los ficheros core volcados con gdb podemos ejecutar el programa con valgrind y sacar ventaja de entorno de simulación de memoria.

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

Tal y como habrá notado, escribir IPC en lenguaje C es divertido pero complicado. Python es la solución: tiene soporte completo para hacer forks así como otras cosas, además de ser extensible en C. Échele un vistazo, vale la pena.

- Python: www.python.org

Descargas

- Pinche aquí para acceder a la página de descargas de este artículo

---

2005−01−10, generated by lfparser_pdf version 2.51