Sistemas Operativos
3ª parte - Processos

Prof. José Rogado
jrogado@ulusofona.pt
Prof. Pedro Gama
pedrogama@gmail.com
Universidade Lusófona
Adaptação e Notas por Dr. Adriano Couto

Processos

- Conceito de Processo
- Escalonamento de Processos
- Operações em Processos
- Cooperação entre Processos
- Comunicação entre Processos

Objectivos

- O conceito de Processo
- Informação relativa ao processo
- O PCB/KPROCESS e a sua gestão
- Ciclo de Vida do Processo
  - diagrama de estados
- Intro ao Escalonamento
- Descrever Mecanismos genéricos de comunicação

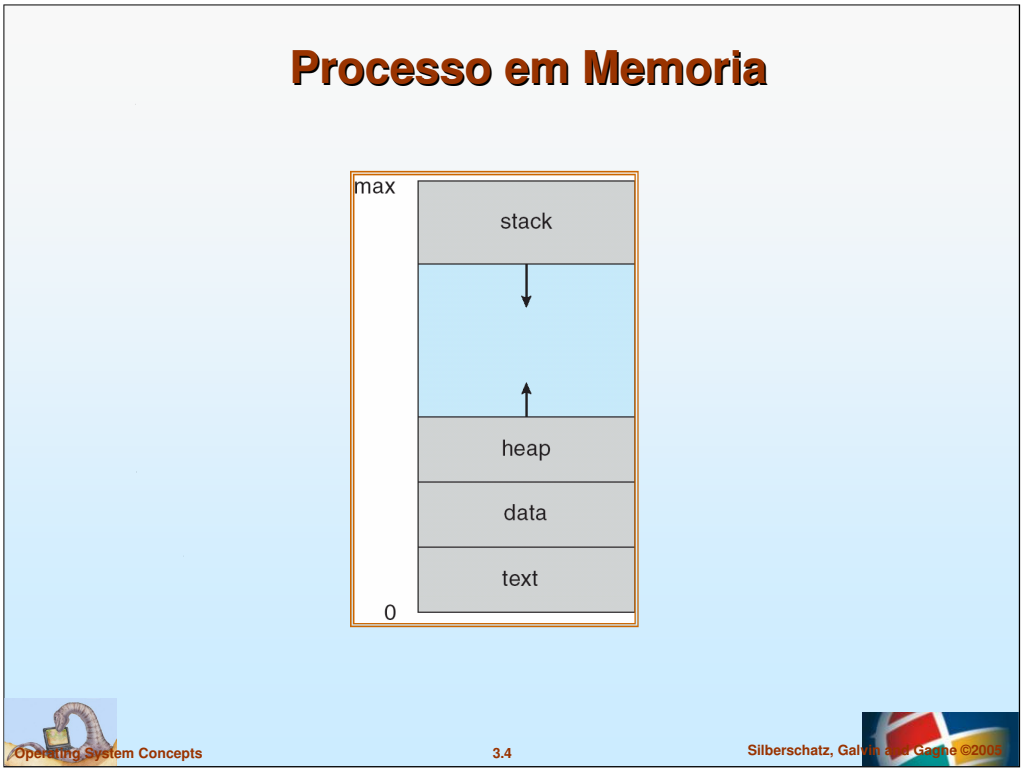
Objectivos:

Conceito de Processo

- Um Sistema Operativo executa uma variedade de programas
  - Sistemas batch - jobs
  - Sistemas em tempo repartido – processos
- **Processo – um programa em execução de forma sequencial**
- Um processo inclui:
  - Program Counter
  - Pilha (Stack) + Heap
  - Secção de Dados
  - Registos
  - Estado do IO
- Utilizaremos *job* ou *process* de forma indiferente

Perceber que o processo é algo que decorre no tempo. Não é estático.

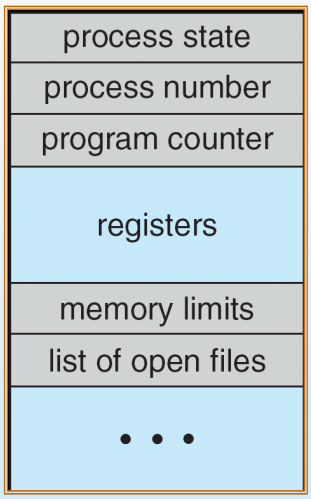
Perceber que vários programas/aplicações podem estar a correr em distintos processos.



