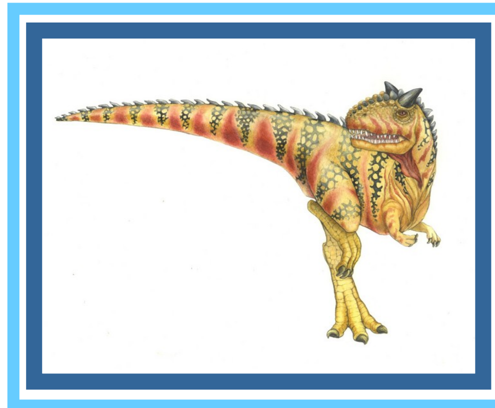
Capítulo 3: Processos



Silberschatz, Galvin and Gagne ©2009

Sobre a apresentação (About the slides)

Os slides e figuras dessa apresentação foram criados por Silberschatz, Galvin e Gagne em 2009. Esse apresentação foi modificada por Cristiano Costa (cac@unisinos.br). Basicamente, os slides originais foram traduzidos para o Português do Brasil.

É possível acessar os slides originais em http://www.os-book.com

Essa versão pode ser obtida em http://www.inf.unisinos.br/~cac

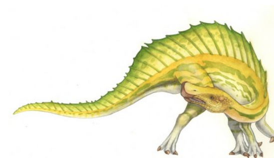
The slides and figures in this presentation are copyright Silberschatz, Galvin and Gagne, 2009. This presentation has been modified by Cristiano Costa (cac@unisinos.br). Basically it was translated to Brazilian Portuguese.

You can access the original slides at http://www.os-book.com

This version could be downloaded at http://www.inf.unisinos.br/~cac

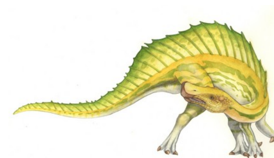
Capítulo 3: Processes

- Conceito de Processo
- Escalonamento de Processos
- Operações com Processos
- Processos Cooperativos
- Comunicação entre Processos
- Comunicação em sistemas Cliente-Servidor



Conceito de Processo

- Um sistema operacional executa uma variedade de programas:
  - Sistema Batch – *jobs*
  - Sistema Tempo Compartilhado (*Time-shared)* – programas do usuário ou tarefas
- Livros usam os termos *job* e *processo* quase que indeterminadamente.
- Processo – um programa em execução; execução do processo deve progredir de maneira seqüencial.
- Um processo inclui:
  - Contador de programa
  - Pilha
  - Seções de dados



Processo na Memória

	Região	Endereço	Descrição
máx	pilha	10000	Chamada de função, parâmetros, variáveis das funções, recursividade
		9488	Topo da pilha (endereço **decresce**)
			Se os topos se encontram temos stack overflow
		6000	Topo do heap (endereço **cresce**).
	heap		Memória dinâmica malloc(), calloc(), realloc()
	dados		
0	texto		

Estados de Processo

- Durante a execução de um processo, ele altera seu *estado*
  - **Novo** (*new*): O processo está sendo criado.
  - **Executando** (*running*): instruções estão sendo executadas.
  - **Esperando** (*waiting*): O processo está esperando algum evento acontecer.
  - **Pronto** (*ready*): O processo está esperando ser associado a um procesador.
  - **Terminado** (*terminated*): O processo terminou sua execução.

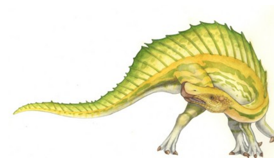


Diagrama de Estados de Processos

novo
admitido
interrupção
saída
encerrado
pronto
em execução
conclusão de I/O
ou evento
despacho do
scheduler
espera por I/O
ou evento
em espera

