

# Trabalho Prático 1

Sistemas Distribuídos – COS470  
Eduardo Araújo e Renan Basilio

## 1. Introdução

Para a realização do trabalho proposto, foi escolhida a linguagem C/C++. Tal escolha foi feita a partir do baixo nível da linguagem, que permite ao programador acesso mais direto a system calls, acesso o qual foi indispensável para o cumprimento da tarefa proposta. Além disso, devido à complexidade da implementação de certos conceitos no sistema operacional Windows (não existência do conceito de sinais, dificuldade de criar pipes entre dois processos) o sistema operacional escolhido para a execução do trabalho foi baseado em Linux (com implementação através da biblioteca Cygwin para funcionar em Windows).



## 2. Sinais

Para a primeira parte do trabalho foram escritos dois programas.

O primeiro é capaz enviar sinais a outros programas através de uma chamada ao método *kill*, presente na header <signal.h> incluída na linguagem C. O método recebe dois valores inteiros, o primeiro sendo o identificador (PID) do processo ao qual se deseja enviar o sinal, e o segundo o sinal a ser enviado. Internamente, o método realiza uma system call de mesmo nome para que o sistema operacional passe ao receptor o valor do sinal desejado. Ainda, é realizada uma verificação de erro, na qual o método retorna -1 caso o PID fornecido não pertença a um programa em execução.

```
Output - SignalSender (Build, Run) x
Signal Sender v0.1
Running with PID=7195

Enter process ID to send signal to: 9999
Enter signal to send: 1
Sending signal 1 to process 9999...
Could not send signal. Process does not exist.
RUN FAILED (exit value 1, total time: 14s)
```

Fig.1: Enviando sinal a processo inexistente

```
Output - SignalSender (Build, Run) x
Signal Sender v0.1
Running with PID=7253

Enter process ID to send signal to: 7253
Enter signal to send: 1
Sending signal 1 to process 7253...
RUN FAILED (exit value 129, total time: 10s)
```

Fig.2: Enviando sinal 1 (terminate) a si próprio

```
Output - SignalSender (Build, Run) x
Signal Sender v0.1
Running with PID=7321

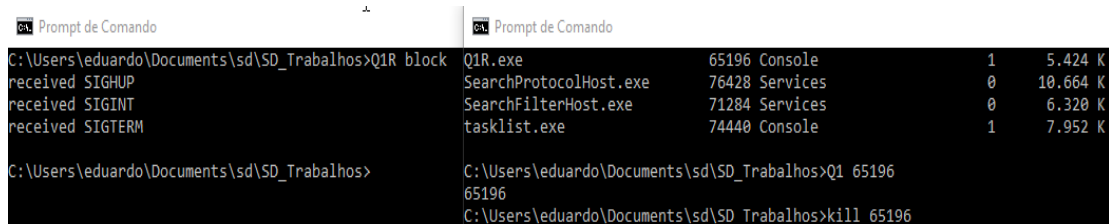
Enter process ID to send signal to: 7321
Enter signal to send: 0
Sending signal 0 to process 7321...
Signal sent successfully.
RUN SUCCESSFUL (total time: 9s)
```

Fig.3: Enviando sinal 0 (NULL) a si próprio.

O segundo é capaz de receber e interpretar os sinais SIGINT (interrupt), SIGHUP (hangup) e SIGTERM (terminate). No caso, foi implementado um handler que imprime na tela o sinal recebido e somente no caso de SIGTERM termina o programa através de uma chamada a *exit(0)*.

Além disso, o programa pode esperar um sinal tanto em blocking quanto em busy wait, o primeiro feito a partir de uma chamada ao método *pause* que bloqueia o programa até receber algum sinal, e o segundo através de um loop infinito vazio na função principal do programa.

