


UNIVALI
Universidade do Vale do Itajaí
Escola Politécnica
Computação

Processos


1



Sumário

- Kernel monolítico e microkernel
- Conceito de Processo
- Agendamento de Processos
- Operações com Processos
- Comunicação entre processos
- IPC com memória compartilhada
- IPC com passagem de mensagens
- Exemplos de sistemas IPC
- Comunicação em sistemas cliente-servidor - Talvez


2



Operações do Sistema Operacional

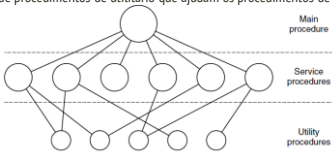
- Programa Bootstrap – código simples para inicializar o sistema, carregar o kernel
- Carregamento do kernel
- Inicia *daemons* do sistema (serviços fornecidos fora do kernel)
- Kernel controlado por interrupção (hardware e software)
 - Interrupção de hardware por um dos dispositivos
 - Interrupção de software (**exceção ou trap**):
 - Erro de software (por exemplo, divisão por zero)
 - Solicitação de serviço do sistema operacional – **system call**
 - Outros problemas de processo incluem loop infinito, processos modificando uns aos outros ou o sistema

3



Sistemas Monolíticos

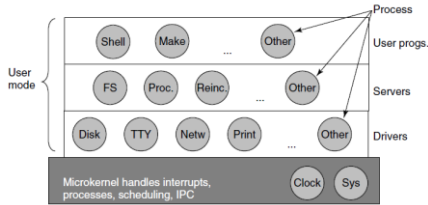
- Estrutura básica do SO
 - Um programa principal que invoca o procedimento de serviço solicitado
 - Um conjunto de procedimentos de serviço que realizam as chamadas do sistema
 - Um conjunto de procedimentos de utilitário que ajudam os procedimentos de serviço



4

Microkernels

- Estrutura simplificada do SO
- Sistema MINIX 3



5

Multiprogramação (sistema de lotes)

- Um único usuário nem sempre pode manter a CPU e os dispositivos de E/S ocupados
- A multiprogramação organiza trabalhos (código e dados) para que a CPU sempre tenha um para executar
- Um subconjunto do total de trabalhos no sistema é mantido na memória
- Um trabalho selecionado e executado por meio de escalonamento de trabalho
- Quando o trabalho precisa aguardar (para E/S, por exemplo), o sistema operacional alterna para outro trabalho

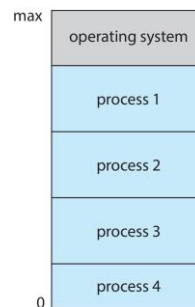
6

Multitarefa (Timesharing)

- Uma extensão lógica dos sistemas em lote – a CPU alterna trabalhos com tanta frequência que os usuários podem interagir com cada trabalho enquanto ele está em execução, criando computação interativa
 - O tempo de resposta deve ser < 1 segundo
 - Cada usuário tem pelo menos um programa em execução na memória \Rightarrow processo
 - Se vários trabalhos estiverem prontos para serem executados ao mesmo tempo \Rightarrow Escalonamento de CPU
 - Se os processos não couberem na memória, a gerência de memória os move para dentro e para fora para serem executados
 - A memória virtual permite a execução de processos que não estão completamente na memória

7

Visão da memória para sistema multiprogramado



8

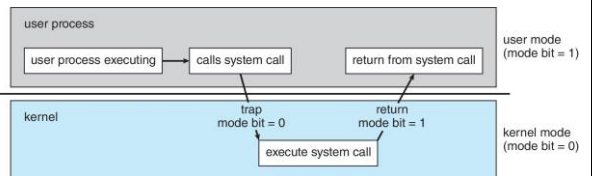
Operação de modo duplo

- A operação de modo duplo permite que o sistema operacional proteja a si mesmo e a outros componentes do sistema
 - Modo de usuário e modo kernel
- Bit de modo fornecido pelo hardware
 - Fornece a capacidade de distinguir quando o sistema está executando o código do usuário ou do kernel
 - Quando um usuário está em execução \Rightarrow bit de modo é "usuário"
 - Quando o código do kernel está em execução \Rightarrow bit de modo é "kernel"
- Como garantimos que o usuário não defina o bit de modo como "kernel"?
 - O modo de alterações de chamada do sistema para o kernel, o retorno da chamada o redefine para o usuário
- Instruções do tipo privilegiadas são somente executáveis no modo kernel



