**RELATÓRIO DE LABORATÓRIO 4**

**SISTEMAS OPERACIONAIS**

VICTOR MONEGO

ENGENHARIA ELETRÔNICA – UTFPR

16 DE ABRIL DE 2024

**1.Introdução Geral**

O seguinte relatório diz respeito ao Laboratório 04 de Sistemas Operacionais, focado na Comunicação Inter Processos por Filas de Mensagens POSIX.

Para a realização das análises a seguir, foi utilizada a plataforma WSL(Ubuntu) no Windows 11, e os códigos foram feitos usando o programa Notepad++.

**2.Comunicação simples entre processos A e B**

A figura 01 abaixo apresenta o código “processA\_transmitter.c”, que é o código do processo A que enviará as mensagens.

|  |
| --- |
| // UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANA  // DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA  // VICTOR AUGUSTO DEL MONEGO - 2378345  // Codigo processA\_transmitter.c : gera numeros aleatorios e envia mensagens para uma fila POSIX  // No prompt do Linux: compilar usando: gcc -o processA\_transmitter processA\_transmitter.c -lrt  #include <stdio.h>  #include <stdlib.h>  #include <mqueue.h>  #include <sys/stat.h>  #include <unistd.h>  // operating system check  #if defined(\_WIN32) || (!defined(\_\_unix\_\_) && !defined(\_\_unix) && (!defined(\_\_APPLE\_\_) || !defined(\_\_MACH\_\_)))  #warning Este codigo foi planejado para ambientes UNIX (LInux, \*BSD, MacOS). A compilacao e execucao em outros ambientes e responsabilidade do usuario.  #endif  #define QUEUE "/my\_queue"  **int** **main** (**int** argc, **char** \*argv[]){    **mqd\_t** queue;//descrição da fila  **struct** mq\_attr attr;//define os atributos da fila  **int** message;//declara a variavel que irá receber a mensagem  attr.mq\_maxmsg = **10**;//isto representa a capacidade do vetor de mensagens  attr.mq\_msgsize = **sizeof**(message);//especifica o tamanho de cada mensagem  attr.mq\_flags = **0**;  umask(**0**);//mascara de permissão    // esta seção do código é referente a criação da fila com permissões 0666  **if**((queue = mq\_open(QUEUE, O\_RDWR|O\_CREAT, **0666**, &attr)) < **0**) {  perror("mq\_open");//imprime mensagem de erro  exit(**1**);  }  **for**(;;){  message = random() % **100**;//message recebe um valor aleatório entre 0 e 99  **if** (mq\_send (queue, (**void**\*) &message, **sizeof**(message), **0**) < **0**){//verifica se houve erro no envio  perror("mq\_send");//caso erro, envia mensagem de erro  exit(**1**);  }  printf("Mensagem enviada com valor %d**\n**", message);//imprime a mensagem transmitida  sleep(**1**);  }  } |

Figura 1: Código processA\_transmitter.c

A figura 02 abaixo apresenta o código “processB\_receiver”, que é o código do processo receptor das mensagens.

