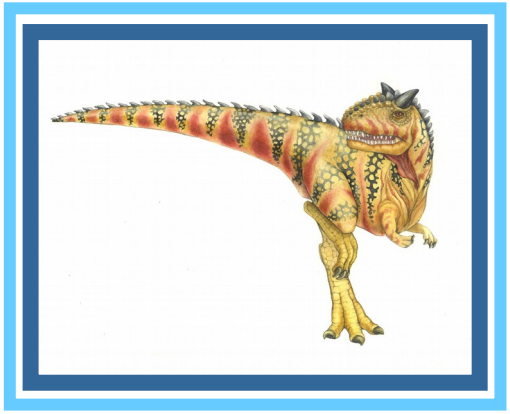
Capítulo 3: Processos (cont.)



Interprocess Communication

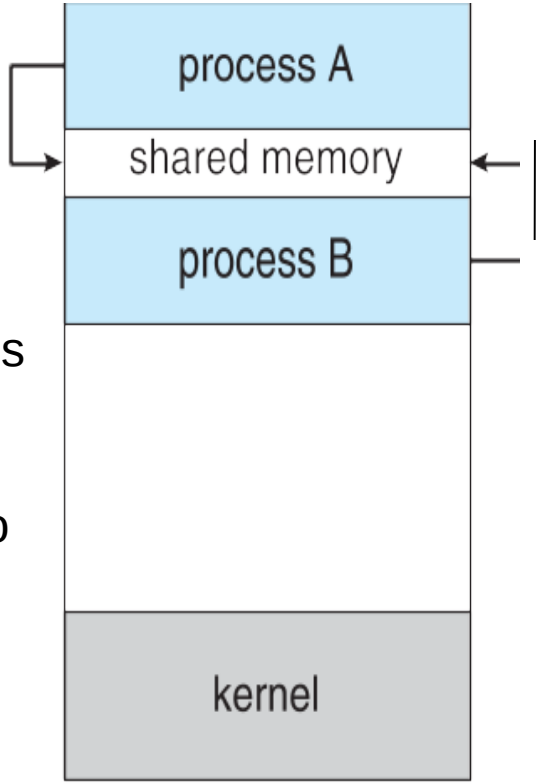
Processos e threads podem se comunicar – **interprocess communication** (**IPC**) – e sincronizar suas ações.

- Mecanismos de comunicação
  - **shared memory (memória compartilhada)**
  - Data-transfer (transferência de dados):
    - **message passing**
    - data exchanged via **pipes** and **FIFOs**

- Mecanismos de sincronização: evitam atualizações simultâneas de memória compartilhada evitando inconsistência dos dados.
  - Semaphores
  - File locks
  - Mutexes (threads)
  - Condition variables (threads)

