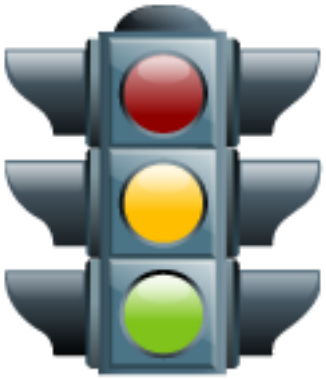
Sistemas Operacionais II

Comunicação entre Processos

Fabricio Breve

Sumário

- Introdução à Comunicação entre Processos
- Memória Compartilhada
- Semáforos de Processos
- Memória Mapeada



Comunicação entre Processos

- Nas aulas sobre processos, vimos como obter o status de saída de um processo filho.
  - Esta é a forma mais simples de comunicação entre processos;
  - Mas não vimos nenhuma forma de comunicação com o processo filho enquanto ele está executando;
  - Também não vimos como fazer a comunicação entre dois processos que não tem uma relação de pai e filho.
- Na aula de hoje veremos meios de fazer comunicação entre processos que superam essas limitações.

Comunicação entre Processos

- Exemplos de comunicação entre processos:
  - Um navegador Web requisita uma página de um servidor Web, que envia uma página HTML.
    - Normalmente utilizando sockets.
  - Imprimir nomes de arquivos com **ls | lpr**
    - O *shell* cria um processo **ls** e um processo **lpr** separados, conectando-os com um *pipe*, representado por **|**
      - O processo **ls** escreve dados no *pipe*, e o **lpr** lê dados do *pipe*

Comunicação entre Processos

- Discutiremos cinco tipos de comunicação entre processos:
  - **Memória compartilhada** permite que processos se comuniquem simplesmente lendo e escrevendo para uma localização especial de memória.
  - **Memória mapeada** é similar a memória compartilhada, exceto por estar associada com um arquivo no sistema de arquivos.
  - **Pipes** permitem comunicação sequencial de um processo para um processo relacionado.
  - **FIFOs** são similares a *pipes*, exceto que processos não relacionados podem se comunicar, pois o *pipe* tem um nome no sistema de arquivos.
  - **Sockets** suportam comunicação entre processos não relacionados mesmo em diferentes computadores.

Comunicação entre Processos

- Os vários tipos de comunicação entre processos **se diferem** pelos seguintes critérios:
  - Restrição à comunicação:
    - Entre processos relacionados (processos com um ancestral comum);
    - Entre processos não relacionados compartilhando o mesmo sistema de arquivos;
    - Qualquer computador conectado na rede.
  - Limitar a comunicação a somente escrever ou ler dados.
  - O número de processos que podem se comunicar.
  - Se os processos se comunicando são sincronizados automaticamente
    - Por exemplo: um processo de leitura poderia parar até que dados estivessem disponíveis para leitura.

MEMÓRIA COMPARTILHADA

Memória compartilhada

- Um dos mais simples métodos de comunicação entre processos é usar memória compartilhada.
- Memória compartilhada permite que dois ou mais processos acessem a mesma memória, como se ambos tivessem chamado **malloc** e este tivesse retornado ponteiros para o mesmo endereço de memória real.
- Quando um processo muda algo que está na memória compartilhada, todos os outros processos veem a modificação.

Comunicação Rápida Local

- Memória compartilhada é a forma mais rápida de comunicação entre processos.
  - Todos os processos compartilham a mesma memória.
  - Acesso a tal memória é tão rápido quanto o acesso à memória não compartilhada de processos.
  - Não é necessário utilizar chamadas de sistema ou entrar no núcleo.
  - Evita a cópia desnecessária de dados.

Comunicação Rápida Local

- O núcleo não sincroniza o acesso à memória compartilhada, portanto você deve fornecer seus próprios meios de sincronização.
  - Por exemplo:
    - Um processo não deve ler da memória antes que os dados sejam escritos lá.
    - Dois processos não podem escrever ao mesmo tempo na mesma localização de memória.
  - Uma solução para evitar tais condições de corrida é usar semáforos, como veremos mais adiante

O Modelo de Memória

- Para usar um segmento de memória compartilhada, **um processo** precisa **alocar** o segmento.
- Então **cada processo** que deseja acessar o segmento, deve **acoplá-lo**.
- Após finalizar o uso do segmento, **cada processo desacopla** o segmento.
- Em algum momento, **um processo** precisa **desalocar** o segmento.