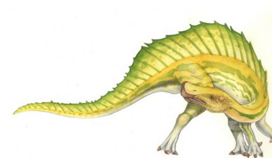
Estados de Processos

Estados

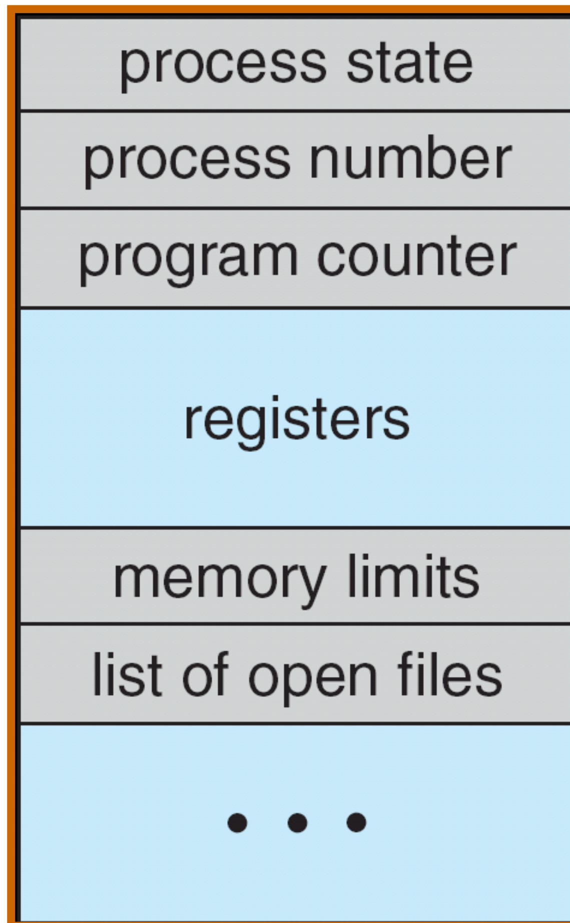
Running
Sleeping
Stoped
Zumbie

Process Control Block (PCB)

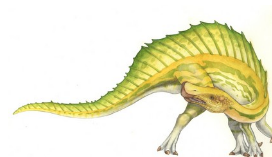
- A PCB ou Bloco de Controle de Processos armazena informações associada com cada processo.
  - Estado do Processo
  - Contador de Programas
  - Registradores da CPU
  - Informações de escalonamento da CPU
  - Informação de Gerenciamento de memória
  - Informação para Contabilidade
  - Informações do status de E/S



Process Control Block (PCB)




```
typedef struct{
        long int number;
        long int counter;
        long int rax;
        long int rbx;
        .
        .
        long int memLimit;
        Tlist *openFiles;
}Tpcb;

Tpcb* PCB;
```



Troca de CPU entre Processos

process P_0 | operating system | process P_1

interrupt or system call

executing

save state into PCB_0

•
•
•

reload state from PCB_1

idle — (ocioso)

(ocioso) — idle

interrupt or system call

executing

save state into PCB_1

•
•
•

reload state from PCB_0

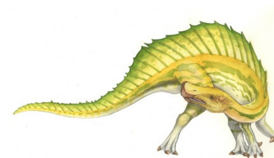
idle

executing

Filas de Escalonamento de Processos

Um **escalonador** tem a função de escolher qual processo estará pronto, executando, bloqueado, em espera, terminado ou abortado. Decide quanto tempo um processo pode ser executado, a prioridade dos processos. É como um técnico de basquete, que coloca e tira jogadores da quadra.

- **Fila de *Job*** – conjunto de todos os processos no sistema.
- **Fila de Processos prontos** (*Ready queue*) – conjunto de todos os processos residentes na memória principal, prontos e esperando para executar.
- **Fila de dispositivos** – conjunto dos processos esperando por um dispositivo de E/S.
- Migração de processos entre as várias filas.



Fila de Processos Pronto e Várias Filas de E/S

queue header

PCB_7

PCB_2

ready queue

head

tail

registers

registers

mag tape unit 0

head

tail

mag tape unit 1

head

tail

PCB_3

PCB_{14}

PCB_6

disk unit 0

head

tail

PCB_5

terminal unit 0

head

tail

Representação de Escalonamento de Processos

fila de prontos

CPU

I/O

fila de I/O

solicitação de I/O

período de tempo expirado

o filho é executado

cria um filho

ocorre uma interrupção

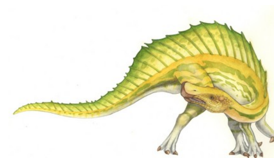
espera uma interrupção

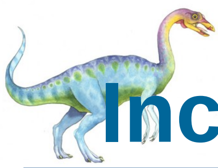
Exemplo de escalonamento

TEMPO	P1	P2	NOTAS	
1	EXECUTANDO	PRONTO		
2	EXECUTANDO	PRONTO		
3	EXECUTANDO	PRONTO		
4	EXECUTANDO	PRONTO	P1 CONCLUÍDO	
5	-	EXECUTANDO		
6	-	EXECUTANDO		
7	-	EXECUTANDO		
8	-	EXECUTANDO	P2 CONCLUÍDO	
ESTES PROCESSOS PODERIAM ALTERNAR AS OCUPAÇÕES DA CPU				
	P1	P2	NOTAS	
TEMPO	EXECUTANDO	PRONTO		
1	EXECUTANDO	PRONTO		
2	EXECUTANDO	PRONTO	P1 INICIA E/S	
3	BLOQUEADO	EXECUTANDO		
4	BLOQUEADO	EXECUTANDO		
5	BLOQUEADO	EXECUTANDO		
6	PRONTO	EXECUTANDO	E/S TERMINADO	
7	PRONTO	EXECUTANDO	P2 CONCLUÍDO	
8	EXECUTANDO	-		
9	EXECUTANDO	-	P1 CONCLUÍDO	

Escalonadores

- **Escalonador de Longo Prazo** (ou escalonador de *Jobs*) – seleciona quais processos devem ser trazidos para a fila de processos prontos.
- **Escalonador de Curto Prazo** (ou escalonador da CPU) – seleciona qual processo deve ser executados a seguir e aloca CPU para ele.