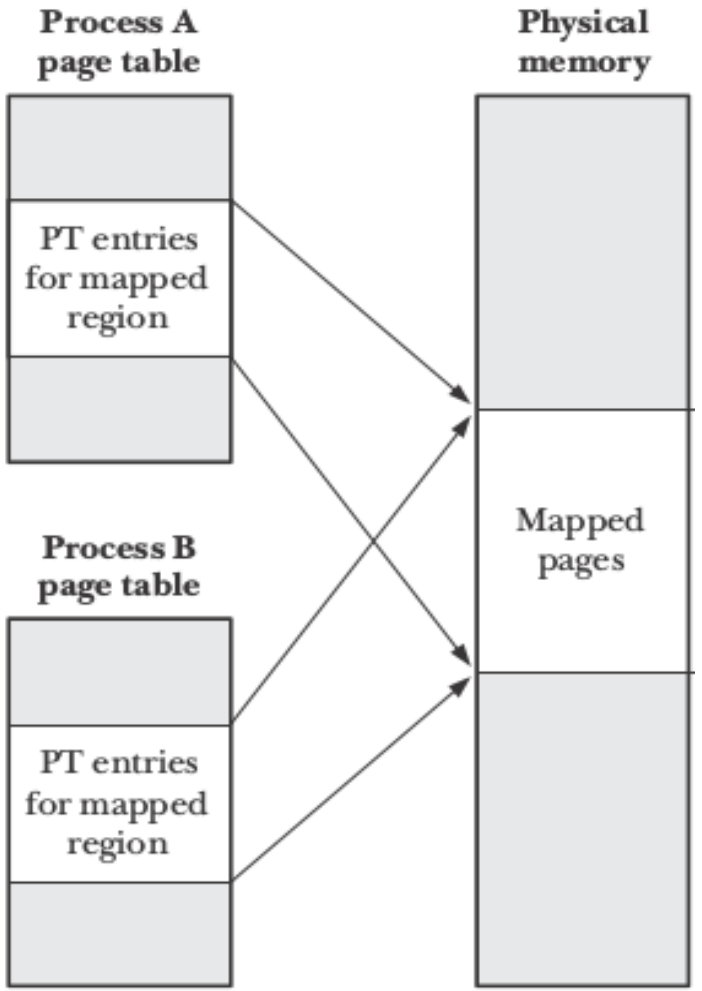
Interprocess Communication – Shared Memory

- Área de memória compartilhada entre processos que desejam se comunicar.
- A comunicação é controlada pelos processos de usuário e não pelo SO.
  - O SO fornece as chamadas de sistema que permitem manipular a área compartilhada.
- Mecanismos de sincronização devem ser utilizados para evitar inconsistência dos dados na memória compartilhada.
- Mecanismos de sincronização serão estudados no próximo capítulo.



Interprocess Communication – Shared Memory

- Como o segmento de memória passa a fazer parte do espaço de endereçamento do processo, não é necessária a intervenção do Kernel para realizar a comunicação entre processos.
- As leituras não “consomem” os dados.
- Operação de leitura não bloqueia o leitor.
- Exige a implementação de um mecanismo de sincronização para coordenar os processos evitando que os dados se tornem inconsistentes. Exemplos de mecanismos de sincronização:
  - Semáforos,
  - Trava em arquivos (file locks)
  - Mutexes e variáveis de condição.



Examples of IPC Systems - POSIX

- Estudar os programas:
  - shm-posix-producer.c
  - shm-posix-consumer.c
  - shm-posix-consumer_fork.c

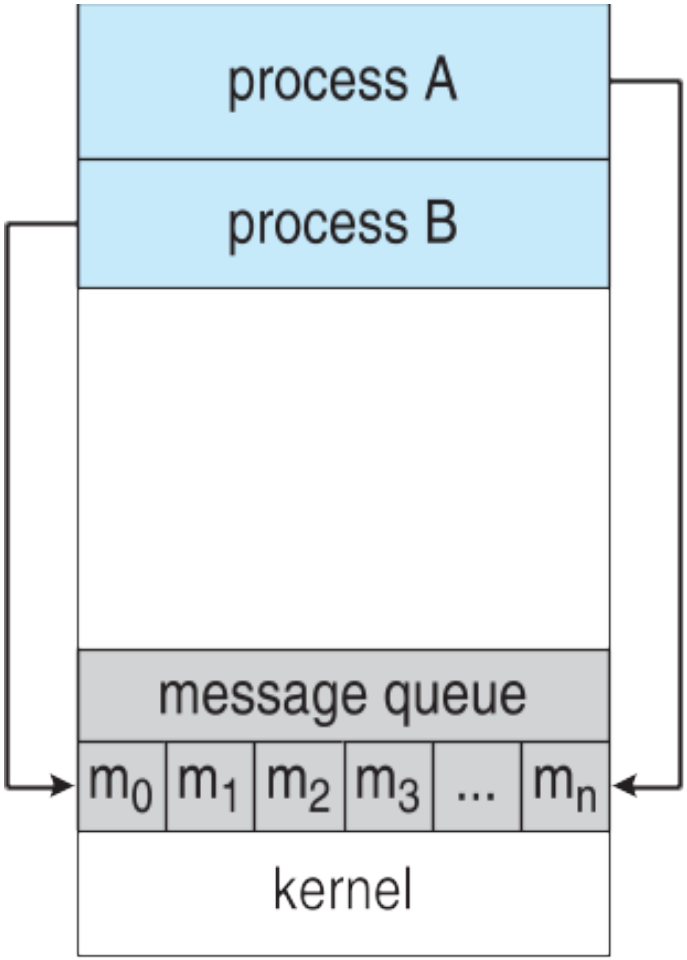
Um objeto compartilhado é removido quando o sistema é reiniciado (*kernel persistence*) ou por meio da função ***shm_unlink( )***