O Modelo de Memória

- Entender o modelo de memória do Linux ajuda a explicar a alocação e o acoplamento de processos.
  - No Linux, a memória virtual de cada processo é dividida em páginas.
  - Cada processo mantém um mapa de seus endereços de memória para estas páginas virtuais de memória, que contém os dados reais.
  - Apesar de cada processo ter seus próprios endereços, múltiplos mapeamentos de processos podem apontar para a mesma página, permitindo o compartilhamento de memória.

O Modelo de Memória

- Alocar um novo segmento de memória compartilhada faz páginas de memória virtual serem criadas.
- Como todos os processos desejam acessar o mesmo segmento compartilhado, apenas um processo deve alocar o novo segmento.
- Alocar um segmento existente não cria novas páginas, mas retorna um identificador para páginas existentes.
- Para um processo usar um segmento de memória compartilhado, ele deve acoplá-lo, o que adiciona entradas no mapa de sua memória virtual para o segmento de página compartilhado.

O Modelo de Memória

- Quando termina com o segmento, tais entradas de mapeamento são removidas.
- Quando mais nenhum processo quer acesso a estes segmentos de memória, **exatamente um** processo precisa desalocar as páginas de memória virtual.
- Todo os segmentos de memória são alocados como múltiplos do *tamanho de página* do sistema, que é o número de bytes em uma página de memória.
  - Em sistemas Linux, o tamanho da página é normalmente 4KB, mas você deve obter esse valor chamando a função **getpagesize**.

Alocação

- Um processo aloca um segmento de memória usando **shmget** (“Shared Memory GET”)
  - Argumentos:
    1. Chave
       - Inteiro que especifica qual segmento criar.
       - Processos não relacionados podem acessar o mesmo segmento compartilhado especificando o mesmo valor de chave.
       - Infelizmente, outros processos podem escolher a mesma chave fixa, o que levaria a conflitos.
         - » Use a constante especial **IPC_PRIVATE** como chave para garantir que um novo segmento de memória seja criado.
    2. Número de bytes no segmento
       - Como os segmentos são alocados usando páginas, o número real de bytes alocados é um múltiplo do tamanho de página (arredondado para cima)
    3. Terceiro parâmetro no próximo slide

Alocação

- O terceiro argumento de **shmget** são *flags* que especificam opções.
  - Algumas *flags*:
    - **IPC_CREAT** – Indica que o novo segmento deve ser criado. Permite a criação de um novo segmento enquanto especifica um valor chave.
    - **IPC_EXCL** – Sempre usada com **IPC_CREAT**, causa uma falha em **shmget** se a chave de segmento especificada já existe, de modo que o processo chamador possa fazer os arranjos para ter um segmento exclusivo.
      - Se esta *flag* não for fornecida e a chave de um segmento existente for usada, **shmget** retorna o segmento existente em vez de criar um novo.
    - *Flags* de modo – valor feito de 9 bits indicando permissões oferecidas ao dono, ao grupo e outros para controlar acesso ao segmento. Bits de execução são ignorados. Uma maneira fácil de especificar permissões é usar constantes definidas em <sys/stat.h> e documentadas na sessão 2 da página de manual de **stat**.
      - Exemplo: **S_IRUSR** e **S_IWUSR** especificam permissões de leitura e escrita para o dono do segmento de memória compartilhada, e **S_IROTH** e **S_IWOTH** especificam permissões de leitura e escrita para outros.

Alocação

- Exemplo:
  - Usando **shmget** para criar um segmento de memória compartilhada (ou acesso a um existente, se **shm_key** estiver em uso) que pode ser lido e gravado pelo dono, mas não por outros usuários:
    - **int segment_id = shmget (shm_key, getpagesize (), IPC_CREAT | S_IRUSR | S_IWUSR);**
  - Se a chamada for bem sucedida, **shmget** retorna um identificador de segmento. Se o segmento de memória compartilhada já existe, as permissões de acesso são verificadas e uma checagem é feita para garantir que o segmento não está marcado para destruição.