9

Transição do modo Usuário para o modo Kernel



10

Timer (temporizador)

- Temporizador para evitar loop infinito (ou recursos de consumo de processo)
 - Timer é definido para interromper o computador após algum período de tempo
 - Mantenha um contador que seja diminuído pelo relógio físico
 - O sistema operacional define o contador (instrução privilegiada)
 - Quando o contador zero gera uma interrupção
 - Configurar antes do processo de agendamento para recuperar o controle ou encerrar o programa que exceda o tempo alocado



11

Conceito de Processo

- Um sistema operacional executa uma variedade de programas que são executados como um processo
- **Processo** – um programa em execução e a execução do processo deve progredir de forma sequencial
 - Nenhuma execução paralela de instruções de um único processo
- Várias peças
 - O código do programa, também chamado de seção de texto
 - Atividade atual, incluindo contador de programas, registradores de processador
 - Pilha contendo dados temporários
 - Parâmetros de função, endereços de retorno, variáveis locais
 - **Seção de dados contendo variáveis globais**
 - **Heap** contendo memória alocada dinamicamente durante o tempo de execução



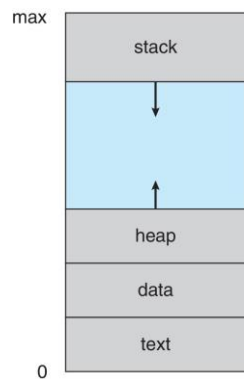
12

Conceito de Processo (Cont.)

- Programa é entidade passiva armazenada no disco (arquivo executável) e se o o processo está ativo
 - Programa torna-se processo quando um arquivo executável é carregado na memória
- Execução do programa iniciado através de cliques do mouse GUI, entrada de linha de comando de seu nome, etc
- Um programa pode ser vários processos
 - Considere vários usuários executando o mesmo programa

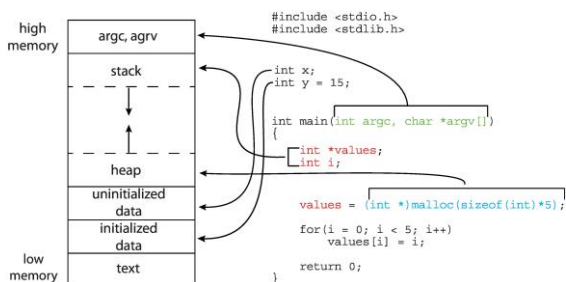
13

Processo na memória



14

Layout de memória de um programa C



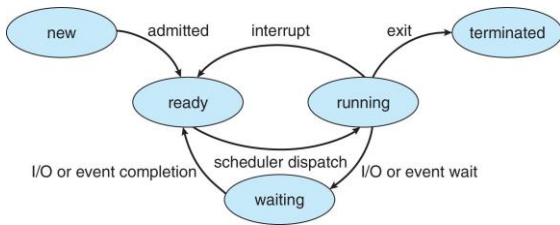
15

Estado do processo

- À medida que um processo é executado, ele muda de estado
 - Novo: o processo está sendo criado
 - Em execução: as instruções estão sendo executadas
 - Aguardando: o processo está aguardando que algum evento ocorra
 - Pronto: o processo está aguardando para ser atribuído a um processador
 - Encerrado: o processo terminou a execução

16

Diagrama do estado do processo



17

Process Control Block (PCB)

Informações associadas a cada processo (também chamado de bloco de controle de tarefas)

- Estado do processo – em execução, esperando, etc.
- Contador de programas – local da instrução para a próxima execução
- Registradores de CPU – conteúdo de todos os registradores centrados no processo
- Informações de escalonamento de CPU - prioridades, ponteiros de fila de agendamento
- Informações de gerenciamento de memória – memória alocada para o processo
- Informações de "tempo" – CPU usada, tempo de clock decorrido desde o início, limites de tempo
- Informações de status de E/S – dispositivos de E/S alocados para processar, lista de arquivos abertos

process state
process number
program counter
registers
memory limits
list of open files
...