Interprocess Communication – Shared Memory

- Um processo filho criado por meio da função *fork()* herda os mapeamentos de memória de seu pai?
- Um mapeamento continua existindo no processo filho quando este executa um *execve()*?
- Os dados são consumidos na operação de leitura da memória compartilhada?
- A comunicação entre processos, por meio de memória compartilhada, pode ser bidirecional?
- Qual a função do Kernel nas operações de leitura e escrita na memória compartilhada?
- Quando a área de memória compartilhada não está mapeada por nenhum processo, ela deixa de existir?

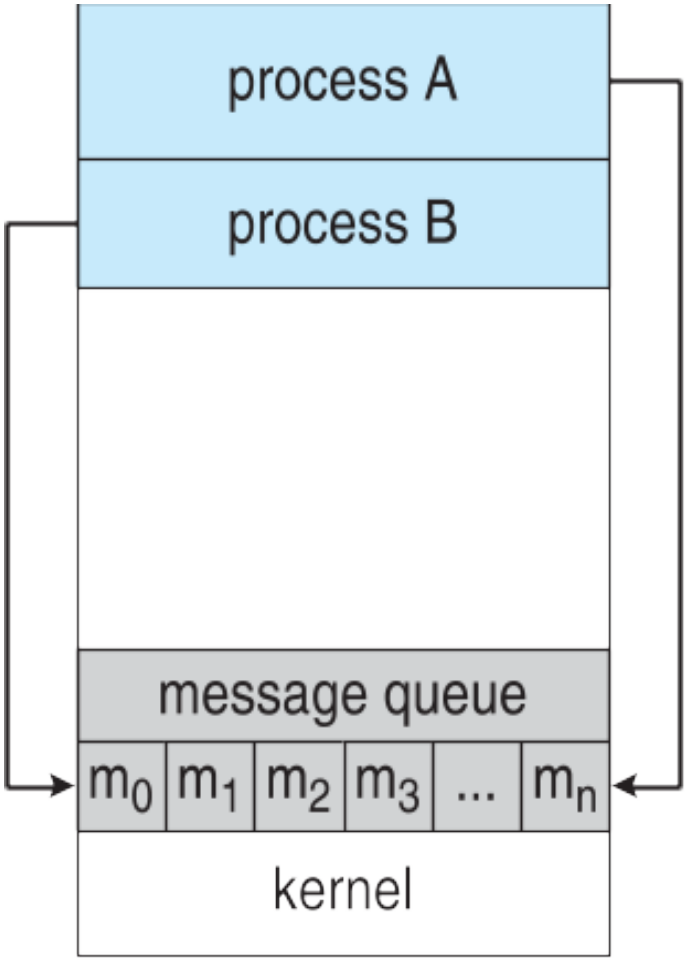
Interprocess Communication – Message Passing

- A comunicação ocorre por meio do Kernel
- Operação:
  - **send**(*message*)
  - **receive**(*message*)
- A fila de mensagem possui um identificador público (**public identifier**)
  - Em POSIX o identificador é um nome de arquivo.



