Sistemas Distribuídos - Trabalho 1

Grupo : Marcos Seefelder, Pedro Eusébio

Professor : Daniel Ratton

1- Introdução

O trabalho foi desenvolvido utilizando a linguagem C++. A escolhe dessa linguagem foi baseada na praticidade em tratar eventos de comunicação entre processos. Linguagens como javascript, php, entre outras, oferecerem mais abstração de maneira que obscurecem o que realmente acontece a nível de sistema. Acreditamos que C++ seja uma das linguagens mais apropriadas para o entendimento dos conceitos do trabalho, uma vez que é possível fazer chamadas de sistema direto do código, porém é mais moderno do que C (linguagem na qual são programadas as bibliotecas do linux que implementam as chamadas de sistema utilizadas).

Esse trabalho contém três partes que serão abordadas nas sessões seguintes deste relatório: A primeira relacionada ao tratamento de sinais; A segunda relacionada a pipes; A terceira relacionada a sockets;

Em sequência, apresentamos uma discussão de aspectos envolvendo o desenvolvimento do trabalho, links para o código fonte e referências.

Como fonte de consulta para realizar a implementação das tarefas, consultamos alguns exemplos disponíveis online (os quais referenciamos em comentários nos códigos) e frequentemente acessamos as man pages do Linux [1] para conferir como funcionam as chamadas de sistema (se são bloqueantes ou não, como são implementadas, o que retornam, entre outros detalhes).

Os trechos de código incluídos nesse documento são resumidos de forma a não ocupar muitas linhas.

2 - Sinais

A primeira parte do trabalho consiste de dois programas que fazem tratamento se sinais entre processos.

O primeiro programa consiste na atribuição de um signal handler (definido em uma função de nome `signal_callback()`) a diversos sinais que o programa possa vir a receber, utilizando a função `signal()` importada da biblioteca `signal.h`.

```
void signal_callback(int signum){
  cout << "caught signal" << signum << endl;
  if(signum == 2)
    exit(signum); //sair ao receber Ctrl+C (SIGINT)
  return ;
}

int main (int argc, char const *argv[]) {
  //...
  signal(SIGINT, signal_callback);//declara signal handler
  //...
}
```

O programa recebe como argumento uma flag que deve ser `0` ou `1`, que define se a espera pelo sinal será espera ocupada - busy waiting, na qual o programa ficha em um loop de `while` - ou bloqueante (blocking) - na qual o sistema faz a chamada `pause()` e espera por um sinal de forma bloqueante -, respectivamente.

O segundo programa recebe dois argumentos na sua chamada, um número de processo e um número de sinal. A execução do processo consiste em utilizar a chamada `kill()`, também importada de `signal.h` que permite enviar um sinal a qualquer processo (se for permitido).

3 - Pipes

Na segunda parte, sobre Pipes, temos a implementação do programa Produtor-Consumidor utilizando a comunicação via pipes (anonymous pipes).

O Produtor é representado pela função `producer()` onde serão gerados os números aleatórios que serão enviados através do pipe. A função recebe como parâmetro quantos números serão gerados e por onde deverá passá-los.

A aplicação do Consumidor é implementada por outra função, `consumer()`, onde os números enviados pelo Produtor serão recebidos, verificados se são primos e, por fim, escritos no terminal da aplicação (junto da informação de primalidade). Para essa função, passa-se apenas um parâmetro que representa por onde os números chegarão.

A implementação do pipe é descrita na função main do programa, onde passamos como parâmetro quantos números deverão ser gerados na inicialização do mesmo. Cria-se um pipe com dois valores: Write end, que será usado pelo Produtor; Read end, que será usado pelo Consumidor;

Depois da criação do pipe, criamos um `fork()` do processo atual :

- O processo no qual `fork()` retornou 0 é o processo filho e faz o papel de consumidor;
- O processo no qual `fork()` retornou um valor maior que 0 (*pid* do filho) é o processo pai e faz o papel de produtor;

Antes de cada thread executar uma das dessas funções é necessário fechar o pipe oposto ao que será usado, o processo filho fechará o write end do pipe para poder fazer a leitura dos dados, já o processo pai, fechará o read end para que seja possível realizar a escrita dos números gerados.

4 - Sockets

Na terceira parte do trabalho, desenvolvemos o Produtor-Consumidor utilizando sockets para fazer a comunicação entre processos. A aplicação que faz o papel do consumidor é implementada como um servidor que aceita a conexão de um cliente produtor e trata as mensagens recebidas do mesmo. Por consequência, a aplicação que faz o papel de produtor é implementada como um cliente.