Inclusão do Escalonador Intermediário

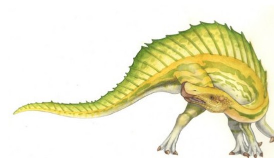
inserção
processos extraídos executados parcialmente
extração
fila de prontos
CPU
fim
I/O
filas de espera de I/O

Escalonadores (Cont.)

- Escalonador de curto prazo é invocado muito freqüentemente (milisegundos) ⇒ (deve ser rápido).
- Escalonador de longo prazo é invocada muito infreqüentemente (segundos, minutos) ⇒ (pode ser lento).
- O escalonador de longo prazo controla o *grau de multiprogramação.*
- Processos podem ser descritos como:
  - **Processos com E/S predominante** (I/O-*bound process)* – gasta mais tempo realizando E/S do que computando, muitos ciclos curtos de CPU.
  - **Processos com uso de CPU predominante** (CPU-*bound process)* – gasta mais tempo realizando computações; poucos ciclos longos de CPU.

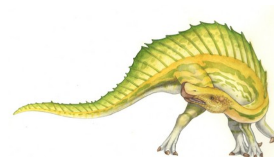
Troca de Contexto

- Quando CPU alterna para outro processo, o sistema deve salvar o estado do processo deixando o processador e carregar o estado anteriormente salvo do processo novo via **troca de contexto**.
- Contexto de um processo é representado na PCB
- Tempo de troca de contexto é sobrecarga no sistema; o sistema não realiza trabalho útil durante a troca de contexto.
- Tempo de Troca de Contexto é dependente de suporte em hardware.



Criação de Processos

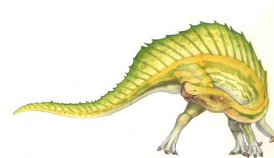
- Processo pai cria processo filho, o qual, por sua vez, pode criar outros processos, formando uma árvore de processos.
- Geralmente, processos são identificados e gerenciados via um **Identificador de Processos** (*Process IDentifier - **PID***)
- Compartilhamento de Recursos
  - Pai e filho compartilham todos os recursos.
  - Filho compartilha um subconjunto dos recursos do pai.
  - Pai e filho não compartilham recursos.
- Execução
  - Pai e filho executam concorrentemente.
  - Pai espera até filho terminar.



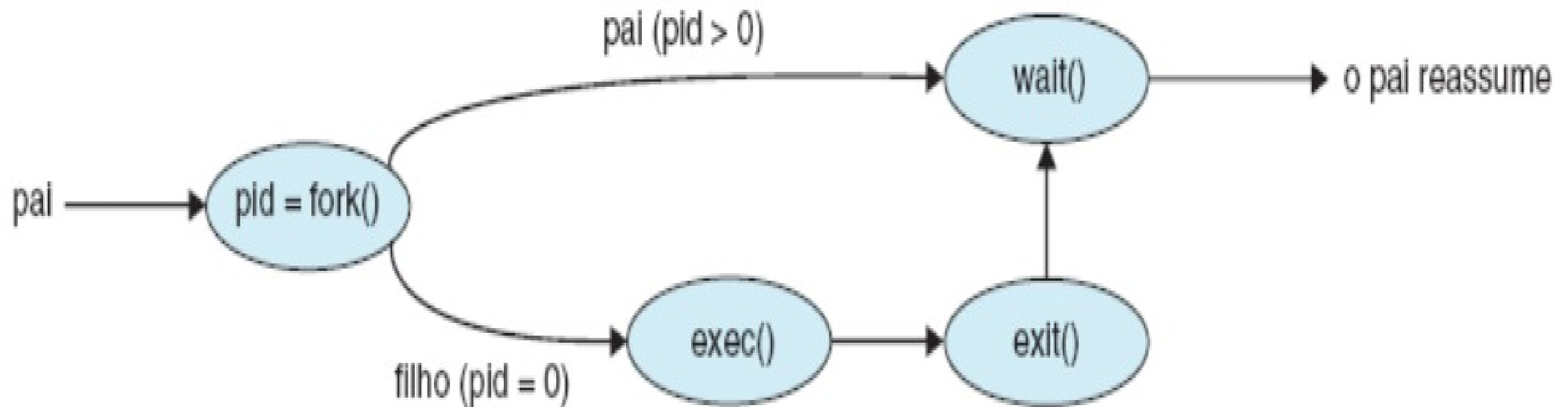
Criação de Processos (Cont.)

- Espaço de endereçamento
  - Filho duplica espaço do pai.
  - Filho tem um programa carregado no seu espaço.
- Exemplos no UNIX
  - Chamada de sistemas ***fork*** cria um novo processo.
  - Chamada de sistemas ***exec*** é usada após o ***fork*** para sobrescrever o espaço de memória do processo com um novo programa.

