Sistemas Distribuídos – Trabalho 1

A linguagem utilizada para a implementação de IPCs baseados em troca de mensagens foi a C++. A troca de mensagens, uma das formas de comunicação entre dois processos, é realizada pelo sistema operacional através de *system calls,* originadas do processo comunicante. O espaço de memória do kernel é utilizado para viabilizar a troca de mensagens.

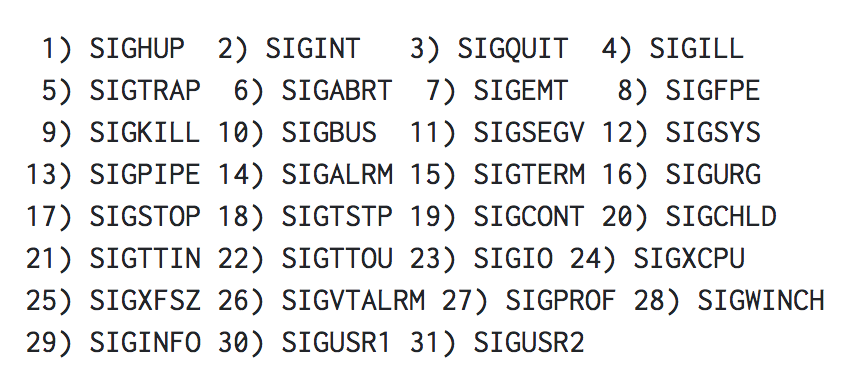
**Primeiro programa – triggerSignal.cpp**

O primeiro programa implementado deveria ser capaz de enviar um sinal a qualquer outro processo.

É útil relatar que cada processo tem um número identificador (PID – *process Identification*) e enquanto o processo estiver em estado *running* (executando) este número é único. O comando do sistema operacional UNIX para procurar o número identificador de um processo é o seguinte:

ps aux | grep (programName)

Todos os sinais têm também um número identificador:



Para a resolução do problema, uma função *main* de um programa em C++ com dois parâmetros foi criada, o primeiro sendo o número identificador do sinal a ser enviado e o segundo o número identificador do processo que irá receber o sinal. Os parâmetros foram usados numa função da biblioteca *signal.h,* que efetivamente envia o sinal ao processo especificado:

kill(processIdentifier, signalIdentifier);

Em caso de falha na função *kill*, a função *main* retorna o parâmetro EXIT\_FAILURE da biblioteca *cstdlib*, indicando erro.

**Segundo programa – handleSignal.cpp**

O segundo programa deveria capturar e reagir a sinais diferentes imprimindo no terminal uma mensagem diferente para cada sinal.

Para a resolução do problema, uma função *main* de um programa em C++ com um parâmetro foi criada, sendo este o modo de espera do próximo sinal (*busy wait ou* *blocking wait*). As primeiras linhas de código da função main fazem chamadas à uma função da biblioteca *signal.h:*

signal(signalIdentifier, signalHandler);

Caso seja enviado ao processo (este programa em execução) o sinal especificado no primeiro parâmetro da função *signal*, a função especificada no segundo parâmetro será executada. Neste programa foram escolhidos 3 sinais a serem capturados, SIGSYS(12), SIGBUS(10) e SIGHUP(1).

A função signalHandler receberá como parâmetro o identificador do sinal, podendo assim imprimir diferentes mensagens independendo do sinal recebido. Além disso, caso capture o sinal SIGHUP, deve encerrar o programa:

void signalHandler(int signalIdentifier) {

cout << "Signal received: " << signalIdentifier << endl;

if (signalIdentifier == 1) {

exit(0);

}

}

Para a implementação dos modos de espera, o *busy wait* consiste num loop infinito, o que significa que o processo continua em estado *running* enquanto aguarda o próximo sinal:

while(1) {

cout << "Busy waiting process number " << getpid() << endl;

}

O *blocking wait* também é um loop infinito, mas a cada iteração faz chamada à função *pause* da biblioteca *unistd*, que coloca o processo em estado *waiting*, liberando espaço para que outro processo utilize a CPU, e colocando-o novamente em estado *running* quando receber outro sinal.

while(1) {

cout << "Blocking waiting process number " << getpid() << endl;

pause();

}

**Terceiro programa – processWithPipe.cpp**

O terceiro programa deveria implementar um programa produtor e um consumidor que se comunicam por meio de um *pipe*. O consumidor deve receber um número aleatório e crescente do produtor e verificar se o mesmo é primo, imprimindo o resultado no terminal.