A aplicação do produtor (cliente) recebe como argumentos o host e a porta do servidor, assim como a quantidade (vamos denominar N, no código `messageNum`) de valores a serem gerados. Em seguida, após conectar-se com o consumidor (servidor) gera uma sequência aleatória e crescente de N valores inteiros e os transmite em forma de strings através do socket. A aplicação fica em loop da seguinte maneira, por N iterações: * Gera um número, o envia através do socket e passa a esperar a resposta do consumidor de forma bloqueante; * Com a resposta recebida, imprime na tela qual o valor e se o mesmo é primo. Ao final das iterações, o produtor envia o número 0 e termina.

```
// PRODUTOR
int main(int argc, char const *argv[]) {
     int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);//Inicializa socket IPv4 TCP
     struct sockaddr_in serv_addr = {}; //é preenchida com iformações do server ...
     connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));//conecta ao server
     char buffer[1] = {}; //buffer para receber resposta
     //configura pseudo-random number generator do c++11...
     for (int i = 0; i < messageNum; ++i) {
        std::string number = //gera número aleatório...
        write(sockfd, number.c_str(), number.length());
          recv(sockfd,buffer,1,0); //Espera por resposta de forma bloqueante
        //Lê se é primo (buffer[0]=='1') ou não (buffer[0]=='0') e imprime...
```

```
    }
    //envia 0 para terminar, fecha o socket e retorna
    write(sockfd,"0",1); close(sockfd); return 0;
}
```

A aplicação do consumidor (servidor) recebe como argumento a porta na qual deve trabalhar. Após conectar-se com o produtor (cliente), trata cada mensagem recebida convertendo-a de string para um valor inteiro e, em seguida, verificando se o valor inteiro é um número primo e respondendo através do socket apenas um caractere (‘0’ no caso de não ser primo e ‘1’ no caso de ser primo). A aplicação fica em loop da seguinte maneira: * Espera o recebimento de uma mensagem no socket de forma bloqueante; * Sai do loop caso o número recebido seja igual a 0. Senão, trata a mensagem para verificar a primalidade; * Responde para o produtor o resultado da verificação e passa para a próxima iteração do loop. Ao sair do loop, a aplicação termina.

```
int main(int argc, char const *argv[]) {
    int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);//Inicializa socket IPv4 TCP
    struct sockaddr_in serv_addr = {};//é preenchida com iformações do server ...
     //É necessário atribuir um nome ao socket, com bind(), antes de
     //receber conexões
     bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
    listen(sockfd, 1); //declara o socket como passivo para conexões
    //Prepara para aceitar conexão
    struct sockaddr_in cli_addr;
    socklen_t clilen = sizeof(cli_addr);
     //Aguarda conexão de forma bloqueante e a aceita
    int newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    char buffer[256] = {};//initialize a clear char buffer for receiving messages
     bool keepReading = true; int number = 0;
     while(keepReading)
     {
          recv(newsockfd,buffer,255,0);//espera mensagem de forma bloqueante
          number = //converte mensagem recebida para inteiro
          if (number == 0) keepReading = false; //termina o loop
          if (isPrime(number))
               write(newsockfd,"1",1);//is prime
          else
               write(newsockfd,"0",1);//is not prime
     }
    close(newsockfd); close(sockfd); return 0; //fecha sockets e retorna
}
```

É importante ressaltar que a espera bloqueante por mensagens foi feita através da chamada `recv()`.

5 - Considerações Finais

Verificação de primalidade:

A função de verificação do número Primo foi implementada de forma que pudesse ser aproveitada tanto pela aplicação de Pipes quanto de Sockets. De maneira simples, o funcionamento da verificação pode ser descrita :

- Verifica se o número N é divisível por 2 ou por 3, eliminando assim uma grande quantidade de números possíveis,

- Caso passem por esse teste, começamos a verificar a divisibilidade por outros números a partir do 5.
- Levando em conta que números primos são da forma $K = (6 \times K \pm 1)$ (mas nem todos os números dessa forma são primos), tentamos a divisão por todos os inteiros I dessa forma que encontramos até que $I^2 > K$;
  - Caso seja encontrado um I pelo qual N é divisível, N não é primo;
  - Caso contrário, o é.

```
bool isPrime(int number) {
  if(number == 2) return true; if(number == 3) return true;
  if(number % 2 == 0) return false; if(number % 3 == 0) return false;

  int i = 5; int w = 2;

  while( i * i <= number ) {
    if(number % i == 0) return false;
    i += w; w = 6-w;
  }

  return true;
}
```

Tratamento de exceções e gerador de números aleatórios:

Diferente dos exemplos através dos quais aprendemos os conceitos implementados, fizemos o tratamento de erros utilizando captura de exceções, por ser um método mais robusto para tal. Além disso, aproveitamos o novo sistema de geração de números pseudo-aleatórios de c++11, mais recomendado do que o `rand()` de C.

6 - Código

O código pode ser encontrado no repositório:

https://github.com/mseefelder/COS470-tp1

Ou, caso queira um link para o download direto do código em um zip:

https://github.com/mseefelder/COS470-tp1/archive/master.zip

7 - Referências

[1] Linux man pages: http://linux.die.net/man/ (último acesso 04/2016)