Nos nossos testes, o SIGINT foi um sinal enviado pelo teclado (ctrl+c), o SIGHUP for enviado pelo primeiro programa e o SIGTERM pelo comando *kill* na linha de comando, que encerrava o processo.



```
C:\Users\eduardo\Documents\sd\SD_Trabalhos>Q1R block
received SIGHUP
received SIGINT
received SIGTERM

C:\Users\eduardo\Documents\sd\SD_Trabalhos>

C:\Users\eduardo\Documents\sd\SD_Trabalhos>Q1R.exe 65196 Console 1 5.424 K
SearchProtocolHost.exe 76428 Services 0 10.664 K
SearchFilterHost.exe 71284 Services 0 6.320 K
tasklist.exe 74440 Console 1 7.952 K

C:\Users\eduardo\Documents\sd\SD_Trabalhos>Q1 65196
65196
C:\Users\eduardo\Documents\sd\SD_Trabalhos>kill 65196
```

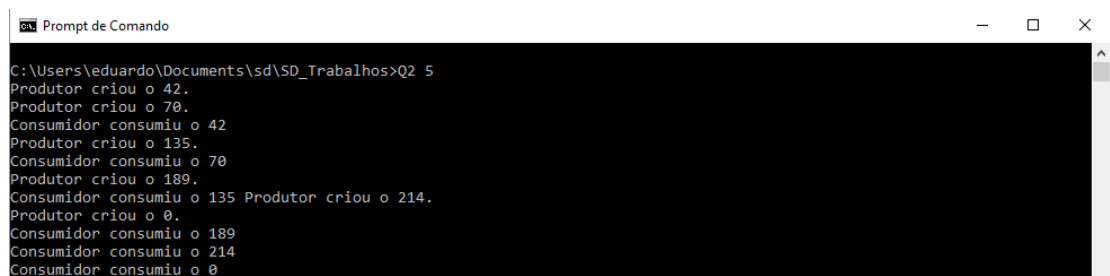
Fig.4: Testes do programa que recebe sinais

### 3. Pipes

A geração de números aleatórios crescentes foi feita somando-se a cada iteração um delta aleatório positivo, obtido por meio da função da rand e da srand com semente dada em função do tempo. A verificação do número ser ou não primo foi realizada por meio de uma função que averigua se o número é divisível apenas por 2 números, “varrendo” os números menores que ele até o número 1, onde a complexidade do algoritmo não foi priorizada, já que não era o objetivo central do estudo.

Na implementação propriamente do pipe, primeiramente criamos um pipe dando como argumento um vetor de dois inteiros, que fazem referência a leitura e a escrita do pipe, dois files descriptors que são correspondentes a entrada e a saída do tubo. Utilizamos também a função fork, que cria um novo “processo filho” e por meio de estruturas condicionais definimos funções que rodariam em cada um dos dois processos, a depender do valor do PID. Assim, definimos uma função produtor e outra consumidor, para serem executadas em cada um dos processos.

Basicamente a função produtor é responsável por escrever na pipe, através da função write, que recebe como parâmetro o file descriptor de escrita, o endereço da variável que contém o número a ser enviado e o tamanho do buffer necessário. Já a função consumidor é responsável por ler os números gerados na função produtor e verificar se estes são primos. A leitura é feita por meio do comando read, semelhante ao write, sendo que para leitura utiliza-se o file descriptor de leitura. Ambos os processos aguardam um tempo aleatório de 1 a 5 segundos entre cada ação de leitura ou escrita. E é provável que os itens sejam produzidos mais rápidos do que consumidos porque há um processamento mais demorado em saber se o número é primo ou não no consumidor.



```
C:\Users\eduardo\Documents\sd\SD_Trabalhos>Q2 5
Produtor criou o 42.
Produtor criou o 70.
Consumidor consumiu o 42
Produtor criou o 135.
Consumidor consumiu o 70
Produtor criou o 189.
Consumidor consumiu o 135 Produtor criou o 214.
Produtor criou o 0.
Consumidor consumiu o 189
Consumidor consumiu o 214
Consumidor consumiu o 0
```