Acoplamento e desacoplamento

- Para tornar o segmento de memória compartilhada disponível, um processo precisa usar **shmat** (“Shared Memory ATtach”).
  - Argumentos:
    1. Identificador de segmento **SHWID** retornado por **shmget**.
    2. Ponteiro que especifica onde no espaço de endereços do seu processo você quer mapear a memória compartilhada
       - Se você especificar NULL, o Linux irá escolher um endereço disponível.
    3. Uma *flag*, como por exemplo:
       - **SHM_RND** indica que o endereço especificado no segundo argumento deve ser arredondado para baixo para um múltiplo do tamanho de página. Se esta *flag* não for especificada, você mesmo precisa arredondar o tamanho de página que está passando para **shmat**.
       - **SHM_RDONLY** indica que o segmento será apenas lido, não escrito.

Acoplamento e desacoplamento

- Se a chamada a **shmat** for bem sucedida, ela retorna o endereço do segmento compartilhado acoplado.
  - Filhos criados com chamadas para **fork** herdam os segmentos de memória acoplados
    - Eles podem desacoplar os segmentos de memória compartilhados, se desejarem.
- Quando você terminar de usar o segmento de memória compartilhado, o segmento deve ser desacoplado usando **shmdt** (“SHared Memory DeTach”)
  - Passe a ele o endereço retornado por **shmat**.
    - Se o segmento for desalocado e este for o último processo usando-o, ele é removido
    - Chamadas para **exit** ou qualquer uma da família **exec** automaticamente desacoplam segmentos.

Controlando e Desalocando Memória Compartilhada

- A chamada **shmctl** (“SHared Memory ConTrol”) retorna informação sobre o segmento de memória compartilhado e pode modificá-lo.
  - O primeiro argumento é o identificador do segmento de memória compartilhado.
  - Para obter informação sobre o segmento de memória compartilhado, passe **IPC_STAT** como segundo argumento e um ponteiro para uma estrutura **shmid_ds** como terceiro argumento.
  - Para remover um segmento, passe **IPC_RMID** como segundo argumento, e **NULL** como terceiro argumento. O segmento é removido quando o último processo acoplado a ele se desacoplar.

Controlando e Desalocando Memória Compartilhada

- Cada segmento de memória compartilhado deve ser explicitamente desalocado usando **shmctl** quando terminar de ser utilizado, evitando violar o limite de segmentos de memória compartilhada do sistema.
  - Chamar **exit** e **exec** desacopla segmentos de memória, mas não os desaloca.
  - Veja a pagina de manual de **shmctl** para uma descrição de outras operações que você pode realizar com segmentos de memória compartilhados.

```c
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Alocar o segmento de memória compartilhado. */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);

  /* Acoplar o segmento de memória compartilhado. */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("Memória compartilhada acoplada no endereço %p\n", shared_memory);
  /* Determinar o tamanho do segmento. */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("Tamanho do segmento: %d\n", segment_size);
  /* Escrever uma string no segmento de memória compartilhado. */
  sprintf (shared_memory, "Olá, mundo.");
  /* Desacoplar o segmento de memória compartilhado. */
  shmdt (shared_memory);

  /* Reacoplar o segmento de memória compartilhado, em um diferente endereço. */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("Memória compartilhada reacoplada no endereço %p\n", shared_memory);
  /* Imprimir a string da memória compartilhada. */
  printf ("%s\n", shared_memory);
  /* Desacoplar o segmento de memória compartilhada. */
  shmdt (shared_memory);

  /* Desalocar o segmento de memória compartilhada. */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

shm.c

Exemplo de uso de memória compartilhada

Depurando

- O comando **ipcs** fornece informações sobre comunicações entre processos em andamento. Use a flag **-m** para obter informações sobre memória compartilhada.
  - Exemplo: o código a seguir ilustra um segmento de memória compartilhado numerado como 163845, em uso