18

Threads

- Até agora, o processo tem um único thread de execução
- Considere ter vários contadores de programas por processo
 - Vários locais podem ser executados de uma só vez
 - Várias threads de controle -> threads
- Deve, então, ter armazenamento para detalhes de thread, vários contadores de programas no PCB

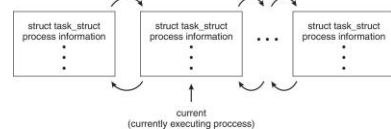
19

Representação de processos no Linux

Representado pela estrutura em C `task_struct`

```

pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
  
```



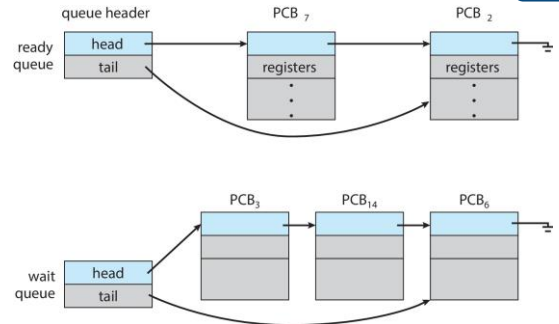
20

Escalonamento de Processos

- O escalonador de processos seleciona entre os processos disponíveis para a próxima execução no núcleo da CPU
- Objetivo -- Maximizar o uso da CPU, alternar rapidamente os processos para o núcleo da CPU
- Mantém filas de agendamento de processos
 - Fila pronta – conjunto de todos os processos que residem na memória principal, prontos e aguardando para execução
 - Filas de espera – conjunto de processos à espera de um evento (ou seja, E/S)
 - Os processos migram entre as várias filas

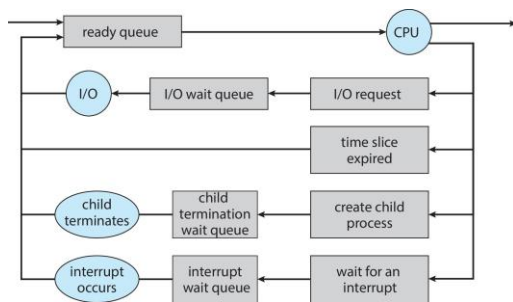
21

Filas de Pronto e Aguardando (Bloqueado)



22

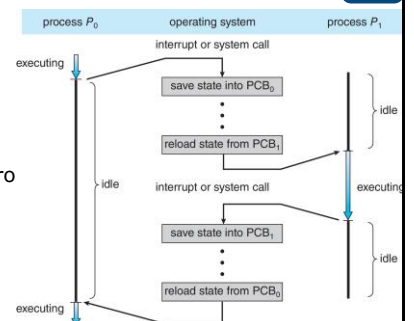
Representação do Escalonamento de Processos



23

Chaveamento de processos em CPU

Uma mudança de contexto ocorre quando a CPU alterna de um processo para outro (chaveamento de contexto)



24

Chaveamento de contexto



- Quando a CPU muda para outro processo, o sistema deve salvar o estado do processo antigo e carregar o estado salvo para o novo processo por meio de uma opção de contexto
- Contexto de um processo representado no PCB
- O tempo de mudança de contexto é pura sobrecarga
 - O sistema não faz nenhum trabalho útil durante a comutação
 - SO e PCB complexos → mais longa a mudança de contexto
- Tempo dependente do suporte de hardware
 - Alguns hardwares fornecem vários conjuntos de registradores por CPU → vários contextos carregados de uma só vez