Perceber o que, no espaço de endereçamento dois processos, apenas o segmento text, contendo o programa (instruções) pode ser comum (se os 2 processos estiverem a correr o mesmo executável)

Bloco de Controle de Processo

- Process Control Block (PCB): Informação associada a cada processo
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- No Windows chamado Kernel Process Block (KPROCESS)



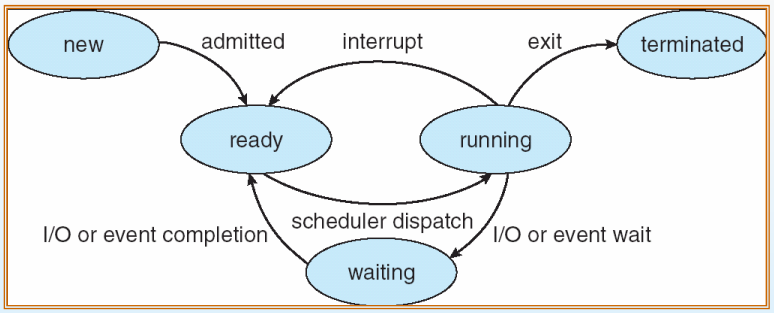
Características Dinâmicas

- Os Processos podem ter dois tipos extremos de comportamentos:
  - **I/O-bound process** – gasta mais tempo a fazer I/O do que computação
    - muitas e curtas utilizações do CPU
  - **CPU-bound process** – gasta mais tempo a fazer computação do que I/O;
    - Poucas mas longas utilizações CPU
- Este tipo de diferenças favorece o **escalonamento de processos**
  - Tira partido do facto que os processos não utilizam o CPU de forma contínua

Um programa que execute um checkcum sobre um ficheiro que tipo de comportamento apresenta?

Como se verá isso em termos do tempo dispendido em modo System e User?

Estados de um Processo

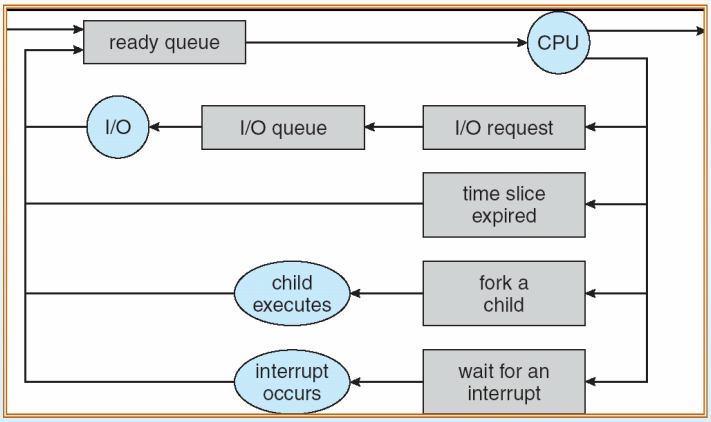


- À medida da sua execução, um processo muda de estado
  - **new**: O processo está a ser criado
  - **ready**: O processo está à espera de ser posto em execução
  - **running**: O processo está a ser executado
  - **waiting**: O processo está à espera de que ocorra um evento
  - **terminated**: O processo acabou a sua execução

Ver exemplos de cada um dos estados. Em que circuntêcnias poderemos apanhar um Processo no estado terminado?

Saber interpretar o output do comando **ps (1)** em UNIX.

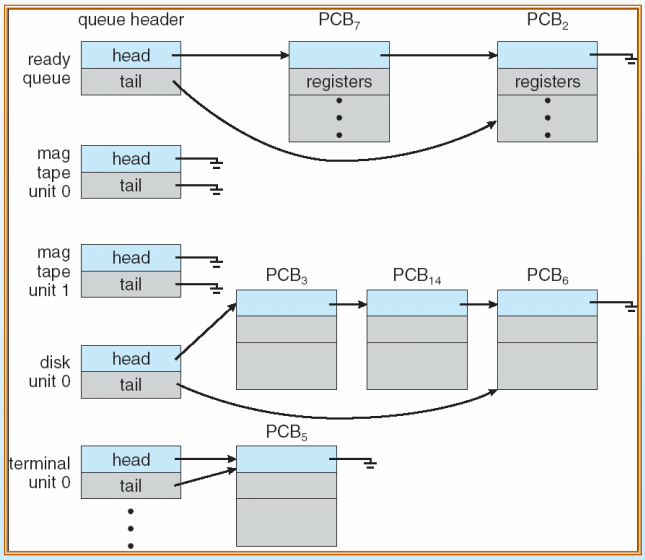
Escalonamento de Processos



- Os Processos transitam entre estas filas durante as várias fases da sua execução
- **Job queue** – conjunto de todos os processos no sistema.
- **Ready queue** – conjunto de todos os processos residentes em memoria, em estado **ready** e à espera de serem executados
- **Device queues** – conjunto de todos os processos que estão à espera de um periférico de I/O

