

Universidade de Brasília
Engenharia de Software
Professor: Fernando William Cruz
Aluno: Luan Guimarães Lacerda
Matrícula: 12/0125773

Implementação da comunicação entre processos independentes usando os mecanismos de filas de mensagens, sockets e memória compartilhada

Github: <https://github.com/luanguimaraesla/unix-messenger>

12 de abril de 2016

0. Objetivos

O objetivo desse trabalho é a implementação da comunicação entre processos independentes usando os mecanismos de filas de mensagens, sockets e memória compartilhada. Levando em consideração a elaboração eficiente de uma esquemática na qual permita o pleno funcionamento de vários desses recursos em conjunto de forma que, ao final do projeto, o conhecimento de tais estruturas seja sólido e passível de aplicações em projetos futuros.

1. Desenvolvimento

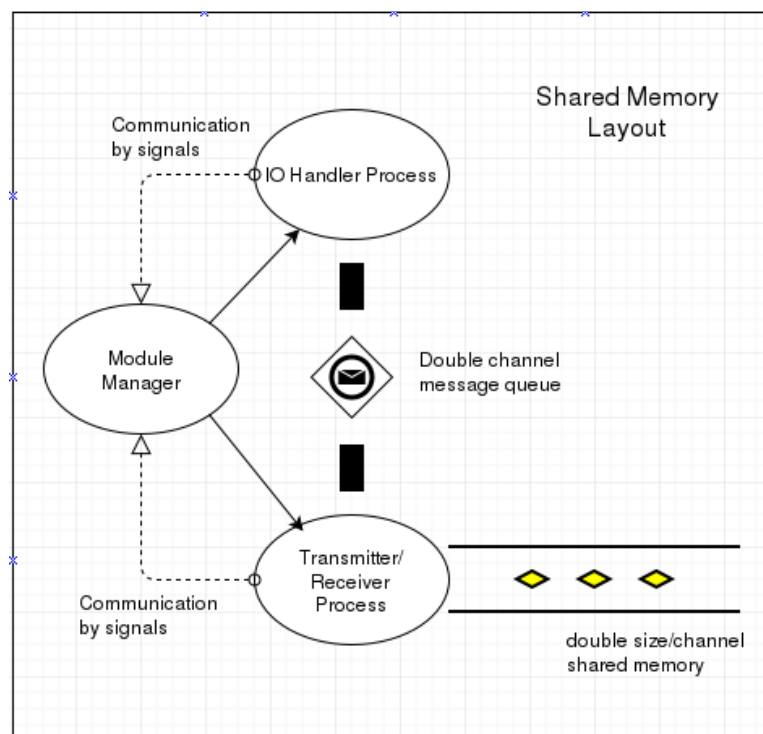
As aplicações foram desenvolvidas utilizando a linguagem C o que aproximou o software de camadas dos modelos reais implementados fazendo-se o uso de funções do próprio sistema operacional.

Diversas estratégias foram estudadas ao longo das atividades. Várias delas podem ser encontradas nos arquivos das pastas “shared_memory/learning” e “socket/learning”. Entretanto, apenas aquelas que se encaixaram de forma adequada no escopo do projeto foram escolhidas para compor o software.

Bem sucedidas, foram as escolhas tomadas, levando a conclusão da implementação duplex tanto utilizando memória compartilhada como socket, cada uma delas com soluções apropriadas para gerir a complexa trama de processos e threads.

1.1 Memória Compartilhada

A solução apresentada contém 3 processos em cada módulo criado. Isto é, além dos dois processos da camada N e N-1, optou-se por instanciar outro cuja função seria facilitar o gerenciamento da comunicação e paralelismo entre os outros dois, possibilitando a manutenção eficaz de casos de prioridade, eventos assíncronos, falhas inesperadas, etc, sem necessariamente prejudicar o fluxo de cada um dos filhos. Segue a esquemática elaborada:



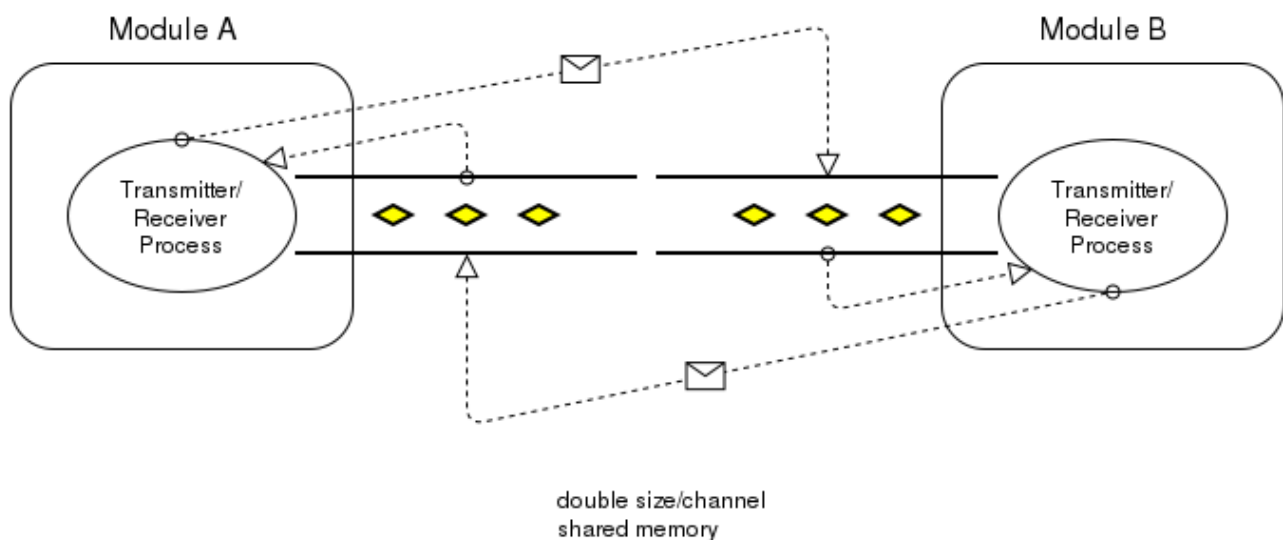
O procedimento de leitura e escrita é simple, e será a base para o entendimento tanto da aplicação que utiliza memória compartilhada, como da aplicação que utiliza socket. Simularemos passo-a-passo uma situação de inicialização de um módulo para seguir em uma linha de raciocínio construtivista.

1. A inicialização se dá pela chamada de função “void init_messenger_module(void)”, na qual primeiramente 4 sinais de controle são definidos no processo Module Manager, são eles:

- a) SIGNAL_MESSAGE_TO_TRANSMIT
Sinal de alerta que o processo IO Handler emite assim que escreve uma mensagem na fila.
- b) SIGNAL_MESSAGE_TO_WRITE
Sinal de alerta que o processo Transmitter/Receiver emite assim que uma mensagem da memória compartilhada é escrita na fila de mensagens.
- c) SIGNAL_TO_FINISH
Sinal de controle que informa ao módulo de gerenciamento que o usuário deseja finalizar o programa.
- d) SIGNAL_TO_KILL EVERYTHING
Sinal definido para que, em caso de algum erro, o usuário possa invocar um “kill -12 <pid>”, forçando a finalização correta do programa.

2. Uma fila de mensagens é então inicializada com dois canais de comunicação, RECEIVE_CHANNEL, no qual o processo Transmitter/Receiver escreve mensagens que chegaram a partir da memória compartilhada e, SEND_CHANNEL, no qual o processo IO Handler envia as mensagens que o usuário digitou para serem transmitidas.

3. Uma memória compartilhada é então criada. Nessa etapa, optou-se por criar uma memória de tamanho 2 x MSG_SIZE, de forma que os processos Transmitter/Receiver pudessem ler e escrever ao mesmo tempo, sem necessariamente, ter que disputar a escrita de um segmento.



4. Duas chamadas `fork()` são realizadas para criar os processos IO Handler e Transmitter/Receiver, cada um deles inicialmente define dois sinais pelos quais serão informados de que algum evento ocorreu.

No processo IO Handler:

a) `SIGNAL_MESSAGE_TO_WRITE`

Sinal que informa o processo que há mensagens na fila aguardando para serem exibidas para o usuário.

b) `SIGNAL_TO_FINISH`

Sinal que informa o processo IO Handler para se auto-finalizar.