Primeiramente foram implementadas as funções para retornar um número aleatório e crescente e para retornar se um número é primo ou não. A primeira faz uso das funções *srand* e *rand* da biblioteca *stdlib* para obter um número aleatório que é posteriormente somado a um número referência. Para que o número aleatório seja crescente, a referência deve ser sempre o número anterior. A segunda função tenta dividir um número por todos os números do intervalo 2 até o número que antecede o próprio número. Se alguma divisão proceder, o número não é primo e será primo caso contrário.

A função *main* do programa recebe como parâmetro o número de produtos que o produtor envia ao consumidor. As suas primeiras linhas de código são dedicadas a criação do *pipe*, mecanismo de comunicação unidirecional entre dois processos, onde um lado é responsável pela escrita de dados e o outro lado pela leitura. Para criar o *pipe* é necessário a chamada à função pipe da biblioteca *unistd.h*:

int fd[2];

int pipeStatus = pipe(fd);

Caso a função *pipe* retorne o valor -1, significa que algum erro ocorreu. fd[0] é a extremidade de leitura, que será utilizada pelo consumidor e fd[1] a de escrita, a ser utilizada pelo produtor.

Toda vez que o produtor desejar escrever no *pipe*, ele deve fechar a outra ponta deste (fd[0]) e o consumidor, quando desejar ler, deve fazer a mesma coisa (fd[1]). Isso evitará a leitura de mensagens vazias, incompletas ou fora de ordem.

Para criar os processos produtor e consumidor, foi utilizada a função fork da biblioteca *unist.hd*:

int forkStatus = fork();

Essa função criará um processo pai, correspondente ao atual e um novo, o filho, cópia do processo atual. Ela retornará o valor -1 em caso de erro, o valor 0 quando o processo filho estiver sendo executado e o número identificador do processo filho (ou seja, um valor maior que 0) quando o processo pai estiver sendo executado. Usando essa lógica, puderam ser implementadas as ações do produtor, processo pai, e do consumidor, processo filho. Para escrever no *pipe* o produtor deve converter o número aleatório e crescente para uma string usando a função *sprintf* da biblioteca *studio.h* e depois usar a função *write* da biblioteca *unistd.h*:

int intMessage = getRandomNumber(lowestNumber);

const char \* charMessage = to\_string(intMessage).c\_str();

write(fd[1], charMessage, 20);

O número deve ser convertido para *string* para evitar problemas na representação numérica no *pipe*.

O consumidor deve ler a mensagem usando a função *read* da biblioteca unistd.h e convertê-la para inteiro usando a função *atoi* da biblioteca *stdlib.h*:

read(fd[0], charMessage, 20);

int intMessage = atoi(charMessage);

int prime = isPrime(intMessage);

**Quarto e quinto programas – producer.cpp e consumer.cpp**

O quarto e último programa deveria implementar dois programas, um consumidor e o outro produtor que se comunicassem através de um socket.

A implementação dos programas seguiu a lógica a seguir, fazendo uso de funções das bibliotecas *sys/types.h*, *sys/socket.h* e *netinet/in.h*:



https://i.stack.imgur.com/VUcT6.png

O programa consumidor foi considerado o servidor, que reage a comunicação do cliente, o produtor. A função *main* do programa consumidor recebe como parâmetro a porta na qual a conexão do socket será feita. A do programa produtor recebe como parâmetros a porta de conexão, o nome do servidor e o número de produtos a serem enviados ao consumidor.

**Código:**

<https://github.com/gabrielalucidi/sistemasdistribuidos2018.1>

**Fontes:**

<http://www.yolinux.com/TUTORIALS/C++Signals.html>

<https://www.cyberciti.biz/faq/unix-kill-command-examples/>

<http://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/pipes/pipes.html>

<https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/>

<http://www.dicas-l.com.br/arquivo/programando_socket_em_c++_sem_segredo.php#.WrhYmJPwYzY>