Referir Preemptiveness e Não-Preeemptiveness. [Windows 3.1 vs Windows 95]

Ready Queue e I/O Queues



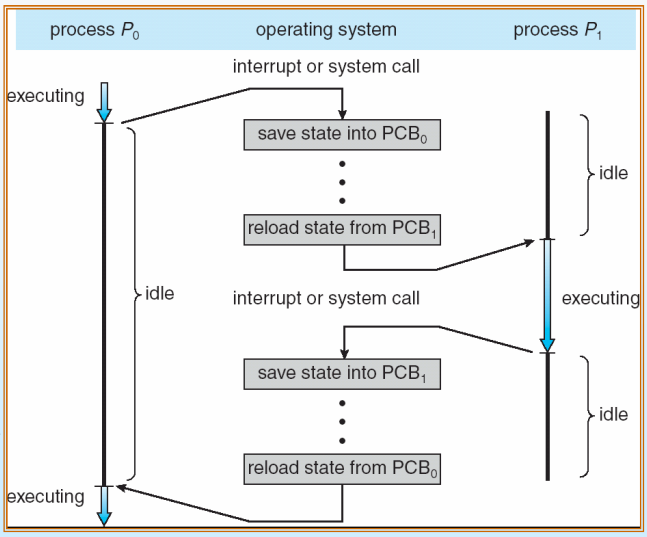
Discutir de que forma a implementação das listas [é um exercicio das práticas] facilita a comutação de um PCB de uma lista para outra.

Comutação de Processos

- Comutação de Processos
  - Quando o *scheduler* retira um processo do estado *running* e o substitui por outro
- Mudança de Contexto
  - Quando o há comutação entre dois processos, o sistema deve salvaguardar o estado do antigo processo e carregar o estado do novo processo
  - Esta operação é fundamental para que os processos possam voltar a passar para o estado *running* sem perder o contexto de execução
  - O tempo de mudança de contexto é *overhead*: o sistema não está a fazer nada de útil para o utilizador durante a comutação
    - Depende do suporte fornecido pelo hardware
    - MMU, hyperthreading, etc…

Essencial perceber o que está em causa na comutaçã de processos e os custos de CPU (overhead) que isso implica. Referir os mecanismos que os hardware põe à disposição do SO para reduzir esse oiverhead.

Passos da Comutação de Processos



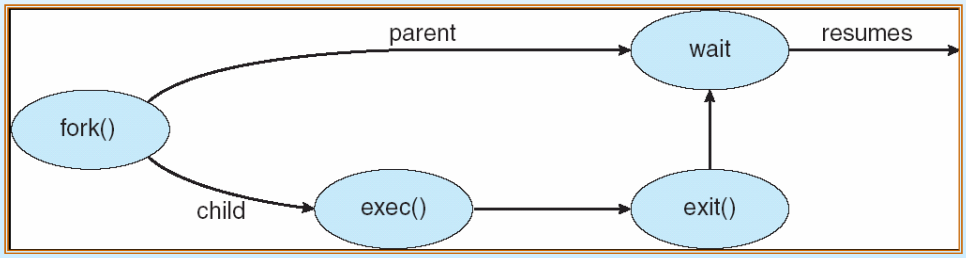
Criação de Processos

- O processo pai cria um processo filho que por seu turno pode criar outros processos, formando uma árvore de processos
  - Analogia com uma árvore genealógica
- Partilha de Recursos: diferentes possibilidades
  - O pai e filho partilham todos os recursos
  - Os recursos do filho são um subconjunto dos do pai
  - O pai e o filho não partilham recursos
- Execução
  - O pai e o filho executam-se concorrentemente
  - O pai espera que o filho termine

Perceber que há várias possibilidades na relação entre um processo e os seus descentes e que isso pode ser definido no momento em que o processo é criado.