```
fabricio@fabricio-virtual-machine:~/so2/aula8$ ipcs -m

- Segmentos da memória compartilhada -
chave      shmid      proprietário perms      bytes      nattch     status
0x00000000 163845     fabricio     777        30492      2          dest
```

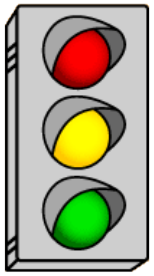
  - Se este segmento de memória foi erroneamente deixado para trás por um programa, você pode usar o comando ipcrm para removê-lo
    - ipcrm shm 163845

Prós e Contras

- Segmentos de memória compartilhados permitem comunicação bidirecional rápida entre qualquer número de processos
  - Cada usuário pode ler e escrever, mas o programa precisa estabelecer e seguir algum protocolo que evite condições de corrida, como sobrescrever informações antes que sejam lidas
  - Infelizmente o Linux não garante acesso exclusivo mesmo que você crie um novo segmento de memória compartilhada com **IPC_PRIVATE**
  - Além disso, para que múltiplos processos usem o mesmo segmento compartilhado, eles precisam fazer arranjos para usarem a mesma chave

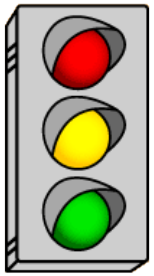
SEMÁFOROS DE PROCESSOS

Semáforos de Processos



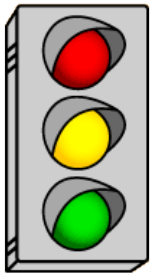
- Como mencionado anteriormente, processos precisam coordenar o acesso à memória compartilhada.
- Como visto na aula anterior, semáforos são contadores que permitem sincronizar múltiplas threads.
- O Linux oferece uma implementação distinta de semáforos para ser usada para sincronizar processos
  - Chamadas de semáforos de processos.
  - São alocados, usados e desalocados como segmentos de memória compartilhados.
  - Apesar de um único semáforo ser suficiente para a maioria dos usos, semáforos de processos vem em conjuntos.

Alocação e Desalocação

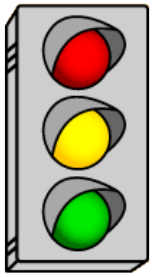


- As chamadas **semget** e **semctl** alocam e desalocam semáforos
  - Análogos a **shmget** e **shmctl** para memória compartilhada
  - Chame **semget** com uma chave especificando um conjunto de semáforos, a quantidade de semáforos no conjunto, e as *flags* de permissão como em **shmget**.
    - O valor de retorno é um identificador de conjunto de semáforos.
    - Você pode obter o identificador de um conjunto de semáforos existentes especificando a chave correta.
      - Nesse caso, a quantidade de semáforos pode ser definida como zero.

Alocação e Desalocação



- Semáforos continuam existindo mesmo após todos os processos que estavam usando-os terem terminado.
- O último processo a usar um conjunto de semáforos deve removê-lo explicitamente para garantir que o sistema operacional não fique sem semáforos.
  - Para tanto, invoque **semctl** com o identificador do semáforo, o número de semáforos no conjunto, **IPC_RMID** como terceiro argumento, e qualquer valor do tipo **union semun** como quarto argumento (que será ignorado).
    - O user ID do processo chamador deve ser o mesmo do alocador do semáforo (ou o chamador deve ser root).
    - Ao contrário de segmentos de memória, remover um conjunto de semáforos faz com que o Linux desaloque-os imediatamente.

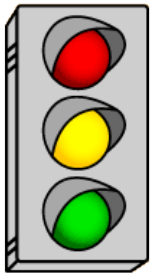


sem_all_deall.c

```c
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* Devemos definir union semun nós mesmos.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Obtenha um ID de semáforo binário, alocando-o se necessário.  */

int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}

/* Desalocar um semáforo binário.  Todos os usuários devem ter terminado seu
   uso.  Retorna -1 se falhar.  */

int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

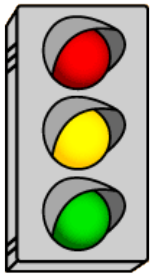
Alocando e desalocando um semáforo binário