Existem várias versõe exec(), como execl(), execlp(), execle(), execv(), execvp().
Veja mais em: https://www.dca.ufrn.br/~adelardo/cursos/DCA409/node39.html

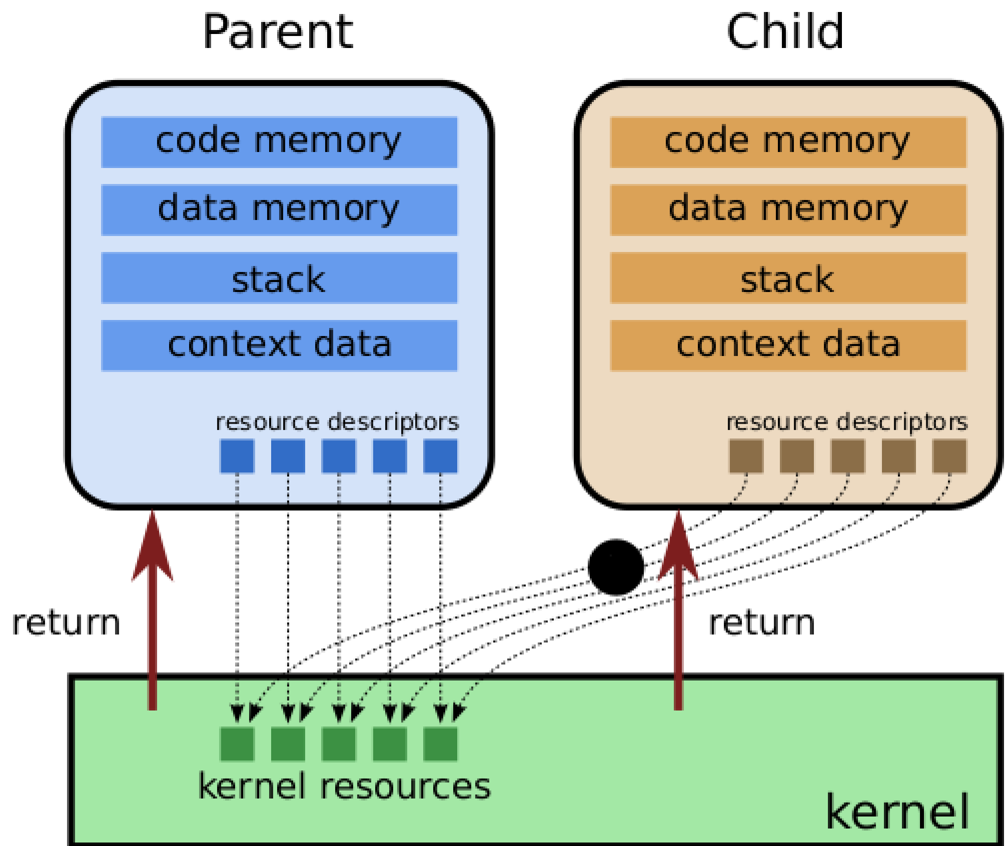


Criação de Processos (Cont.)



fork() cria um filho como cópia da estrutura do pai, mas é uma cópia. Embora as variáveis tenham o mesmo deslocamento e nome, estão em seções diferentes.

Criação de Processos (Cont.)



Criação de Processos (Cont.)

PAI	FILHO
Code Memory Data Memory Stack	Code Memory Data Memory Stack
… fork() → … …. wait() ← …. exit()	…. ….. …. exit()

wait() é uma função de sistema blocante. O processo pai fica parado até o processo filho termine.

Criação de Processos (Cont.)

PAI	FILHO
Code Memory Data Memory Stack	Code Memory Data Memory Stack
… fork() → … …. exit()	…. ….. …. exit()

Sem **wait()** o processo pai não coleta ou espera que o filho termine. Ou seja pode terminar antes do filho.

Filho zumbi

```
1: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─agetty
        ├─avahi-daemon───avahi-daemon
```

O processo pai deve esperar o término do processo filho coletando dados através do **wait()**.

Quando processo filho termina e o pai não pega os resultados do filho, este se torna um zumbi.

É um processo não se consegue nem matar, *pois enquanto o processo pai está executando o processo filho não pode ser morto*. Não ocupa memória, não está executando, mas está lá.

Se o processo pai morre antes do filho, o filho não consegue enviar os dados processados para o pai. Nesse caso o processo **systemd** (é o init da máquina) assume o filho.

O processo init é pai de todos os processos.