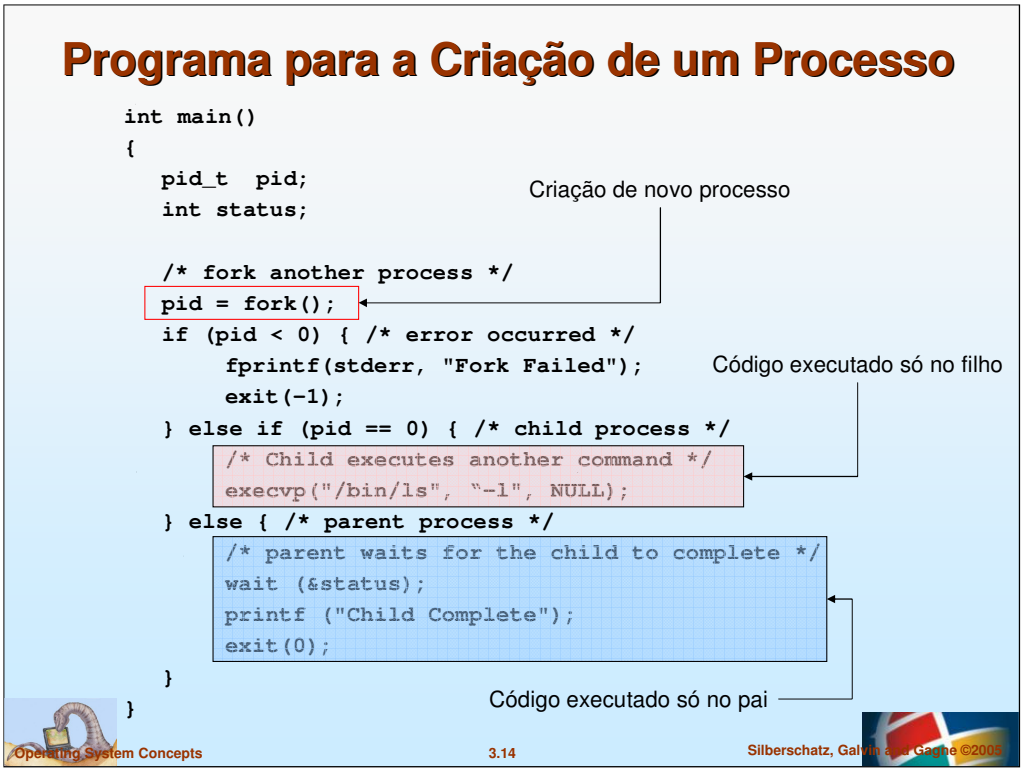
Criação de Processos (Cont.)

- Espaço de endereçamento
  - O filho duplica o do pai
  - O filho carrega outro programa no seu próprio espaço
- Exemplos UNIX
  - O system call **fork** cria um novo processo
  - O system call **exec** é utilizado depois do fork para carregar outro programa no espaço de endereçamento do filho



Essencial entender que o fork() cria um processo que é um clone e que vai o program counter na mesma instrução. Mas que dái para a frente evolui separadamente.

Perceber que o exec() “transmuta” o processo que efectua a chamada: carrega um novo executável.



É importante perceber este mecanismo, que é específico do UNIX, mas é fundamental.

A lógica assinalada nas caixas é fácil de entender.

Chamar a atenção para a referência ao executável a correr, como argumento do **exec**

Referir que há variantes do exec

Criação de Processo (Windows)

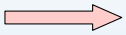
```
BOOL CreateProcess (
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpsaProcess,
  LPSECURITY_ATTRIBUTES lpsaThread,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurDir,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcInfo)
```

- CreateProcess recebe entre outros:
  - o nome do executável, argumentos de linha de comando
  - Flags que indicam se o processo vai ter a sua consola ou não, se é criado no estado SUSPENSO, se vai o pai de um novo grupo de processos
  - Se o processo vai partilhar recursos (referidos handles) do pai
  - A prioridade do novo processo
- Preenche lpProcInfo com a infor do processo criado

Referir que o conceito de criação de processos existe em todos os sistemas, mas com diferente mecanismos.

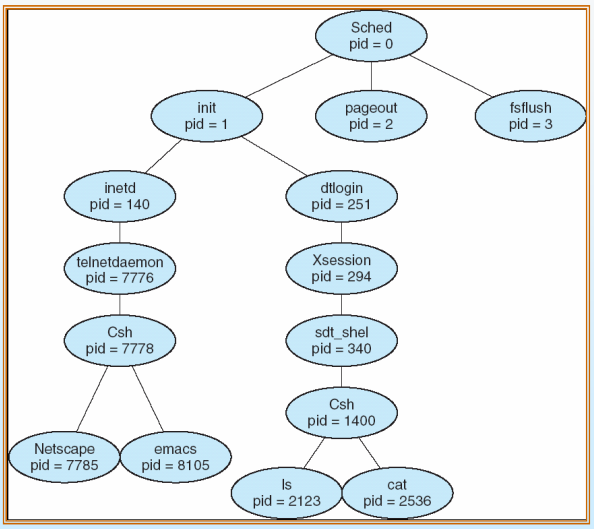
Chamar a atenção para a referência ao executável a correr.