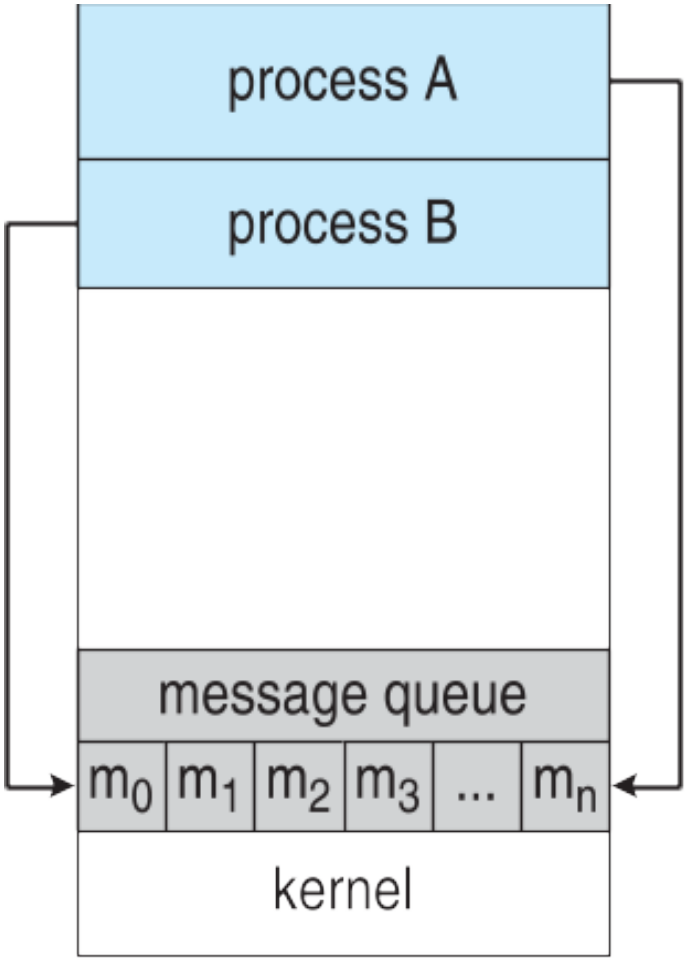
IPC – Message Passing – Posix message queue

- Após abrir uma fila, mensagens podem ser enviadas ou recebidas por processos.
- A mensagem é “consumida” da fila após a leitura.
- Quando não há nenhuma mensagem na fila, um pedido de leitura feito por um processo (a process request) pode:
  - Ser notificado imediatamente com um código de erro (*error code)*;
  - Ficar bloqueado até que uma mensagem esteja na fila.
  - Receber uma notificação de forma assíncrona (via threads).



IPC – Message Passing – Posix message queue

- Sistemas POSIX permitem definir um limite para a quantidade e o tamanho de cada mensagens na fila.
- Fila cheia (queue full): o processo pode ser notificado imediatamente com um código de erro ou ficar suspenso até que exista espaço na fila.
- Uma fila somente é removida quando houver um pedido de remoção e não estiver sendo usada por nenhum processo



Comunicação direta e indireta

- **Direta**: os processos devem nomear o destinatário explicitamente:
  - send (P, message) – envia uma mensagem ao processo P
  - receive(Q, message) – recebe uma mensagem de um processo Q
  - O *link* é estabelecido entre pares de processos.
  - O *link* pode ser unidirecional ou bidirecional.

Comunicação direta e indireta

• **Indireta**: mensagens são enviadas e recebidas por meio de caixas de mensagens ('mailboxes')

- send(A, message) – envia mensagem para “mailbox” A.
- receive(A, message) – recebe mensagem de “mailbox” A.
- Cada “mailbox” possui um *id* único.
- Processos se comunicam via caixa de mensagem compartilhada.
- Um *link* pode estar associado a vários processos.
- Cada par de processos pode compartilhar vários *links*.
- Um *link* pode ser unidirecional ou bidirecional
- Quem pega a mensagem?