Criando filho zumbi

criaZumbi.c

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void fazoqueopaifaz(){
    printf("%d: Sou o pai\n",getpid());
}

void fazoqueofilhofaz(){
    printf("%d: Sou o filho\n",getpid());
    exit(0); // aborta, filho morre na hora
    printf("Ainda vivo?"); // não vai aparecer
}

int main(void){
    pid_t p;
    p=fork();
    if(p>0)
        fazoqueopaifaz();
    else if(p==0)
            fazoqueofilhofaz();
         else
            printf("FORK falhou");
    sleep(50);
}
```

1: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS

```
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$ ./criaZumbi
35481: Sou o pai
35482: Sou o filho
```

2: ojacques@ojacques-LinuxI7: ~

```
        │                  ├─{pulseaudio}(1499)
        │                  └─{pulseaudio}(1539)
        ├─scp-dbus-servic(17744)─┬─{scp-dbus-servic}(17755)
        │                        ├─{scp-dbus-servic}(17756)
        │                        ├─{scp-dbus-servic}(17768)
        │                        └─{scp-dbus-servic}(17776)
        ├─tilix(33914)─┬─bash(33924)───criaZumbi(35481)───criaZumbi(35482)
        │              ├─bash(34936)───pstree(35484)
        │              ├─{tilix}(33915)
```

Criando filho zumbi

```
1 / 1                    Tilix: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS
1: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$ ./criaZumbi
35882: Sou o pai
35883: Sou o filho

2: ojacques@ojacques-LinuxI7: ~
(base) ojacques@ojacques-LinuxI7:~$ ps x
```

```
1: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$ ./criaZumbi
35963: Sou o pai
35964: Sou o filho