Criação Processo Windows



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

   ZeroMemory( &si, sizeof(si) );
   si.cb = sizeof(si);
   ZeroMemory( &pi, sizeof(pi) );

   // Start the child process.
   if( !CreateProcess( NULL,   // No module name (use
command line).
      "C:\\WINDOWS\\system32\\mspaint.exe", // Command
line.
      NULL,             // Process handle not inheritable.
      NULL,             // Thread handle not inheritable.
      FALSE,             // Set handle inheritance to FALSE.
      0,               // No creation flags.
      NULL,             // Use parent's environment block.
      NULL,             // Use parent's starting directory.
      &si,              // Pointer to STARTUPINFO structure.
      &pi )             // Pointer to PROCESS_INFORMATION
structure.
   )
   {
      printf( "CreateProcess failed (%d).\n", GetLastError() );
     return -1;
   }

   // Wait until child process exits.
   WaitForSingleObject( pi.hProcess, INFINITE );

   // Close process and thread handles.
   CloseHandle( pi.hProcess );
   CloseHandle( pi.hThread );
}
```

Árvore de Processos



Perceber o interesse as árvore do ponto de vista da administração. Por exemplo, o que fazer quando o utilizador termina uma sessão interactiva.

Árvore de Processos XP

Process	PID	CPU	User Name	Description
System Idle Process	0	98.46	NT AUTHORITY\SYSTEM	
Interrupts	n/a			Hardware Interrupts
DPCs	n/a			Deferred Procedure Calls
System	4		NT AUTHORITY\SYSTEM	
smss.exe	788		NT AUTHORITY\SYSTEM	Windows NT Session Manager
csrss.exe	860		NT AUTHORITY\SYSTEM	Client Server Runtime Process
winlogon.exe	888		NT AUTHORITY\SYSTEM	Windows NT Logon Application
services.exe	936	1.54	NT AUTHORITY\SYSTEM	Services and Controller app
lsass.exe	948		NT AUTHORITY\SYSTEM	LSA Shell (Export Version)
ati2evxx.exe	2472		NT AUTHORITY\SYSTEM	ATI External Event Utility EXE Module
asghost.exe	2608		JQR-LPT\Jose Rogado	Global Virtual Card Host
explorer.exe	12612		JQR-LPT\Jose Rogado	Windows Explorer
POWERPNT....	18840		JQR-LPT\Jose Rogado	Microsoft Office PowerPoint
procexp.exe	18800		JQR-LPT\Jose Rogado	Sysinternals Process Explorer
AcroRd32.exe	19660		JQR-LPT\Jose Rogado	Adobe Reader 7.0
devcpp.exe	19732		JQR-LPT\Jose Rogado	Dev-C++ IDE

Árvore de Processos Linux

Name	PID	User%	System%	Nice	VmSize	VmRss	Login
kdeinit	3646	0,00	0,00	0	22.312	9.476	jrogado
artsd	3670	0,25	0,25	0	21.012	7.248	jrogado
evolution-alarm	3717	0,00	0,00	0	61.536	8.780	jrogado
gnome-system-mo	4012	2,51	1,25	0	25.880	11.124	jrogado
kio_file	3694	0,00	0,00	0	24.012	10.100	jrogado
klauncher	3651	0,00	0,00	0	23.744	10.032	jrogado
konqueror	3713	0,00	0,00	0	25.988	11.948	jrogado
konsole	4017	0,00	0,00	0	27.236	14.376	jrogado
bash	4018	0,00	0,00	0	3.016	1.660	jrogado
emacs	4021	0,00	0,00	0	11.524	8.252	jrogado
vi	4025	0,00	0,00	0	11.816	4.576	jrogado
ksnapshot	4028	0,25	0,25	0	26.368	13.452	jrogado