25

Multitarefa em Sistemas Mobile



- Alguns sistemas mobile (por exemplo, a versão inicial do iOS) permitem que apenas um processo seja executado, outros suspensos
- Devido ao espaço real da tela, os limites da interface do usuário que o iOS fornece para um
 - Processo de primeiro plano único - controlado através da interface do usuário
 - Vários processos em segundo plano - na memória, em execução, mas não na tela e com limites
 - Os limites incluem uma tarefa curta, receber notificação de eventos, tarefas específicas de longa duração, como reprodução de áudio
- O Android executa em primeiro plano e em segundo plano, com menos limites
 - O processo em segundo plano usa um serviço para executar tarefas
 - O serviço pode continuar em execução mesmo que o processo em segundo plano esteja suspenso
 - O serviço não tem interface de usuário, pequeno uso de memória

26

Operações em Processos



- O sistema deve fornecer mecanismos para:
 - Criação de processos
 - Encerramento de processos

27

Criação de Processos

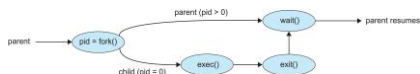


- O processo pai cria processos filhos, que, por sua vez, criam outros processos, formando uma árvore de processos
- Geralmente, o processo é identificado e gerenciado por meio de um identificador de processo (pid)
- Opções de compartilhamento de recursos
 - Pai e filhos compartilham todos os recursos
 - As crianças compartilham um subconjunto de recursos dos pais
 - Pai e filho não compartilham recursos
- Opções de execução
 - Pai e filhos executam simultaneamente
 - Pai aguarda até que os filhos terminem

28

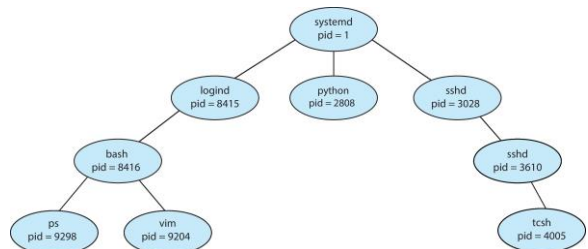
Criação de processos (Cont.)

- Espaço de endereço
 - Duplicata filho do pai
 - A criança tem um programa carregado nele
- Exemplos de UNIX
 - **fork()** chamada do sistema cria novo processo
 - **exec()** chamada do sistema usada após um **fork()** para substituir o espaço de memória do processo por um novo programa
 - As chamadas de processo do pai **wait()** aguardam o encerramento da criança



29

Uma árvore de processos no Linux



30

Processo separado de bifurcação do programa C

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
    
```

31

Criando um processo separado via API do Windows

```

#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\cmd.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }

    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
    
```

32

Finalização do processo



- O processo executa a última instrução e, em seguida, solicita que o sistema operacional a exclua usando a chamada do sistema `exit()`
 - Retorna dados de status de filho para pai (via `wait()`)
 - Os recursos do processo são alocados pelo sistema operacional
- O pai pode encerrar a execução de processos filhos usando a chamada do sistema `abort()`. Algumas razões para fazê-lo
 - Criança excedeu os recursos alocados
 - A tarefa atribuída ao filho não é mais necessária
 - O pai está saindo e os sistemas operacionais não permitem que um filho continue se seu pai for encerrado

33

Encerramento do Processo



- Alguns sistemas operacionais não permitem que o filho exista se seu pai tiver sido encerrado
 - Se um processo termina, então todos os seus filhos também devem ser encerrados
 - **terminação em cascata:** Todos os filhos, netos, etc., são finalizados
 - A finalização é iniciada pelo SO
 - O processo pai pode aguardar o encerramento de um processo filho usando a chamada `wait()` ou `system`
 - A chamada retorna informações de status e o pid do processo encerrado
- ```
pid = wait(&status);
```
- Se nenhum pai aguardando (não invocou `wait()`) processo é um zumbi
  - Se o pai for encerrado sem invocar `wait()`, o processo será órfão

34

## Hierarquia de importância do processo Android



- Os sistemas operacionais móveis geralmente precisam encerrar processos para recuperar recursos do sistema, como memória. Do mais para o menos importante:
  - Processo em primeiro plano
  - Processo visível
  - Processo de serviço
  - Processo em segundo plano
  - Processo vazio
- O Android começará a encerrar processos que são menos importantes

35

## Arquitetura multiprocesso – Navegador Chrome



- Muitos navegadores da Web eram executados como um único processo (alguns ainda o fazem)
  - Se um site causar problemas, o navegador inteiro pode travar ou falhar
- Google Chrome é multiprocessado com 3 tipos diferentes de processos:
  - **Browser:** processo gerencia interface do usuário, disco e E/S de rede
  - **Renderer:** processo renderiza páginas web, lida com HTML, Javascript. Um novo renderizador criado para cada site aberto
    - Executa em sandbox restringindo E/S de disco e rede, minimizando o efeito de explorações de segurança
  - **Plug-in:** processo para cada tipo de plug-in



Each tab represents a separate process.

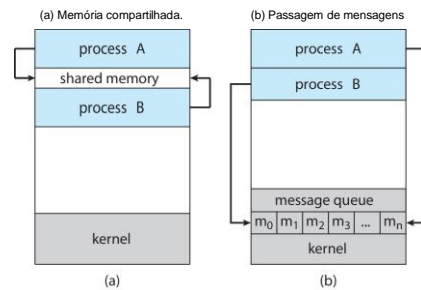
36

## Comunicação entre processos

- Os processos dentro de um sistema podem ser independentes ou cooperar
- O processo de cooperação pode afetar ou ser afetado por outros processos, incluindo o compartilhamento de dados
- Razões para os processos de cooperação:
  - Compartilhamento de informações, aceleração da computação, modularidade e conveniência
- Processos cooperantes precisam de **interprocess communication (IPC)**
- Dois modelos de IPC
  - **Memória compartilhada**
  - **Passagem de mensagens**

37

## Modelos de Comunicação



38

## Problema produtor-consumidor

- Paradigma para processos cooperativos:
  - O processo de produção produz informações que são consumidas por um processo de consumo
- Duas variações:
  - **buffer ilimitado** não impõe qualquer limite prático ao tamanho da memória intermédia:
    - Produtor nunca espera
    - O consumidor aguarda se não houver buffer para consumir
  - **buffer limitado** pressupõe que há um tamanho de buffer fixo
    - O produtor deve aguardar se todos os buffers estiverem cheios
    - O consumidor aguarda se não houver buffer para consumir

39

## IPC – Memória compartilhada

- Uma área de memória compartilhada entre os processos que desejam se comunicar
- A comunicação está sob o controle dos processos dos usuários, não do sistema operacional
- Os principais problemas são fornecer um mecanismo que permita que os processos do usuário sincronizem suas ações quando acessarem a memória compartilhada

40

## Bounded-Buffer – Solução de memória compartilhada

- Dados compartilhados

```
#define BUFFER_SIZE 10

typedef struct {
 . . .
} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

- A solução está correta, mas só pode usar elementos BUFFER\_SIZE-1

41

## Processos produtor e consumidor – Memória Compartilhada

```
item next_produced;
while (true) {
 /* produce an item in next produced */
 while (((in + 1) % BUFFER_SIZE) == out)
 ; /* do nothing */
 buffer[in] = next_produced;
 in = (in + 1) % BUFFER_SIZE;
}

item next_consumed;
while (true) {
 while (in == out)
 ; /* do nothing */
 next_consumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 /* consume the item in next consumed */
}
```

42

## Condição da Corrida

- counter++** poderia ser implementado como

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter--** poderia ser implementado como