2: ojacques@ojacques-LinuxI7: ~
  27451 ?        Sl     0:07 kolourpaint
  28701 ?        Sl     0:36 xreader /home/ojacques/Dropbox/AULAS ON LINE/SO/Livr
  28716 ?        SLl    0:00 /usr/lib/x86_64-linux-gnu/webkit2gtk-4.0/WebKitNetwo
  29895 ?        Sl     0:17 /usr/lib/firefox/firefox-bin -contentproc -childID 8
  31286 ?        Sl     0:37 xed /home/ojacques/Dropbox/AULAS ON LINE/SO/FORKS/cr
  33914 ?        Sl     0:29 /usr/bin/tilix --gapplication-service
  33924 pts/0    Ss     0:00 /bin/bash
  34039 ?        Sl     0:06 /usr/bin/gnome-screenshot --gapplication-service
  34936 pts/2    Ss     0:00 /bin/bash
  35820 ?        Sl     0:00 /usr/lib/firefox/firefox-bin -contentproc -childID 1
  35858 ?        Sl     0:00 /usr/lib/firefox/firefox-bin -contentproc -childID 1
  35900 ?        Sl     0:00 /usr/lib/firefox/firefox-bin -contentproc -childID 1
  35938 ?        Sl     0:00 /usr/lib/firefox/firefox-bin -contentproc -childID 1
  35963 pts/0    S+     0:00 ./criaZumbi
  35964 pts/0    Z+     0:00 [criaZumbi] <defunct>
  35965 pts/2    R+     0:00 ps x
```

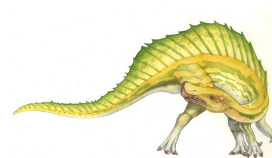
Programa em C Criando Processos Separados

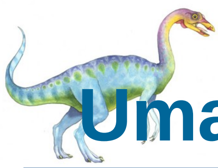
```
int main()
{
Pid_t  pid;
      /* cria outro processo */
      pid = fork();
      if (pid < 0) { /* ocorrência de erro*/
                  fprintf(stderr, “Criação Falhou");
                  exit(-1);
      }
      else if (pid == 0) { /* processo filho*/
                  execlp("/bin/ls", "ls", NULL);
      }
      else { /* processo pai */
                  /* pai irá esperar o filho completar execução
      */
                  wait (NULL); // evita que o filho termine
      como zumbi
                  printf (“Filho Completou Execução");
                  exit(0);
      }
}
```

Após a criação de um novo processo por fork(), quem decide quem será executado primeiro é o SO, através do scheduller (escalonador).
fork() precisa de <unistd.h> e <sys/types.h>
pid_t é um tipo especificado em <sys/types.h>
No exemplo, execlp() substitui o processo filho pelo processo ‘ls’.

Funções e primitvas

- fork()
- wait(), waitpid(id)
- system()
- exec()
  - execl(),execlp(), execle(), execvp()
- Comando ps
- Comando pstree
- Comando top
- Comando htop





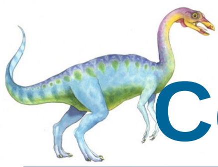
Uma Árvore de Processos em um Sistema Solaris

- Sched pid = 0
  - init pid = 1
    - inetd pid = 140
      - telnetdaemon pid = 7776
        - Csh pid = 7778
          - Netscape pid = 7785
          - emacs pid = 8105
    - dtlogin pid = 251
      - Xsession pid = 294
        - sdt_shel pid = 340
          - Csh pid = 1400
            - ls pid = 2123
            - cat pid = 2536
  - pageout pid = 2
  - fsflush pid = 3

Veja comando pstree (linux)

Terminação de Processos

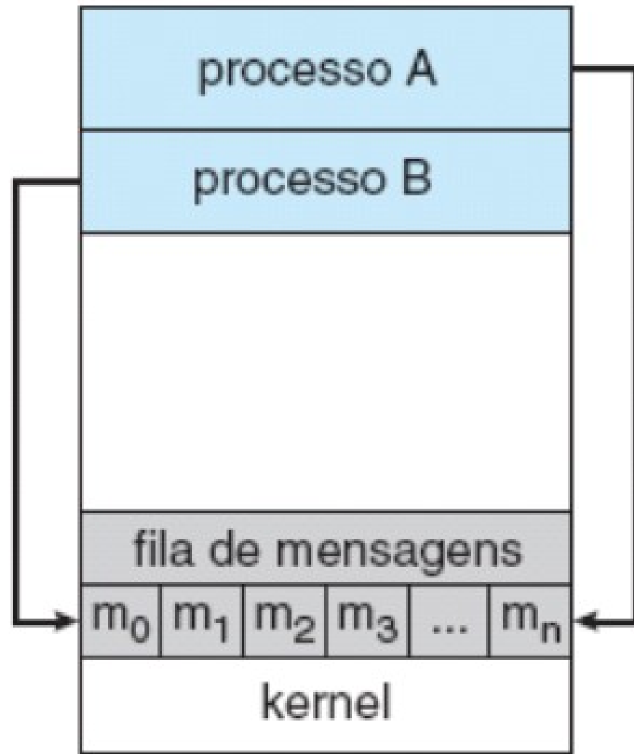
- Processo executa última declaração e pede ao sistema operacional para decidir (***exit***).
  - Dados de saída passam do filho para o pai (via ***wait***).
  - Recursos do processo são desalocados pelo sistema operacional.
- Pai pode terminar a execução do processo filho (***abort***).
  - Filho se excedeu alocando recursos.
  - Tarefa delegada ao filho não é mais necessária.
  - Pai está terminando.
    - Sistema operacional não permite que um filho continue sua execução se seu pai terminou.
    - Todos os filhos terminam - Terminação em cascata.



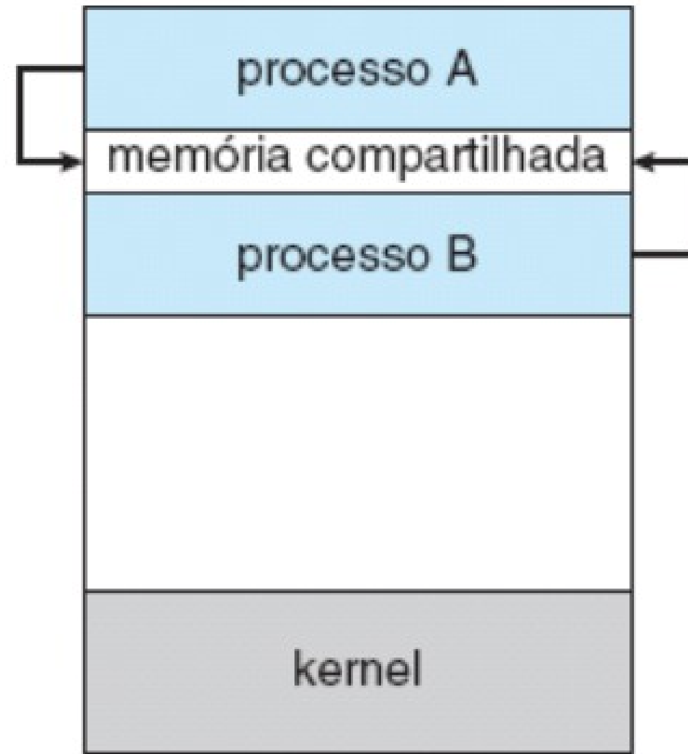
Comunicação entre Processos (IPC)

- Processos em um sistema podem ser **Independentes** ou **Cooperantes**
- Processos **Independentes** não podem afetar ou ser afetados pela execução de outro processo.
- Processos **Cooperantes** podem afetar ou ser afetados pela execução de outro processo
- Razões para cooperação entre processos:
  - Compartilhamento de Informações
  - Aumento na velocidade da computação
  - Modularidade
  - Conveniência
- Processos cooperantes precisam de **Comunicação entre Processos** (**IPC** – *interprocess communication*)
- Dois modelos de IPC: memória compartilhada e troca de mensagens

Modelos de Comunicações



Transmissão de mensagens

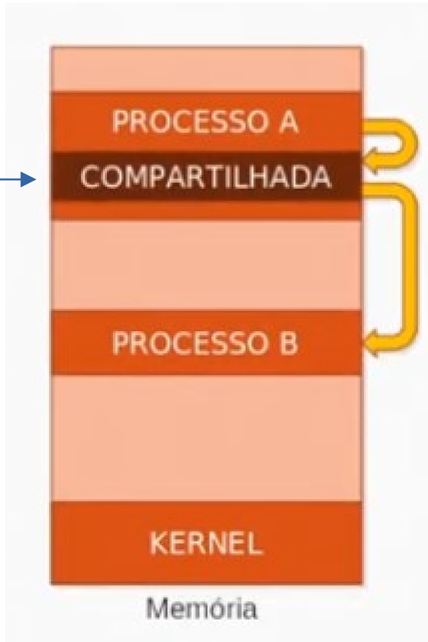


Memória compartilhada

Modelos de Comunicações

MEMÓRIA COMPARTILHADA

1)Processo A cria a memória e retorna um identificador
idSeg=shmget()



2)Processo B anexa memória
idMemSh=shmat(idSeg,..)

3)Processo A escreve na memória
sprintf(idMemSh, “Olá”)

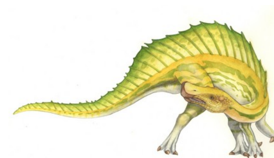
4)Processo B imprime a memória
printf(“%s”,idMemSh)

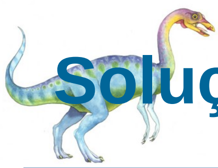
5)Processo B desanexa a memória
shmdt(idMemSh,..)

6)Processo A destroi a memória
shmctl(idSeg,..)

Problema do Produtor-Consumidor

- Paradigma para processos cooperantes, processo *produtor* produz informação que é consumida por um processo *consumidor*.
  - Buffer de tamanho ilimitado *(unbounded-buffer)* não coloca limite prático no tamanho do buffer.
  - Buffer de tamanho fixo (*bounded-buffer*) assume que existe um tamanho fixo do buffer.





Solução Buffer Tamanho Fixo - Memória Compartilhada

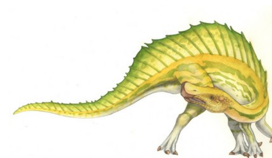
- Dados Compartilhados