kwin	3680	1,25	0,25	0	24.820	13.196	jrogado
nautilus	3798	0,00	0,00	0	33.408	16.700	jrogado
kdesktop	3685	0,25	0,00	0	31.192	17.180	jrogado

Terminação de Processos

- Depois de executar a última instrução o processo pede ao SO para o destruir (via **exit**)
  - Pode retornar valores para o pai que os recebe via **wait**
  - Os recursos do filho são libertados pelo SO
- O pai pode terminar a execução de um processo filho (via **abort**) por vários motivos
  - O filho excedeu a sua quota de recursos
  - O processo já não é necessário
  - Se o pai termina
    - Alguns SOs não deixam que o(s) filho(s) continuem a execução
      - Todos os filhos são terminados – *terminação em cascata*

Cooperação e comunicação entre Processos

- Objectivos
  - Perceber as vantagens da cooperação entre – procesos
    - Reutilização, arquitectura, manutenção
  - Perceber as dificuldades da cooperação
    - Problemas inrínsecos à comunicação (em geral)
  - Modelos IPC

Cooperação entre Processos

- Processos **independentes** não podem ser afectados pela execução uns dos outros
- processos **cooperantes** podem afectar ou ser afectados pela execução uns dos outros
- Vantagens da cooperação entre processos
  - Partilha de informação
    - Melhor gestão de recursos (memória...)
  - Aumento de performance de computação
    - Efectuar computação enquanto esperamos pelo I/O
  - Modularidade
  - Conveniência (por exemplo actualizações partiais para o cliente; desenvolvimeno por diferentes equipas)
- Requer mais know-how e cuidado na implementação!
  - Esmagamento de dados, deadlock,...

Perceber as vantagens e desvantagens de desenhar uma aplicação como um conjunto de processos cooperantes. Referir que a maior parte das grandes e médias aplicações empresariais é constítuida por mais do que um processo.

Mesmo as aplicações de Desktop usam geralmente mais do que um processo.

Questões de Implementação

- Como são estabelecidos os canais?
- Um canal pode estar associado a mais de dois processos ?
- Quantos canais podem ser criados entre um par de processos que comunicam?
- Qual é a capacidade do canal?
- O tamanho das mensagens de um dado canal é fixo ou variável?
- Um canal é mono ou bidireccional?

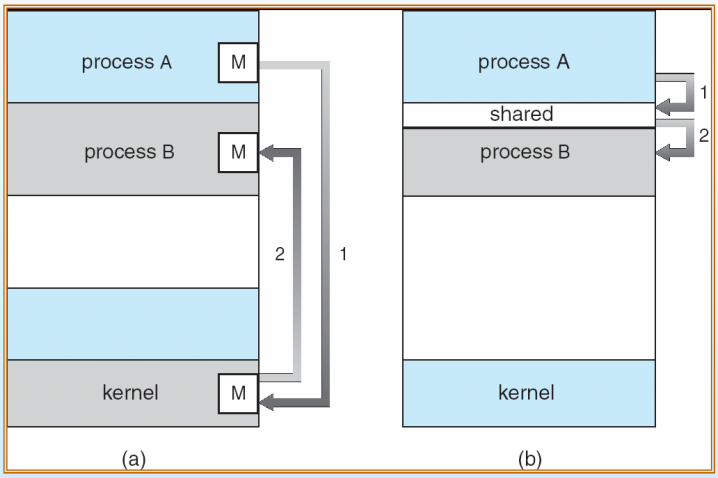
Os problemas da comunicação. Também ocorrem na vida real.

Inter Process Communication (IPC)

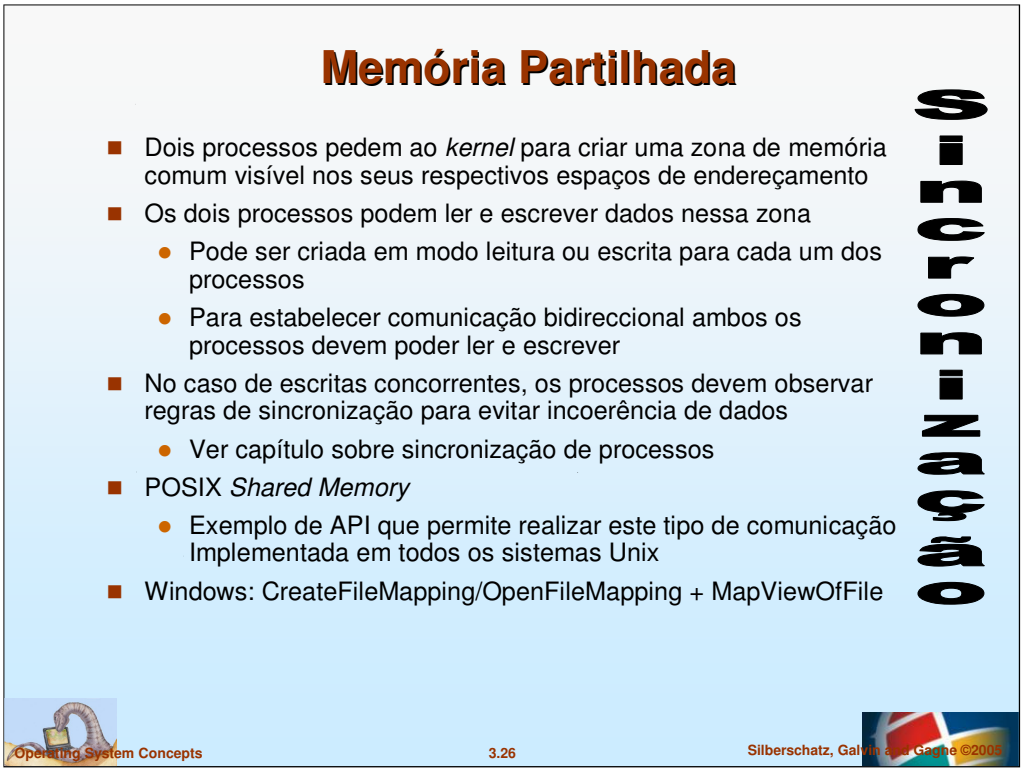
- Mecanismos que permitem aos processos comunicar e sincronizar as suas interacções
- Sistema de Mensagens – os processos comunicam sem recurso a espaços comuns aos dois processos, fornecendo duas operações:
  - **send**(*message*) – tamanho da mensagem fixo ou variável
  - **receive**(*message*)
- Sistema de Memória Partilhada - os processos pedem ao sistema para criar uma zona de memória comum
- Se *P* e *Q* desejam comunicar, precisam de:
  - Estabelecer um *canal de comunicação* entre eles
  - Trocar mensagens através das primitivas fornecidas pelo canal