|  |
| --- |
| // UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANA  // DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA  // VICTOR AUGUSTO DEL MONEGO - 2378345  // Codigo processB\_receiver.c : recebe mensagens de uma fila de mensagens POSIX  // No prompt do Linux: compilar usando: gcc -o processB\_receiver processB\_receiver.c -lrt  #include <stdio.h>  #include <stdlib.h>  #include <mqueue.h>  #include <sys/stat.h>  #include <unistd.h>  // operating system check  #if defined(\_WIN32) || (!defined(\_\_unix\_\_) && !defined(\_\_unix) && (!defined(\_\_APPLE\_\_) || !defined(\_\_MACH\_\_)))  #warning Este codigo foi planejado para ambientes UNIX (LInux, \*BSD, MacOS). A compilacao e execucao em outros ambientes e responsabilidade do usuario.  #endif  #define QUEUE "/my\_queue"  **int** **main** (**int** argc, **char** \*argv[])  {  **mqd\_t** queue; //descrição da fila  **struct** mq\_attr attr; //define os atributos da fila  **int** message; //declara a variavel que irá receber a mensagem  attr.mq\_maxmsg = **10**; //isto representa a capacidade do vetor de mensagens  attr.mq\_msgsize = **sizeof**(message); //especifica o tamanho de cada mensagem  attr.mq\_flags = **0**;    umask(**0**); //mascara de permissão    // esta seção do código é referente a criação da fila com permissões 0666  **if**((queue = mq\_open(QUEUE, O\_RDWR|O\_CREAT, **0666**, &attr)) < **0**)  {  perror("mq\_open"); //imprime mensagem de erro  exit(**1**);  }  //esta seção se refere à recepção de mensagens  **for**(;;)  {  **if**((mq\_receive (queue, (**void**\*) &message, **sizeof**(message), **0**)) < **0**) //verifica se houve erro de recepção  {  perror("mq\_receive:"); //imprime mensagem de erro  exit(**1**);  }  printf("Recebido a mensagem com valor %d**\n**", message); //imprime um aviso de recebimento de mensagem  }  } |

Figura 2: Código processB\_receiver.c

O princípio de funcionamento é de que o processo A cria um buffer de mensagens de um tamanho estipulado no próprio código, e gera números aleatórios para usar como mensagens, que preenchem o vetor. O processo B então “abre” este buffer e lê as mensagens. Caso já existam mensagens no buffer, ele lê todas sequencialmente. Caso o processo B não seja executado, o processo A envia mensagens até encher o buffer, e após isso não consegue mandar mais mensagens, portanto o algoritmo indiretamente “pausa” até o processo B começar a ler as mensagens.

Interface gráfica do usuário, Texto

Descrição gerada automaticamenteA figura 3 abaixo representa a execução de ambos os processos simultaneamente.

Figura 3: Execução da comunicação A - B

Observando os dados enviados pelo processo A e recebidos pelo processo B, percebemos que todos os dados escritos são lidos, logo nenhuma informação que foi transmitida se perde, e é lida normalmente. O sistema utilizado, de criar um buffer de mensagens e preenchê-lo, é eficiente em garantir com segurança que as mensagens não se percam na pipeline, e a transmissão ocorra com êxito.

Tendo isso em mente, existe outra implementação possível.

**3. Comunicação entre processos por memória compartilhada**

A figura 4 abaixo representa o código “sharedmem\_transmitter.c”, onde um processo aloca, cria e mapeia uma memória compartilhada e envia mensagens de forma similar ao processo A.