Inicializando Semáforos



- Alocar e inicializar semáforos são duas operações separadas. Para inicializar um semáforo, use **semctl** com zero como segundo argumento e **SETALL** como terceiro argumento. Para o quarto argumento, você deve criar um objeto **union semun** e apontar o seu campo array para valores **unsigned short**. Cada valor é usado para inicializar um semáforo

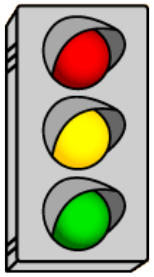
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Devemos definir union semun nós mesmos. */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Inicializar um semáforo binário com o valor de um. */

int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```



sem_init.c

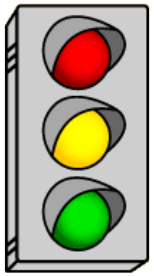
Inicializando um Semáforo Binário

Operações **Wait** e **Post**



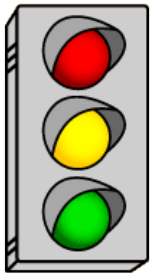
- Cada semáforo tem um valor não-negativo e suporta operações **wait** e **post**.
- A chamada de sistema **semop** implementa ambas as operações.
  - Argumentos:
    - Identificador do conjunto de semáforos
    - Array de elementos **struct sembuf** que especifica a operação a ser realizada
    - Comprimento do array

Operações **Wait** e **Post**



- Os campos de **struct sembuf** são:
  - **sem_num** é o número do semáforo no conjunto de semáforos no qual a operação será realizada.
  - **sem_op** é um inteiro que especifica a operação do semáforo.
    - Se positivo, o número é adicionado ao valor do semáforo imediatamente.
    - Se negativo, o valor absoluto do número é subtraído do valor do semáforo
      - Se isto for tornar o valor do semáforo negativo, a chamada bloqueia até que o semáforo se torne tão grande quanto o valor de **sem_op**
  - **sem_flg** é um valor de *flag*
    - Use **IPC_NOWAIT** para evitar o bloqueio da operação
      - A operação falha em vez de bloquear
    - Use **SEM_UNDO** para que o Linux automaticamente desfaça a operação no semáforo quando o processo sair
      - Também funciona em términos anormais

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Espera em um semáforo binário.  Bloqueia até que o valor do semáforo seja
   positivo, então o decrementa em um.  */

int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* Usa o primeiro (e único) semáforo.  */
  operations[0].sem_num = 0;
  /* Decrementa em 1.  */
  operations[0].sem_op = -1;
  /* Permite desfazer.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}

/* Incrementa um semáforo binário: incrementa seu valor em um.  Este
   retorna imediatamente.  */