- Implementação do *canal de comunicação*
  - física (e.g., memoria partilhada, bus hardware)
  - lógica (e.g., propriedades do canal)

Dois modelos de comunicação: troca de mensagens e memória partilhada

Modelos de Comunicação



(a) Mensagens (b) Memória Partilhada



A memória partilhada evidencia a necessidade de de sincronização entre processos.

Que será tratada no Cap. 6

Comunicação por Mensagens

- As mensagens são dirigidas e recebidas em *mailboxes* (também designadas por portos)
  - Cada mailbox tem um único endereço
  - Os processos podem comunicar se partilharem uma *mailbox*
- Propriedades do canal de comunicações indirecto
  - O canal é estabelecido só se os processos partilharem uma *mailbox* comum
  - Um canal pode ser associado com vários processos
  - Cada par de processos pode partilhar vários canais
  - Os canais podem ser mono ou bi-direccionais

O objectivo é perceber o conceito deste modelo de comunicação.

Apresenta menos requisitos de sincronização entre processos, pois as mensagens, podem delimitadas, os processos de escrita e leitura não conflituam. Permite que mais do que processo use o mesmo canal.

Comunicação por Mensagens (cont.)

- Operações
  - Criar uma nova *mailbox*
  - Enviar e receber mensagens através da *mailbox*
  - Apagar uma *mailbox*
- As operações definidas são:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

O objectivo é deixar claro a existências de primitivas genericamente designadas **send** e **receive** (poden os nomes várias conforme a implementação) e que há graus de liberdade adicionais na selecção dos destinatário, do receptor e tipo de mensagens.

Comunicação Indirecta (cont.)

- Partilha de *mailbox*
  - P_1, P_2, e P_3 partilham a *mailbox* A
  - P_1, envia; P_2 e P_3 recebem
  - Quem recebe a mensagem?
- Soluções
  - Só permitir canais entre dois processos
  - Permitir que só um processo de cada vez execute a operação de recepção
  - Deixar o sistema seleccionar arbitrariamente o que recebe.
    - O processo que enviou é informado de quem a recebeu.

Exemplos de Comunicação por Mensagens

- Windows:
  - **Clipboard**: permite uma troca indirecta de mensagens em que os evetuais destinatários extraem a mensagem com base no formato desta.
  - **Data Copy**: permite trocar mensagens entre aplicações que tenham janelas (podem ser invísiveis). O emissor usa a função **SendMessage** dirgida a uma janela de outra aplicação; esta recebe-a como uma mensagem
  - **MailSlots**: one-Way; usa as primitivas CreteMailSlot, ReadFile e WriteFile. Mensagens até ~400 bytes. Pouco usado
  - **Pipes**:
    - Anonymous para unidirectional pai-filho
    - Named: \\.\pipe*PipeName*
  - DDE: descontinuado

Meramente ilustrativo na aplicação de troca de mensagens.

Exemplos de Comunicação por Mensagens

- UNIX
  - **Pipes**
    - requer processos relacionados (pai/filho)