- Qualquer processo pode pegar a mensagem ou o próprio sistema escolhe arbitrariamente um destinatário. O remetente pode ser notificado sobre qual processo recebeu a mensagem.

Synchronization

- A troca de mensagem pode ser *blocking or non-blocking*
- **Blocking** é considerada **synchronous**
  - **Blocking send** – o remetente fica bloqueado até que a mensagem seja lida.
  - **Blocking receive** – o destinatário fica bloqueado até que a mensagem esteja disponível.
- **Non-blocking** é considerada **asynchronous**
  - **Non-blocking send** – o destinatário envia mensagem e continua.
  - **Non-blocking receive** – ao executar a operação de leitura o processo leitor recebe:
    - uma mensagem válida ou
    - null

IPC – Message Passing – Posix message queue

- Estudar programas:
  - p01_send.c
  - p01_receive.c
  - p02_send.c
  - p02_receive.c
  - p03_send.c
  - p03_receive.c
  - p04_send_receive.c

Messages Queue - POSIX

- Aspectos de implementação:
  - Um canal (link) de comunicação pode ser asssociado a mais do que dois processos?
  - O tamanho da mensagem trocada pode ser de tamanho variável?
  - O canal de comunicação é unidirecional ou bidirecional?
- A fila de mensagens continua existindo quando não há nenhum processo associado a ela?
- Quando algum processo associado a uma fila termina por algum erro (ex. falha de segmentação), o que acontece com a fila?

Messages Queue - POSIX

- Site que contém exemplos sobre fila de mensagens em POSIX.

http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch06.html

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Ordinary Pipes

- *Ordinary Pipes* permitem a comunicação no estilo produtor-consumidor.
- Produtor escreve em uma extremidade do pipe (the **write-end** of the pipe).
- Consumidor faz a leitura em outra extremidade (the **read-end** of the pipe).
- Requer uma relação pai-filho entre os processos comunicantes.

Ordinary Pipes

- Estudo o programa: unix_pipe.c

- No programa, “unix_pipe.c”, o pipe estará acessível caso o processo filho execute um “execve”?

Ordinary Pipes

- Pipe é um *stream de* bytes. Não existe o conceito de mensagem ou limite de mensagens.
- Pipe é simplesmente um *buffer* mantido na memória do Kernel.
- Os dados são transferidos sequencialmente. Não é possível acessar dados randomicamente.
- Desde o Kernel 2.6.11 do Linux, o limite do pipe é de 65536 bytes. Em versões anteriores, o limite era de 4096 bytes.
- Tentativas de leitura de um *pipe* que está vazio podem bloquear o processo leitor ou retornar um "*fail*".
- Tentativas de escrita em um pipe que está cheio podem bloquear o processo escritor ou retornar um "*fail*"

Ordinary Pipes

- Em Linux, a capacidade de um pipe pode ser modificada (*fcntl* function) até certo valor definido em /proc/sys/fs/pipe-max-size.
- Embora seja possível para os processos pai e filho lerem e escreverem de um mesmo pipe, isso não é comum pois requer um mecanismo de sincronização para evitar inconsistência dos dados.
- Comunicação bidirecional é implementada por meio de 2 pipes.

Pipes

- **Ordinary pipes** – tipicamente usados na relação pai-filho.
- **Named pipes (Linux FIFO)** – podem ser acessados fora de uma relação pai-filho.

Named Pipes / FIFOs (Unix)

- A comunicação ocorre entre processos no mesmo computador.
- Em Linux, *Named Pipes* são referidos como FIFOs.
- A comunicação é bidirecional.
  - Porém dois FIFOs são normalmente utilizados quando se deseja comunicação bidirecional.
- Vários processos podem usar um *named pipe* para comunicação.
- É mantido pelo SO até que seja explicitamente apagado do sistema.
  - *Kernel persistence*

Fim do capítulo 3