int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Usa o primeiro (e único) semáforo.  */
  operations[0].sem_num = 0;
  /* Incrementa em 1.  */
  operations[0].sem_op = 1;
  /* Permite desfazer.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}
```



sem_pv.c

Operações Wait e Post para um Semáforo Binário

MEMÓRIA MAPEADA

Memória Mapeada

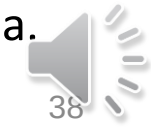
- A memória mapeada permite que diferentes processos se comuniquem através de um arquivo compartilhado.
  - Forma uma associação entre um arquivo e a memória de um processo.
    - O Linux divide o arquivo em pedaços do tamanho de páginas e então os copia para páginas de memória virtual para que sejam disponibilizados no espaço de endereços de um processo.
    - Dessa forma, o processo pode ler e gravar o conteúdo do arquivo com acessos ordinários à memória.

Memória Mapeada

- Você pode imaginar a memória mapeada como um buffer que armazena o conteúdo completo de um arquivo, com operações para ler o arquivo completo para o buffer e gravar do buffer para o arquivo caso hajam modificações.
  - O Linux cuida das operações de leitura e escrita para você.
- Além de comunicação entre processos, memória mapeada também tem outros usos, que veremos mais adiante.

Mapeando um Arquivo Comum

- Para mapear um arquivo comum para a memória de um processo, use a chamada **mmap**
  - (“Memory MAPped”, pronuncia-se “em-map”)
  - Argumentos:
    1. Endereço em que o Linux mapeará o arquivo no espaço de endereços do processo. O valor NULL permite ao Linux escolher um endereço livre.
    2. Tamanho do mapa em bytes.
    3. Especifica a proteção na faixa de endereços mapeados. Bits conectados por “ou”: **PROT_READ**, **PROT_WRITE**, e **PROT_EXEC**, correspondendo a proteções de leitura, escrita e execução, respectivamente.
    4. *Flag* de opções adicionais. (próximo slide)
    5. Descritor de arquivos aberto para o arquivo a ser mapeado.
    6. Deslocamento do início do arquivo para o ponto onde deve ser iniciado o mapeamento. Pode-se mapear todo ou parte do arquivo, selecionando o deslocamento e o tamanho de maneira adequada.



Mapeando um Arquivo Comum

- O valor de *flag* é uma sequência de bits conectados por “ou” das seguintes restrições:
  - **MAP_PRIVATE**
    - Escritas para a faixa de memória não serão escritas no arquivo anexado, mas em uma cópia privada do arquivo, que outros processos não podem ver.
  - **MAP_SHARED**
    - Escritas são imediatamente refletidas no arquivo em vez de serem “bufferizadas”. É o modo utilizado para comunicação entre processos.
  - MAP_PRIVATE e MAP_SHARED não podem ser usados simultaneamente.
  - Há outras restrições disponíveis no Linux (não compatíveis com POSIX).

Mapeando um Arquivo Comum

- Retorno de **mmap**:
  - Se bem sucedido, retorna um ponteiro para o início da memória.
  - Se falhou, retorna **MAP_FAILED**.
- Quando terminar de usar um mapeamento de memória, libere-o com **munmap**, passando o endereço inicial e o comprimento da região de memória mapeada.
  - O Linux automaticamente desmapeia regiões mapeadas quando um processo termina.

Exemplos

- Para ilustrar mapeamentos de memória, veremos dois exemplos:
  - O primeiro gera um número aleatório e o escreve para um arquivo de memória mapeada.
  - O segundo lê o número, imprime-o, e o substitui no arquivo de memória mapeada pelo dobro do valor.
- Ambos recebem como argumento o nome do arquivo a ser mapeado.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Retorna um valor aleatório uniforme na faixa [baixo,alto]. */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* Semeie o gerador de números aleatórios. */
  srand (time (NULL));

  /* Prepare um arquivo grande o suficiente para armazenar um inteiro sem sinal. */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Crie o mapeamento de memória. */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Escreva um inteiro aleatório para a área de memória mapeada. */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Libere a memória (desnecessário, visto que o programa termina). */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

mmap-write.c

Escreve um número aleatório para um arquivo de memória mapeada

Mapeando um Arquivo Comum

- fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  - O terceiro argumento especifica que o arquivo deve ser aberto para escrita e leitura
- lseek (fd, FILE_LENGTH+1, SEEK_SET);
  - Usado para garantir que o arquivo é grande o suficiente para armazenar um inteiro
    - Em seguida é usado novamente para retornar ao início do arquivo.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Abre um arquivo.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Cria o mapeamento de memória.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);

  /* Lê o inteiro, imprime-o, e dobra-o.  */
  sscanf (file_memory, "%d", &integer);
  printf ("Valor:  %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Libera a memória (desnecessário, visto que o programa termina).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

mmap-read.c