```
#define BUFFER_SIZE 10
Typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solução está correta, mas somente pode usar BUFFER_SIZE-1 elementos

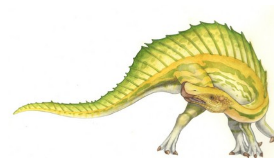
Buffer Tamanho Fixo – Produtor

```
while (true) {
   /* Produz um item */

   while (((in + 1) % BUFFER_SIZE)  == out)
         ;   /* não faz nada – sem buffers livres*/
    /* Insere um item no buffer */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
 }
```

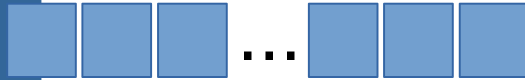


Buffer Tamanho Fixo – Consumidor

```
while (true) {
     while (in == out)
            ; /* não faz nada -- nada para consumir */
     /* Remove um item do buffer */
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
     return item;

     /* Consome um item */
 }
```



Produtor-Consumidor



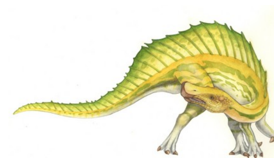

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;    //posição livre, pronta para produzir (inserir)
int out = 0;   //posição preenchida, pronta para consumir (ler, retirar)
```

```
item nextProduced;
while (true) {
  /* Produz um item */

  //buffer cheio
  while (((in = (in + 1) % BUFFER_SIZE)  == out )
        ;   /* faz nada –sem buffers livres*/
  /* Insere um item no buffer */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

```
item nextConsumed;
while (true) {
    while (in == out)  //buffer vazio
          ; /* faz nada --nada para consumir *
    /* Remove um item do buffer */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;

    /* Consome um item */
}
```

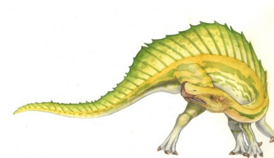
Questões de Implementação

- Como são estabelecidas as ligações?
- Pode um link estar associado com mais de dois processos?
- Quantos links podem existir entre cada par de processos comunicantes?
- Qual a capacidade de um link?
- O tamanho da mensagem utilizado pelo link é fixo ou variável?
- O link é unidirecional ou bidirecional?



Comunicação Direta

- Processos devem nomear (identificar) o outro explicitamente:
  - **send** (*P, mensagem*) – envia uma mensagem ao processo P
  - **receive**(*Q, mensagem*) – recebe uma mensagem do processo Q
- Propriedades dos links de comunicação
  - Links são estabelecidos automaticamente.
  - Um link é associado com exatamente um par de processos comunicantes.
  - Entre cada par de processos existe exatamente um link.
  - O link pode ser unidirecional, mas é usualmente bidirecional.

