

Universidade Federal do Rio de Janeiro



Universidade Federal
do Rio de Janeiro

Escola Politécnica

Sistemas Distribuídos - IPC

Entendendo mecanismos de comunicação

Alunos	Luis Eduardo Pessoa Vinicius Alves
Professor	Daniel Ratton Figueiredo
Horário	Ter e Qui - 10:00-12:00

Rio de Janeiro, 31 de Março de 2019

Conteúdo

1	Objetivo	1
1.1	Github	1
2	Sinais	1
2.1	Decisões de Projeto	1
2.2	Implementação das funcionalidades	1
2.2.1	sinais_sender	1
2.2.2	sinais_receiver	1
2.3	Estudos de caso	3
2.4	Working Proof	3
3	Pipes	4
3.1	Decisões de Projeto	4
3.2	Implementação das funcionalidades	4
3.3	Estudos de caso	5
3.4	Working Proof	6
4	Sockets	6
4.1	Decisões de Projeto	6
4.2	Implementação das funcionalidades	6
4.2.1	server_socket	6
4.2.2	client_socket	7
4.3	Estudos de caso	7
4.4	Working Proof	7
5	Conclusão	8

1 Objetivo

O objetivo deste trabalho é se familiarizar com os principais mecanismos de IPC (Interprocess Communication) baseados em troca de mensagens.

1.1 Github

O código fonte pode ser encontrado no seguinte repositório:

*[https://github.com/vinicius-alves/
Sistemas-Distribuidos-Trabalhos.git](https://github.com/vinicius-alves/Sistemas-Distribuidos-Trabalhos.git)*

2 Sinais

Comunicação Interprocessos usando **sinais**

2.1 Decisões de Projeto

Decidimos usar C++ por se tratar de uma linguagem já conhecida para implementação do exercício.

2.2 Implementação das funcionalidades

Foram escritos dois programas para implementação do exercício: **sinais_sender** e **sinais_receiver**

2.2.1 sinais_sender

É iniciado tendo como *input* o número do processo destino e o *id* do sinal a ser enviado.

Caso nenhum argumento seja passado, o processo termina e retorna uma string especificando esse erro.

Caso o argumento que define o pid seja passado, e não corresponda a nenhum processo, retorna o string que especifica esse erro.

2.2.2 sinais_receiver

É iniciado com um único argumento, que especifica se o processo deve ficar em **busy-wait** (argv=1) ou **blocking-wait** (argv != 1). É perceptível na máquina usada para os testes o aumento do barulho do cooler ao deixar o processo em **busy-wait**.

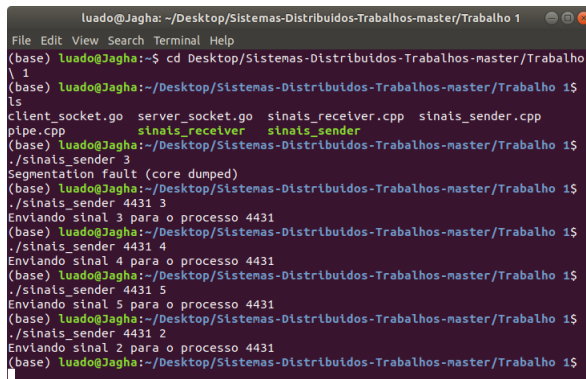
Após iniciado imprime na tela o ***pid*** do processo e o modo de espera escolhido.

2.3 Estudos de caso

Mediante recebimento de sinal, *sinais_receiver* executará uma das instruções definidas no switch:

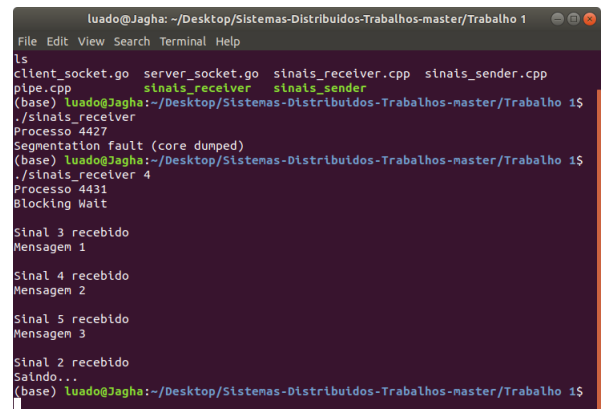
```
switch (signum) {  
    case 2:  
        cout<<"Saindo... " <<endl;  
        exit(1);  
        break;  
    case 3:  
        cout<<"Mensagem 1 " <<endl;  
        break;  
    case 4:  
        cout<<"Mensagem 2 " <<endl;  
        break;  
    case 5:  
        cout<<"Mensagem 3 " <<endl;  
        break;  
}
```

2.4 Working Proof



```
luado@Jagha: ~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1  
File Edit View Search Terminal Help  
(base) luado@Jagha:~$ cd Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho  
1  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
ls  
client_socket.go  server_socket.go  sinais_receiver.cpp  sinais_sender.cpp  
pipe.cpp          sinais_receiver  sinais_sender  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_sender 3  
Segmentation fault (core dumped)  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_sender 4431 3  
Enviando sinal 3 para o processo 4431  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_sender 4431 4  
Enviando sinal 4 para o processo 4431  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_sender 4431 5  
Enviando sinal 5 para o processo 4431  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_sender 4431 2  
Enviando sinal 2 para o processo 4431  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
```

Figura 1: Sinais Sender



```
luado@Jagha: ~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1  
File Edit View Search Terminal Help  
ls  
client_socket.go  server_socket.go  sinais_receiver.cpp  sinais_sender.cpp  
pipe.cpp          sinais_receiver  sinais_sender  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_receiver  
Processo 4427  
Segmentation fault (core dumped)  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$  
./sinais_receiver 4  
Processo 4431  
Blocking Wait  
Sinal 3 recebido  
Mensagem 1  
Sinal 4 recebido  
Mensagem 2  
Sinal 5 recebido  
Mensagem 3  
Sinal 2 recebido  
Saindo...  
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
```

Figura 2: Sinais Receiver

3 Pipes

Comunicação Interprocessos usando **pipes**.

3.1 Decisões de Projeto

Decidimos usar C++ por se tratar de uma linguagem já conhecida para implementação do exercício.

3.2 Implementação das funcionalidades

Duas variáveis são declaradas de início, *pipefd[2]* que armazena os estados do *read-end* e *write-end* do pipe nessa ordem, e *cpid*, variável tipo "signed integer" que armazena o valor do retorno das chamadas de *fork()*.

Três funções foram criadas para serem chamadas dentro da *main()*:

- **"bool e_primo"**: faz as checagens para saber se cada número será primo ou não.
- **"void processo_consumidor"**: Fecha a ponta de escrita do pipe. Consome o que é passado pelo pipe, fazendo a leitura de cada elemento armazenado no pipe, até que se esvazie. Chama *e_primo* para cada elemento lido e retorna o string pertinente para cada saída booleana de *e_primo*. Quando chega ao fim imprime na tela: "Consumidor saindo..."
- **"void processo_produto"**: Fecha a ponta de leitura do pipe. Recebe como parâmetro de entrada a quantidade de números que devem ser produzidos (o default é 1000). Declara a variável *number* que será usada para iterar sobre a quantidade de números pseudo aleatórios a serem produzidos. Dentro do loop, conferimos a cada iteração se a quantidade de números gerada já alcançou o desejado. Se sim, o último número é informado na tela e escrito no pipe. Se não, a variável *number* é atualizada como pede o exercício, e seu valor escrito no pipe. Ao chegar ao fim imprime na tela "Produtor saindo..."

Dentro da função *main()*: um tratamento é feito para quando usuário informa não informa argumentos, de forma que se *argc* \leq 1 o valor da var **qtd_numeros** volta a ser 1000. Cria-se o pipe com *pipe(pipefd)* fazendo o tratamento de erro para se a criação do pipe retornar **-1** (criação do pipe falhou). O *fork* é criado logo após o pipe, permitindo que os processos se

comuniquem lendo e escrevendo na região de memória compartilhada. Um switch é usado de forma que caso **cpid** do processo seja **0** (*child_process*) a função *processo_consumidor* será chamada. Caso seja **-1** mensagem de erro é acionada informando falha na criação do processo filho. O comportamento *Default* é a invocação do *processo_produto*r. Após finalizado o switch, *main()* retorna 0.