Fig.5: Execução do PIPE com 5 itens produzidos

Mantendo o tempo de espera do Produtor e diminuindo o do consumidor (para 1 segundo de espera entre cada ação de consumo), notamos que há a possibilidade dessa relação sucessiva de consumo e produção ser praticamente de um para um.

```

C:\Users\eduardo\Documents\sd\SD_Trabalhos>Q2 10
Produtor criou o 41.
Consumidor consumiu o 41 - primo
Produtor criou o 86.
Produtor criou o 164. Consumidor consumiu o 86
Consumidor consumiu o 164 Produtor criou o 227.
Produtor criou o 276. Consumidor consumiu o 227 - primo
Consumidor consumiu o 276 Produtor criou o 358.
Produtor criou o 425. Consumidor consumiu o 358
Consumidor consumiu o 425
Produtor criou o 481.
Consumidor consumiu o 481
Produtor criou o 527.
Consumidor consumiu o 527
Produtor criou o 591.
Consumidor consumiu o 591
Produtor criou o 0.
Consumidor consumiu o 0

```

Fig.6: Execução da pipe com consumidor com tempo reduzido de espera

#### 4. Socket

Para esta tarefa o produtor será referido como cliente enquanto o consumidor será o servidor. Em ambos programas é inicializado primeiramente um socket genérico.

1. No servidor esse socket é configurado com as informações locais do sistema no qual ele está executando, e posteriormente o socket configurado é registrado no sistema operacional para receber pacotes na porta requisitada através da operação *bind*. Essa operação retorna um erro caso a porta requisitada esteja em uso. Finalizada essa configuração, é feita a chamada *listen* no socket, e o programa bloqueia até que o sistema operacional receba uma mensagem na porta em que o socket se registrou.
2. No cliente esse socket é configurado com as informações do servidor remoto (IP e porta). No caso utilizamos o endereço de loopback do computador, 127.0.0.1, visto que ambos o cliente e servidor estavam rodando na mesma máquina.

Configurado o socket no cliente, o mesmo faz uma chamada a *connect*, que estabelece uma conexão com o servidor, acordando o mesmo e inicializando a conexão. Ao aceitar o pedido de conexão, o servidor descarta o socket utilizado até o momento e passa a trabalhar com a conexão em si presente no socket.

A partir deste momento o programa é bem semelhante ao que foi feito para a questão anterior; são utilizados os métodos *send* e *recv* para enviar e receber mensagens de um lado ao outro, e ao esgotar o número requisitado de números aleatórios gerados o cliente envia o número 0 que indica ao servidor o fim do processamento, causando com que o mesmo encerre a conexão e feche o socket.

```

Output x
SocketServer (Build, Run) x SocketClient (Build, Run) x
Initializing generic socket...
Socket initialized with handle 3
Configuring socket as client type.
Attempting to connect to server.
Connected to server.
Enter number of random numbers to generate:
5
Generated 83. Server response: Prime
Generated 169. Server response: Not Prime
Generated 246. Server response: Not Prime
Generated 261. Server response: Not Prime
Generated 354. Server response: Not Prime
RUN SUCCESSFUL (total time: 3s)

```

Fig.7: Cliente de socket após execução

```

Output x
SocketServer (Build, Run) x SocketClient (Build, Run) x
Initializing generic socket...
Socket initialized with handle 3
Configuring socket as server type.
Configuration finished successfully.
Listening for incoming connections.
Client connected! Awaiting messages.
Received message from client: 83
Verifying primality...
Number is prime. Sending reply.
Received message from client: 169
Verifying primality...
Number is not prime. Sending reply.
Received message from client: 246
Verifying primality...
Number is not prime. Sending reply.
Received message from client: 261
Verifying primality...
Number is not prime. Sending reply.
Received message from client: 354
Verifying primality...
Number is not prime. Sending reply.
Received message from client: 0
Received number 0. Stopping...
RUN SUCCESSFUL (total time: 6s)

```

Fig.8: Servidor de socket após execução