Lê um inteiro do Arquivo de Memória Mapeada, e o Dobra

Mapeando um Arquivo Comum

- Dessa vez não é necessário usar **lseek**, pois assumimos que o arquivo é grande o suficiente para armazenar um inteiro sem sinal.
- Note que **sscanf** e **sprinf** são usados para ler e escrever os inteiros como *strings* no arquivo de memória.
  - Porém não é necessário que o conteúdo dos arquivos de memória mapeada sejam texto. Também é possível ler e escrever qualquer binário arbitrário.

Acesso Compartilhado a um Arquivo

- Diferentes processos podem se comunicar usando regiões de memória mapeada associadas com o mesmo arquivo.
  - Especifique a *flag* **MAP_SHARED** para que qualquer escrita sejam transferidas imediatamente para o arquivo correspondente e fiquem visíveis para outros processos
    - Se a *flag* não for especificada, o Linux pode fazer buffer das escritas antes de transferi-las para o arquivo.

Acesso Compartilhado a um Arquivo

- Uma outra alternativa é forçar o Linux a incorporar as escritas “bufferizadas” ao arquivo em disco chamando **msync**
  - Os primeiros dois argumentos especificam a região de memória mapeada (nome, tamanho).
  - O terceiro argumento pode receber estas flags:
    - **MS_ASYNC** – A atualização é agendada, mas não necessariamente executa antes do retorno da chamada.
    - **MS_SYNC** – A atualização é imediata, a chamada a **msync** bloqueia até que seja feita.
    - **MS_INVALIDATE** – Todos os outros mapeamentos são invalidados para que possam ver os valores atualizados.

Acesso Compartilhado a um Arquivo

- Exemplo:
  - Para atualizar um arquivo de memória no endereço **mem_addr** de tamanho **mem_length** bytes, chame:
    - **msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);**
- Da mesma forma que com segmentos de memória, usuários de regiões de memória mapeada devem estabelecer e seguir um protocolo para evitar condições de corrida.
  - Por exemplo, um semáforo pode ser usado para evitar que mais de um processo acesse o mapeamento de memória ao mesmo tempo.
    - Outra alternativa é usar uma trava de escrita ou leitura no arquivo.

Mapeamentos Privados

- Especificando **MAP_PRIVATE** para **mmap** cria uma região de “cópia-na-escrita”
  - Outros processo que mapeiam a mesma região não verão as mudanças.
  - O processo escreve para uma cópia privada da página.
    - Todas as leituras e escritas subsequentes usarão tal página

Outros usos para *mmap*

- **mmap** pode ser usado para outras finalidades, além de comunicação entre processos
- Exemplos:
  - Substituto de **read** e **write**.
    - Em vez de escrever explicitamente conteúdo de arquivos na memória, um mapeamento pode ser usado e o programa utilizará operações simples de leitura.
      - Em alguns programas, o mapeamento de memória é mais conveniente e mais rápido que usar operações explícitas de entrada/saída.
  - Salvar estruturas de dados em arquivo.
    - Estruturas de dados podem ser gravadas em arquivos e restauradas nas próximas execuções do programa.
  - Mapear **/dev/zero**.
    - Fonte infinita de bytes 0 na leitura.
    - Escritas são descartadas.

Exercício 1

- Faça um programa que faça as seguintes operações (nesta ordem):
  - Crie um segmento de memória compartilhada ou de memória mapeada e armazene nele o número 2 em uma variável numérica de ponto flutuante.
  - Crie um processo filho, que herdará esse segmento.
  - Ambos o processo pai e o processo filho devem executar simultaneamente um loop de 1.000.000 de iterações em que façam as seguintes operações em sequência:
    - Elevar o valor da variável numérica ao quadrado e armazenar o resultado na mesma variável;
    - Tirar a raiz quadrada da variável numérica e armazenar o resultado na mesma variável;
    - Imprimir o valor atual da memória compartilhada (ou mapeada) com pelo menos 4 casas decimais.
- Apareceu algum valor diferente de 2?
  - Se sim. Por que?
  - Se não. Poderia acontecer? Por que?
- Com qual valor a variável ficou ao final das iterações dos dois processos?

Atenção: Coloque as respostas dos exercícios de 1 a 2 e os respectivos códigos-fontes no *Google Classroom*.

Exercício 2

- Refaça o exercício anterior garantindo que não existam condições de corrida.
- Como você implementou tal garantia?
- Apareceu algum valor diferente de 2?
  - Se sim. Por que?
  - Se não. Poderia acontecer? Por que?
- Com qual valor a variável ficou ao final das iterações dos dois processos?

Atenção: Coloque as respostas dos exercícios de 1 a 2 e os respectivos códigos-fontes no *Google Classroom*.

Referências Bibliográficas

1. NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.;. *Manual Completo do Linux: Guia do Administrador*. São Paulo: Pearson Prentice Hall, 2007. Cap. 4.
2. DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; *Sistemas Operacionais: terceira edição*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20.
3. MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 5.
4. TANENBAUM, Andrew S.; *Sistemas Operacionais Modernos*. 3ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 10.