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Considere esta execução intercalando com "contagem = 5" inicialmente:

|                        |                           |                 |
|------------------------|---------------------------|-----------------|
| S0: produtor executa   | register1 = counter       | {register1 = 5} |
| S1: produtor executa   | register1 = register1 + 1 | {register1 = 6} |
| S2: consumidor executa | register2 = counter       | {register2 = 5} |
| S3: consumidor executa | register2 = register2 - 1 | {register2 = 4} |
| S4: produtor executa   | counter = register1       | {counter = 6}   |
| S5: produtor executa   | counter = register2       | {counter = 4}   |

44

## IPC – Passagem de Mensagem

- Os processos se comunicam entre si sem recorrer a variáveis compartilhadas
- A instalação IPC oferece duas operações:
  - `send(message)`
  - `receive(message)`
- O tamanho da mensagem é fixo ou variável

45

## Passagem de Mensagem (Cont.)



- Se os processos P e Q desejam se comunicar, eles precisam:
  - Estabeleça um link de comunicação entre eles e trocar mensagens via send/receive
- Problemas de implementação:
  - Como são estabelecidos os vínculos?
  - Um link pode ser associado a mais de dois processos?
  - Quantos links podem existir entre cada par de processos de comunicação?
  - Qual é a capacidade de um link?
  - O tamanho de uma mensagem que o link pode acomodar é fixo ou variável?
  - Um link é unidirecional ou bidirecional?

46

## Implementação do Link de Comunicação



- Físico:
  - Memória compartilhada
  - Barramento de hardware
  - Rede
- Lógico:
  - Direto ou indireto
  - Síncrono ou assíncrono
  - Buffering automático ou explícito

47

## Comunicação Direta



- Os processos devem nomear uns aos outros explicitamente:
  - `send(P, message)` - enviar uma mensagem para o processo P
  - `receive(Q, message)` - receber uma mensagem do processo Q
- Propriedades do link de comunicação
  - Os links são estabelecidos automaticamente
  - Um link está associado a exatamente um par de processos de comunicação
  - Entre cada par existe exatamente um elo
  - O link pode ser unidirecional, mas geralmente é bidirecional

48

## Comunicação Indireta



- As mensagens são direcionadas e recebidas de caixas de correio (também conhecidas como portas)
  - Cada caixa de correio tem uma ID exclusiva
  - Os processos só podem se comunicar se compartilharem uma caixa de correio
- Propriedades do link de comunicação
  - Link estabelecido somente se os processos compartilharem uma caixa de correio comum
  - Um link pode estar associado a muitos processos
  - Cada par de processos pode compartilhar vários links de comunicação
  - O link pode ser unidirecional ou bidirecional

49

## Comunicação Indireta



- Operações
  - Criar um novo mailbox (porta)
  - Enviar e receber mensagens através de mailbox
  - Excluir um mailbox
- Primitivas são definidas como:
  - `send(A, message)` – enviar uma mensagem para a caixa de correio A
  - `receive(A, message)` – receber uma mensagem da caixa de correio A

50

## Comunicação Indireta (Cont.)



- Compartilhamento de Mailbox
  - $P_1, P_2$  e  $P_3$  compartilham o mailbox A
  - $P_1$  envia para  $P_2$  e  $P_3$  que recebem
  - Quem recebe a mensagem?
- Soluções
  - Permitir que um link seja associado a, no máximo, dois processos
  - Permitir que apenas um processo de cada vez execute uma operação de recebimento
  - Permita que o sistema selecione arbitrariamente o receptor. O remetente é notificado de quem era o destinatário

51

## Sincronização



- O bloqueio é considerado síncrono
  - Bloqueando o envio -- o remetente é bloqueado até que a mensagem seja recebida
  - Bloqueio de recebimento -- o receptor é bloqueado até que uma mensagem esteja disponível
- O não-bloqueio é considerado assíncrono
  - Envio sem bloqueio -- o remetente envia a mensagem e continua
  - Recebimento sem bloqueio -- o receptor recebe:
    - Uma mensagem válida, ou
    - Mensagem nula
- Diferentes combinações possíveis
  - Se tanto o envio quanto o recebimento estiverem bloqueando, teremos um encontro