|  |
| --- |
| // UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANA  // DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA  // VICTOR AUGUSTO DEL MONEGO - 2378345  // Arquivo sharedmem\_transmitter.c: cria e usa uma área de memória compartilhada. Apenas escreve. Ademais utiliza semaforos para sincronizar parmissões de escrita  // No prompt do Linux: compilar usando: gcc -o sharedmem\_transmitter sharedmem\_transmitter.c -lrt  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/mman.h>  #include <unistd.h>  **int** **main** (**int** argc, **char** \*argv[])  {  **int** fd, value, \*ptr;  **int** s\_fd, s\_value, \*s\_ptr; // usados para o semaforo    // Passos 1 a 3: abre/cria uma area de memoria compartilhada  fd = shm\_open("/sharedmem", O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR);  **if**(fd == -**1**) {  perror ("shm\_open");  exit (**1**) ;  }  // Abre/cria uma área de memória compartilhada para utilizar como semáforo  s\_fd = shm\_open("/sharedmem\_s", O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR);  **if**(s\_fd == -**1**) {  perror("shm\_s\_open");  exit (**1**);  }  // Passos 1 a 3: ajusta o tamanho da area compartilhada  **if** (ftruncate(fd, **sizeof**(value)) == -**1**) {  perror ("ftruncate");  exit (**1**) ;  }  // Faz o mesmo para a area de memoria do semáforo  **if** (ftruncate(s\_fd, **sizeof**(s\_value)) == -**1**) {  perror ("ftruncate\_s");  exit (**1**);  }  // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo  ptr = mmap(NULL, **sizeof**(value), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, **0**);  **if**(ptr == MAP\_FAILED) {  perror ("mmap");  exit (**1**);  }  // Faz o mesmo para a área de memória do semáforo  s\_ptr = mmap(NULL, **sizeof**(s\_value), PROT\_READ|PROT\_WRITE, MAP\_SHARED, s\_fd, **0**);  **if**(s\_ptr == MAP\_FAILED) {  perror ("mmap\_s");  exit (**1**);  }  **for** (;;) {  // Passo 5: escreve um valor aleatorio na area compartilhada  **if** ((\*s\_ptr) == **1**)  sleep(**1**);  **else** {  value = random () % **1000** ;  (\*ptr) = value ;  (\*s\_ptr) = **1**; // seta um flag para fazer o transmissor aguardar antes de escrever  printf ("Wrote value %i**\n**", value) ;  sleep (**1**);  }  }  } |

Figura 4: código sharedmem\_transmitter.c

A figura 5 abaixo representa o código “sharedmem\_reciever.c”, onde um processo aloca, cria e mapeia uma memória compartilhada e recebe mensagens de forma similar ao processo B.

|  |
| --- |
| // UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANA  // DEPARTAMENTO ACADEMICO DE ENGENHARIA ELETRONICA  // VICTOR AUGUSTO DEL MONEGO - 2378345  // Arquivo sharedmem\_receiver.c: cria e usa uma área de memória compartilhada. Apenas lê. Ademais também utiliza semaforos para liberar para a escrita.  // No prompt do Linux: compilar usando: gcc -o sharedmem\_receiver sharedmem\_receiver.c -lrt  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <sys/mman.h>  #include <unistd.h>  **int** **main** (**int** argc, **char** \*argv[])  {  **int** fd, value, \*ptr;  **int** s\_fd, s\_value, \*s\_ptr; // usados para o semaforo  // Passos 1 a 3: abre/cria uma area de memoria compartilhada  fd = shm\_open("/sharedmem", O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR);  **if**(fd == -**1**) {  perror ("shm\_open");  exit (**1**) ;  }  // Abre/cria uma área de memória compartilhada para utilizar como semáforo  s\_fd = shm\_open("/sharedmem\_s", O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR);  **if**(s\_fd == -**1**) {  perror("shm\_s\_open");  exit (**1**);  }  // Passos 1 a 3: ajusta o tamanho da area compartilhada  **if** (ftruncate(fd, **sizeof**(value)) == -**1**) {  perror ("ftruncate");  exit (**1**) ;  }  // Faz o mesmo para a area de memoria do semáforo  **if** (ftruncate(s\_fd, **sizeof**(s\_value)) == -**1**) {  perror ("ftruncate\_s");  exit (**1**);  }  // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo  ptr = mmap(NULL, **sizeof**(value), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, **0**);  **if**(ptr == MAP\_FAILED) {  perror ("mmap");  exit (**1**);  }  // Faz o mesmo para a área de memória do semáforo  s\_ptr = mmap(NULL, **sizeof**(s\_value), PROT\_READ|PROT\_WRITE, MAP\_SHARED, s\_fd, **0**);  **if**(s\_ptr == MAP\_FAILED) {  perror ("mmap\_s");  exit (**1**);  }  **for** (;;) {  // Passo 5: le e imprime o conteudo da area compartilhada.  value = (\*ptr);  (\*s\_ptr) = **0**; //libera para escrita, retirando o flag inserido pelo transmissor  printf("Read value %i**\n**", value);  sleep(**1**);  }  } |