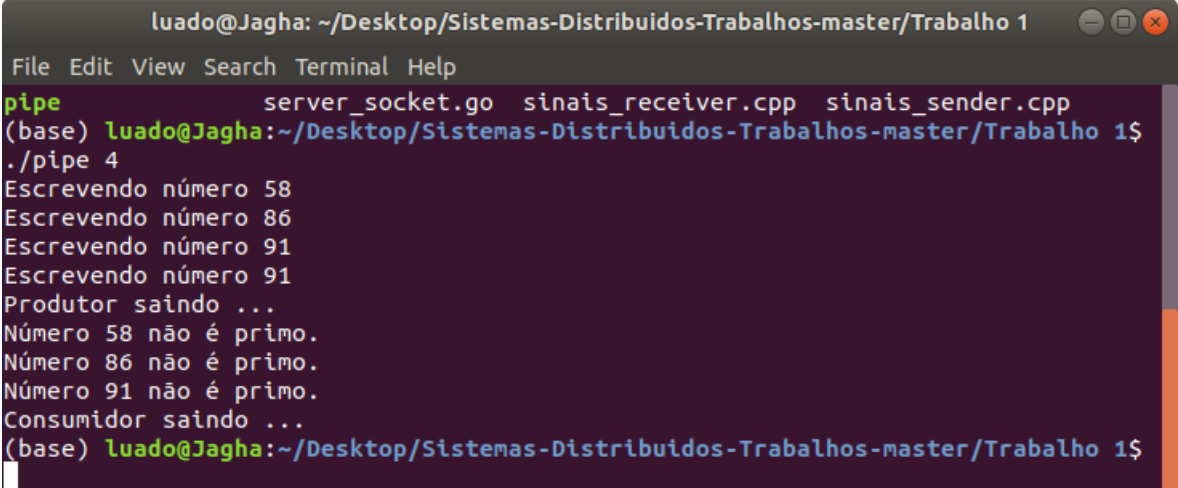
3.3 Estudos de caso

Como a função *read()* recebem como parâmetro o tamanho da mensagem que deve ser lida, não é possível que o *processo_leitor* termine a execução ao encontrar um pipe vazio, de forma que, o programa deve rodar normalmente independente da ordem que os processos pai e filho forem executados, e a arquitetura do pipe possibilitará a comunicação de maneira confiável.

Dentro da *main()* algumas condicionais alteram o comportamento do programa e o fluxo de funcionamento e término:

- Ao ser chamado caso o programa não receba argumentos ($argc \leq 1$), o programa define a quantidade de números a serem gerados como sendo o default (*1000 números*).
- Caso o pipe não possa ser criado, (*retorno = -1*), o programa encerra e produz a string de erro que comunica essa falha de criação.
- Após a criação do fork bem sucedida, dois processos pai e filho executarão em sincronia, e um switch é usado para que o código certo execute em cada processo. Medida necessária dado que um fork cria processos com código idêntico. Para esse switch três casos foram usados, atendendo o caso que o processo filho não pode ser criado (*retorno da função fork = -1*), caso em que deve ser executado o processo_consumidor (*retorno do fork = 0*) e o caso default, onde quem deve ser executado é o processo_produtor (*default*).

3.4 Working Proof



```
luado@Jagha: ~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1
File Edit View Search Terminal Help
pipe server_socket.go sinais_receiver.cpp sinais_sender.cpp
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
./pipe 4
Escrevendo número 58
Escrevendo número 86
Escrevendo número 91
Escrevendo número 91
Produtor saindo ...
Número 58 não é primo.
Número 86 não é primo.
Número 91 não é primo.
Consumidor saindo ...
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
```

Figura 3: Pipe Piping Away

4 Sockets

Comunicação Interprocessos usando **sockets**

4.1 Decisões de Projeto

Decidimos usar Golang por se tratar de uma linguagem construída para facilitar a implementação de paralelização de processos, e com grande potencial de crescimento.