    - As mensagens são escritas atomicamente, mas não delimitadas
  - **FIFOS** ou Named Pipes
    - Criado com a chamama *mkfifo()* em que se explicita um *pathname* que permite a identificação do *pipe* por processos não relacionados.
    - As mensagens são escritas atomicamente, mas não são delimitadas
  - ***Message Queues***
    - *Mensagens persistentes,* ***delimitadas,*** escrita e leitura atómicas.

Ilustrativo do modelo de comunicação por mensagens em UNIX

Sincronização

- O envio de mensagens pode ser bloqueante ou não
- Quando há bloqueio, o envio é **síncrono**
  - Envio **síncrono**: o emissor espera até que a mensagem seja recebida
  - Recepção **síncrona**: o receptor espera até receber uma mensagem
- Quando não há bloqueio, o envio é **assíncrono**
  - Envio **assíncrono**: o emissor envia uma mensagem e continua a execução
  - Recepção **assíncrona**: o receptor é prevenido de que recebeu uma mensagem através de um evento externo (ex: interrupção)

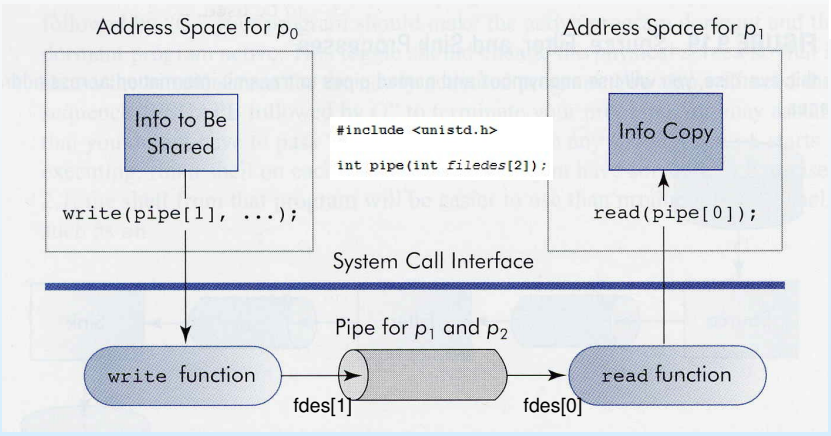
É importante perceber o que é comunicação síncrona e assíncrona.

Buffering

- O *Buffering* designa a possibilidade de existirem filas de espera de mensagens associadas a um canal
- Pode haver três tipos de *buffering*, associados ao tamanho da fila
  1. Capacidade nula – 0 mensagens
     Sender must wait for receiver (rendezvous)
  2. Capacidade limitada – tamanho finito de *n* mensagens
     Sender must wait if link full
  3. Capacidade ilimitada – tamanho infinito
     Sender never waits

É importante perceber a forma como a capacidade é relevante para o modo de comunicação.

Pipes Unix



Source: Operating Systems, Gary Nutt
Copyright © 2004 Pearson Education, Inc.

- Um pipe é um canal de comunicação criado através do sistem call *pipe()*
- São criados dois *file descriptors* que permitem enviar e receber mensagens entre dois processos pai e filho
- Um pipe é um canal *half-duplex*

Perceber que este é um tipo de comunicação por mensagens, usado nas shells, pois requer proc essos relacionados. Não é adapatado ao modelo cliente-servidor a referir mais à frente.

Programa para a Criação de um Pipe

```
int main()
{
    pid_t  pid;
    int pipeID[2];
    char buffer[64];
    char message = "Hello: Message from parent process";
    /* open a pipe */
    if (pipe(pipeID) < 0){
          perror("pipe");
          exit(-1);
    }
    pid = fork(); /* fork another process */
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
    } else if (pid == 0) { /* child process */
          /* Child reads from pipe */
          read(pipeID[0], buffer, sizeof(buffer));
          printf("Received: %s\n", buffer);
          exit(0);
    } else { /* parent process */
          /* parent writes message to pipe */
          write(pipeID[1], message, sizeof(message));
          exit(0);
    }
}
```

Criação do *pipe* de comunicação

Filho lê do *pipe* e imprime

Pai escreve no *pipe*

Fim da 3ª Parte