A passagem de mensagens pode estar bloqueando ou não bloqueando

52

## Buffering



- Fila de mensagens anexadas ao link
- Implementado de uma das três maneiras
  1. Capacidade zero – nenhuma mensagem é enfileirada em um link  
O remetente deve aguardar o destinatário (encontro)
  2. Capacidade limitada – comprimento finito de n mensagens  
O remetente deve aguardar se o link estiver cheio
  3. Capacidade ilimitada – comprimento infinito  
O remetente nunca espera

53

## Exemplos de Sistemas IPC - POSIX

- Memória compartilhada em POSIX

- Processo primeiro cria segmento de memória compartilhada  
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
- Também usado para abrir um segmento existente
- Definir o tamanho do objeto  
`ftruncate(shm_fd, 4096);`
- Usar `mmap()` para mapear a memória de um ponteiro de arquivo para o objeto de memória compartilhada
- A leitura e a gravação na memória compartilhada são feitas usando o ponteiro retornado por `mmap()`

54

## IPC POSIX - Produtor

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
 /* the size (in bytes) of shared memory object */
 const int SIZE = 4096;
 /* name of the shared memory object */
 const char name = "OS";
 /* strings written to shared memory */
 const char *message0 = "Hello";
 const char *message1 = "World!";

 /* shared memory file descriptor */
 int shm_fd;
 /* pointer to shared memory object */
 void *ptr;

 /* create the shared memory object */
 shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

 /* configure the size of the shared memory object */
 ftruncate(shm_fd, SIZE);

 /* memory map the shared memory object */
 ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

 /* write to the shared memory object */
 sprintf(ptr, "%s", message0);
 ptr += strlen(message0);
 sprintf(ptr, "%s", message1);
 ptr += strlen(message1);

 return 0;
}
```

55

## IPC POSIX Consumidor

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

int main()
{
 /* the size (in bytes) of shared memory object */
 const int SIZE = 4096;
 /* name of the shared memory object */
 const char *name = "OS";
 /* shared memory file descriptor */
 int shm_fd;
 /* pointer to shared memory object */
 void *ptr;

 /* open the shared memory object */
 shm_fd = shm_open(name, O_RDONLY, 0666);

 /* memory map the shared memory object */
 ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

 /* read from the shared memory object */
 printf("%s", (char *)ptr);

 /* remove the shared memory object */
 shm_unlink(name);

 return 0;
}
```

56

## Pipes

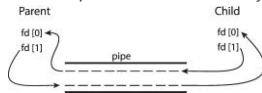
- Atua como um canal que permite que dois processos se comuniquem
- Questões:
  - A comunicação é unidirecional ou bidirecional?
  - No caso da comunicação bidirecional, é half ou full-duplex?
  - Deve existir uma relação (ou seja, pai-filho) entre os processos de comunicação?
  - Os pipes podem ser usados através de uma rede?
- **Ordinary pipes** – não pode ser acessado de fora do processo que o criou. Normalmente, um processo pai cria um pipe e o usa para se comunicar com um processo filho que ele criou
- **Named pipes** – pode ser acessado sem uma relação pai-filho

59

## Ordinary Pipes



- Ordinary Pipes permitem a comunicação no estilo padrão produtor-consumidor
- O produtor armazena em uma extremidade (a extremidade de armazenamento do pipe)
- O consumidor lê da outra extremidade (a extremidade de leitura do tubo)
- Ordinary Pipes são, portanto, unidirecionais
- Exigir relação pai-filho entre processos de comunicação



- Windows chama de **pipes anônimo**

60

## Named Pipes (Pipes Nomeado)



- Named Pipes são mais poderosos do que ordinary pipes
- A comunicação é bidirecional
- Nenhuma relação pai-filho é necessária entre os processos de comunicação
- Vários processos podem usar o Named Pipes para comunicação
- Fornecido em sistemas UNIX e Windows

61