4.2 Implementação das funcionalidades

4.2.1 server_socket

O programa *server_socket* possui três funções implementadas além da função *main()*:

- **SocketServer:** Responsável por criar o lado servidor. Usa a declaração *defer* que garante que *listen.Close()* só fechará a conexão após todo o resto do código dentro da função que cria o socket for executado. É a única função chamada pela *main()*. Recebe como argumento a porta onde o socket será criado, definida dentro do código da função *main()*.

- **handler:** É chamada dentro da função que cria o socket. Cria um buffer que lê os inputs recebidos e envia, com uso do *package bufio* as respostas para o *client_server*.
- **isPrime:** Implementa a função que avalia o número recebido pelo *server_socket*.

4.2.2 client_socket

O programa *client_socket* possui uma função implementada além da função *main()*:

- **SocketClient:** Responsável por criar o lado cliente. Usa a declaração *defer* que garante que *conn.Close()* só fechará a conexão após todo o resto do código dentro da função for executado, prevenindo que a conexão se feche antes do devido momento. É a única função definida pelo usuário chamada pela *main()*. Recebe como argumento **ip** e **porta** do *socket_server*, além do número de iterações de envio *N* definidos dentro do código da função *main()*.

4.3 Estudos de caso

Caso haja algum erro durante a criação do socket o programa retorna o string relativo a esse erro. Caso não, o programa segue, e uma execução com a geração de **N** números segue como mostram as figuras abaixo, onde **N = 4**:

4.4 Working Proof

```

luado@Jagha: ~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1
File Edit View Search Terminal Help
(base) luado@Jagha:~$ cd Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho
\ 1
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
ls
client_socket.go  server_socket.go  sinais_sender.cpp
pipe.cpp          sinais_receiver.cpp
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
go run server_socket.go
2019/04/02 08:09:05 Consumidor iniciando
2019/04/02 08:09:05 Ouvindo porta: 3333
2019/04/02 08:10:10 Recebido: 23
- Enviado: false
2019/04/02 08:10:10
2019/04/02 08:10:10 Recebido: 100 - Enviado: false
2019/04/02 08:10:10
2019/04/02 08:10:10 Recebido: 118 - Enviado: false
2019/04/02 08:10:10
2019/04/02 08:10:10 Recebido: -1 - Enviado: false
2019/04/02 08:10:10
2019/04/02 08:10:10 Consumidor saindo...
exit status 2
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$

```

Figura 4: Server Side

```

luado@Jagha: ~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1
File Edit View Search Terminal Help
(base) luado@Jagha:~$ cd Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho
\ 1
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
ls
client_socket.go  server_socket.go  sinais_sender.cpp
pipe.cpp          sinais_receiver.cpp
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$
go run client_socket.go
2019/04/02 08:10:10 Produtor iniciando
2019/04/02 08:10:10 Enviou: 23 - Recebeu false
2019/04/02 08:10:10
2019/04/02 08:10:10 Enviou: 100 - Recebeu false
2019/04/02 08:10:10
2019/04/02 08:10:10 Enviou: 118 - Recebeu false
2019/04/02 08:10:10
2019/04/02 08:10:10 Enviou: -1 - Recebeu false
2019/04/02 08:10:10
2019/04/02 08:10:10 Produtor saindo...
(base) luado@Jagha:~/Desktop/Sistemas-Distribuidos-Trabalhos-master/Trabalho 1$

```

Figura 5: Client Side

5 Conclusão

As diferentes implemtações de cada mecanismos de implementação entre processos, deixa claro que cada um tem seu papel. Enquanto um sinal é rápido ele é muito específico e não transmite mensagens como um pipe poderia fazer. Mas um pipe enquanto bom transmissor de mensagens não é o suficiente para estabelecermos comunicação entre processos de máquinas distintas, conectadas por rede.