No processo Transmitter/Receiver:

a) `SIGNAL_MESSAGE_TO_TRANSMIT`

Sinal que informa o processo de que há mensagens na fila para serem transmitidas através da memória compartilhada.

b) `SIGNAL_TO_FINISH`

Faz o mesmo que no processo IO Handler

5. Cada um desses dois processos entra em um laço infinito de acordo com sua função:

IO Handler:

Escaneia o teclado e então, assim que recebe uma mensagem de entrada, a escreve na fila de mensagens, informa ao Manager que existe uma nova mensagem para ser transmitida e volta a escanear o teclado.

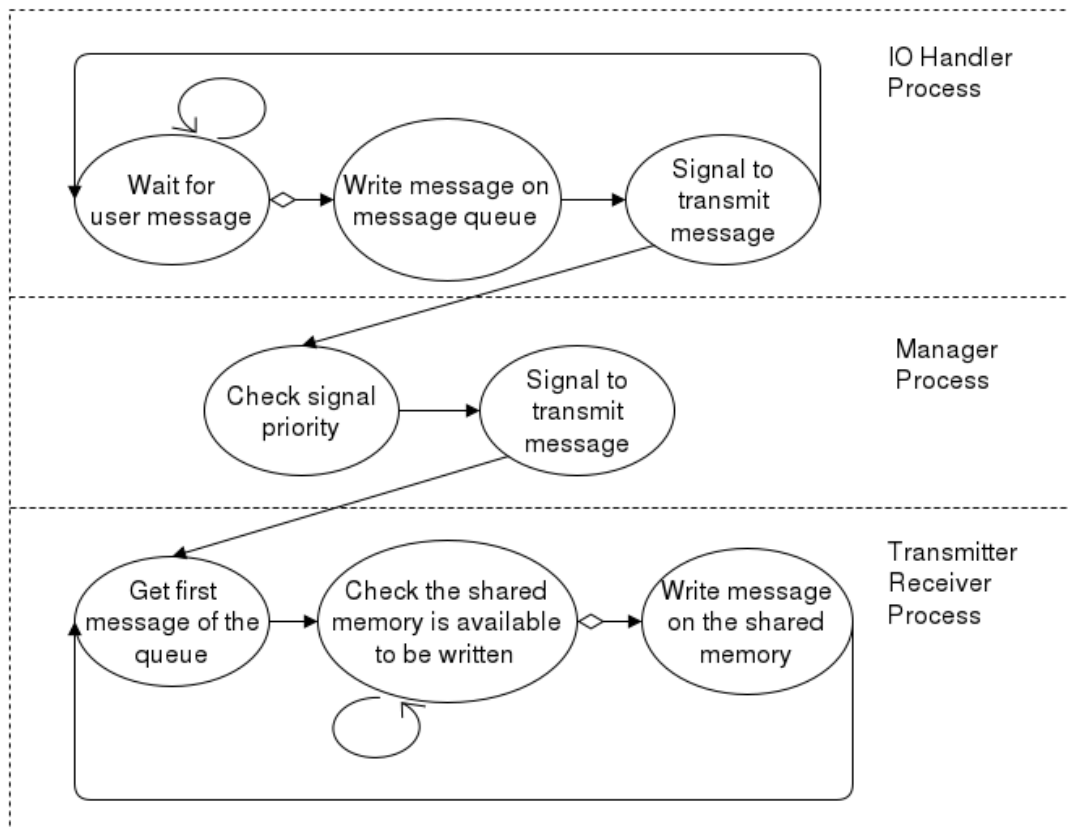
Eventualmente, quando acionado pelo Manager via sinal, recupera uma mensagem da fila e imprime para que o usuário veja.

Transmitter/Receiver:

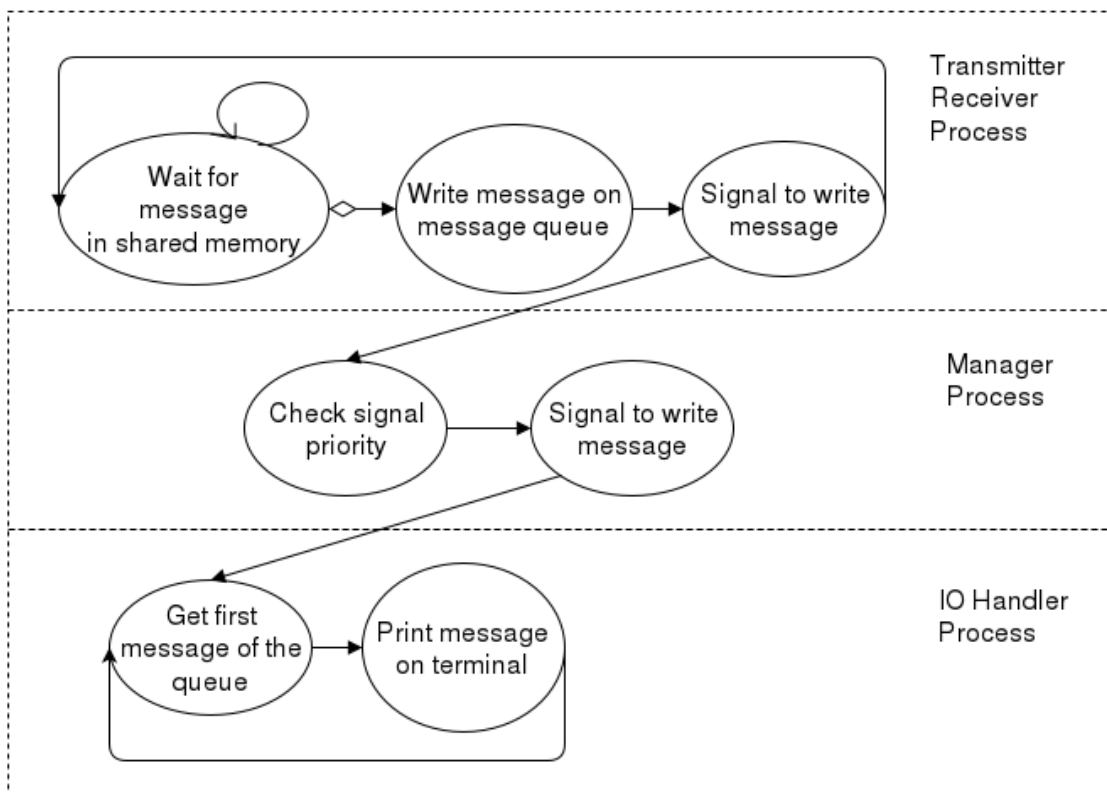
Verifica a cada segundo se há alguma nova mensagem na seu espaço de leitura da memória compartilhada, quando existe, bloqueia a escrita, a copia para a fila de mensagens e então libera a escrita no seu espaço de leitura na memória compartilhada.

Eventualmente, quando assionada pelo Manager via sinal, recupera uma mensagem da fila de mensagens e tenta escreve-la no espaço de leitura do outro módulo, para que o mesmo possa tratá-la.

Dessa forma, o processo de escrita pode ser ilustrado da seguinte maneira.



O processo de leitura é descrito da seguinte maneira:



2. Atividades

Foi solicitado no trabalho uma descrição das atividades e a evolução do software. Felizmente, a utilização da ferramenta Git possibilita uma visão real do desenvolvimento incluindo gráficos precisos sobre cada etapa da construção de códigos. As correções de erros e bugs podem ser rastreadas pelos títulos dos commits assim como os passos incrementais para a elaboração da solução final.

Link: <https://github.com/luanguimaraesla/unix-messenger/commits/master>

3. Melhorias

Diversos pontos de melhorias foram elicitados. Alguns deles são:

- a) Salvar os diálogos;
- b) Cria contas de usuários;
- c) Permitir que o chat funcione para mais de dois módulos;
- d) Permitir que o usuário selecione o destino da mensagem;
- e) Construir uma interface gráfica que favoreça a visualização;

Vários outros pontos arquiteturais foram observados, entretanto, é necessário se fazer um melhor estudo sobre sockets e comunicação entre diferentes hosts para que possa ser proposta uma melhoria efetiva.

4. Conclusão

A elaboração do projeto foi de grande valia para a absorção dos conteúdos referentes ao gerenciamento de processos em sistemas operacionais UNIX entre eles, destaca-se a importância necessária da extrema atenção ao se codificar softwares multiprogramados, já que é extremamente difícil a depuração do mesmo. Um sistema de log eficiente foi construído para reduzir as horas decorridas na correção de bugs e falhas. A redução foi significativa.

Outro ponto positivo observa-se quando a organização modular favorece intensivamente a alteração de pequenas partes encapsuladas do sistema, facilitando tanto a correção de erros como a evolução geral do programa.

5. Referências

[Tanenbaum, 2003] Tanenbaum, A. Sistemas Operacionais Modernos. 2a. Ed