Figura 5: código sharedmem\_receiver.c

Por mais que seja mais versátil, existe um grande ponto fraco de implementar memorias compartilhadas para o compartilhamento de mensagens. Nos códigos acima, se for implementado apenas um espaço simples de memória compartilhada, não existe a segurança de que todas as informações que estão sendo transmitidas serão devidamente recebidas. Como visto anteriormente na comunicação simples, existe um “hard barrier” que evita que o processo A transmita mensagens em um momento em que o processo B não irá receber. Logo, para que a memória compartilhada, é necessário implementar um mecanismo de contingência. No caso desses códigos, é o semáforo.

Vamos observar as figuras 6 e 7 abaixo.

|  |
| --- |
| // Abre/cria uma área de memória compartilhada para utilizar como semáforo  s\_fd = shm\_open("/sharedmem\_s", O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR);  **if**(s\_fd == -**1**) {  perror("shm\_s\_open");  exit (**1**);  }  // Faz o mesmo para a area de memoria do semáforo  **if** (ftruncate(s\_fd, **sizeof**(s\_value)) == -**1**) {  perror ("ftruncate\_s");  exit (**1**);  }  // Faz o mesmo para a área de memória do semáforo  s\_ptr = mmap(NULL, **sizeof**(s\_value), PROT\_READ|PROT\_WRITE, MAP\_SHARED, s\_fd, **0**);  **if**(s\_ptr == MAP\_FAILED) {  perror ("mmap\_s");  exit (**1**); |

Figura 6: declarações do semáforo

|  |
| --- |
| **for** (;;) {  // Passo 5: le e imprime o conteudo da area compartilhada.  value = (\*ptr);  (\*s\_ptr) = **0**; //libera para escrita, retirando o flag inserido pelo transmissor  printf("Read value %i**\n**", value);  sleep(**1**);  } |

Figura 7: loop de execução com semáforo incluso

Interface gráfica do usuário, Texto

Descrição gerada automaticamenteInterface gráfica do usuário, Texto

Descrição gerada automaticamenteOs trechos de código nos demonstram que o semáforo se trata de um espaço de memória compartilhada adicional, com uma flag. Esse flag é utilizado com o novo “hard barrier” para garantir a sincronização de ambos os processos. O processo transmissor configura o flag para 1 e aguarda. Quando o processo receptor lê a mensagem, configura o flag para 0, e só então o transmissor envia outra mensagem. Essa lógica é exemplificada nas figuras 8 e 9.

Figura 9: transmissor e receptor sincronizados

Figura 8: processo transmissor aguardando processo receptor

Recapitulando, o que podemos observar como diferença entre os dois tipos de comunicação é a maneira como transmitem dados. Na comunicação entre processos A e B, é utilizado um buffer simples de mensagens, e na comunicação entre transmissor e receptor, é utilizado um espaço de memória compartilhada entre ambos os processos, juntamente com um espaço adicional configurado como semáforo.

**Referências:**

**-** MAZIERO, C. A. **Construção de semáforos**. Disponível em:[https://wiki.inf.ufpr.br/maziero/doku.php?id=so:semaforos&s[]=ipc](https://wiki.inf.ufpr.br/maziero/doku.php?id=so:semaforos&s%5b%5d=ipc). Acesso em: abril de 2024

- MAZIERO, C. A. **Filas de mensagens**. Disponível em:[https://wiki.inf.ufpr.br/maziero/doku.php?id=so:filas\_de\_mensagens&s[]=mensagens](https://wiki.inf.ufpr.br/maziero/doku.php?id=so:filas_de_mensagens&s%5b%5d=mensagens). Acesso em: abril de 2024

- COPETTI, L. F. **ELF66-S12 - Sistemas Operacionais 2024-1**. Disponível em: <https://moodle.utfpr.edu.br/course/view.php?id=18965>. Acesso em: abril de 2024