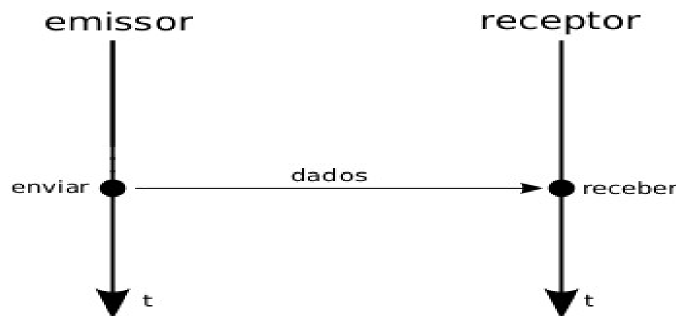


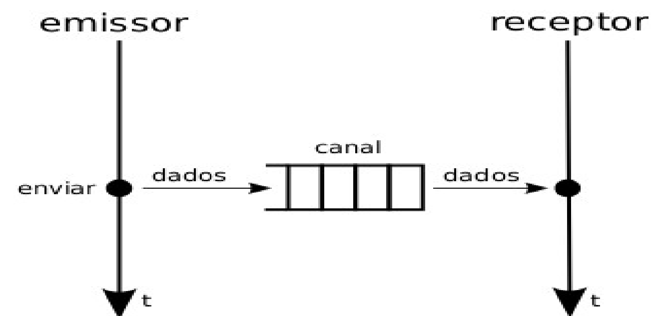
Comunicação entre Processos – Troca de mensagens

- Mecanismo para processos se comunicarem e sincronizarem suas ações.
- Sistema de mensagens – processos se comunicam uns com os outros sem utilização de variáveis compartilhadas.
- Suporte a IPC (*InterProcess Communication*) provê duas operações uma para envio outra para recebimento:
  - **send**(*mensagem*) – tamanho da mensagem fixo ou variável
  - **receive**(*mensagem*)
- Se *P* e *Q* querem se comunicar, eles necessitam:
  - Estabelecer um *link de comunicação* entre eles
  - Trocar mensagens via send/receive
- Implementação de links de comunicação
  - Físico (ex. Memória compartilha, barramento de hardware)
  - Lógico (ex. Propriedades lógicas)

Comunicação Indireta

- Mensagens são dirigidas e recebidas de caixas postais – *mailboxes* (também chamadas de *portas* ou *canal de comunicação*). Os processos não precisam se conhecer.
  - Cada *mailbox* possui uma única identificação.
  - Processos podem se comunicar somente se eles compartilham a *mailbox*.
- Propriedades do link de comunicação:
  - O link é estabelecido somente se os processos compartilham uma *mailbox* comum
  - Um link pode estar associado com muitos processos.
  - Cada par de processos pode compartilhar vários links de comunicação.
  - Link pode ser unidirecional ou bidirecional.





DIRETA

INDIRETA

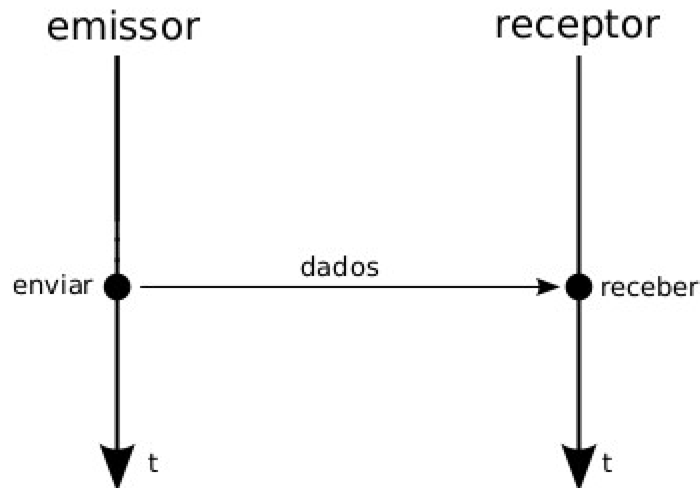
Comunicação Indireta (Cont.)

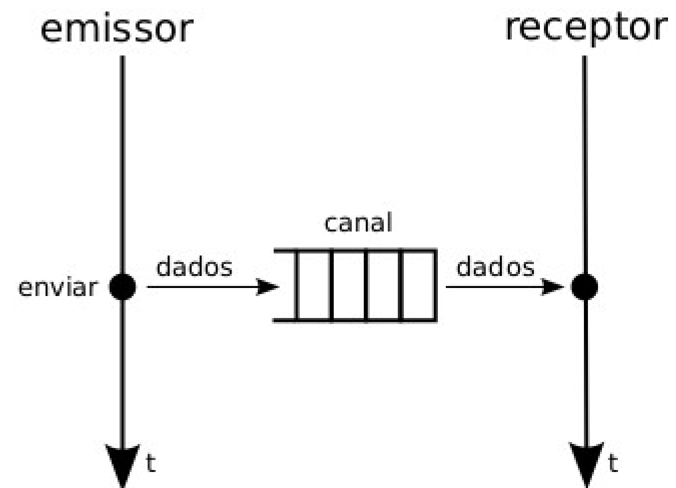
- Operações
  - Criar uma nova caixa postal
  - Enviar e receber mensagens através da caixa postal
  - Destruir uma caixa postal

- Primitivas são definidas como:

  send(*A, mensagem*) – envia uma mensagem para a caixa postal A

  receive(*A, mensagem*) – recebe uma mensagem da caixa postal A

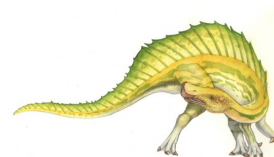




Comunicação Indireta (Cont.)

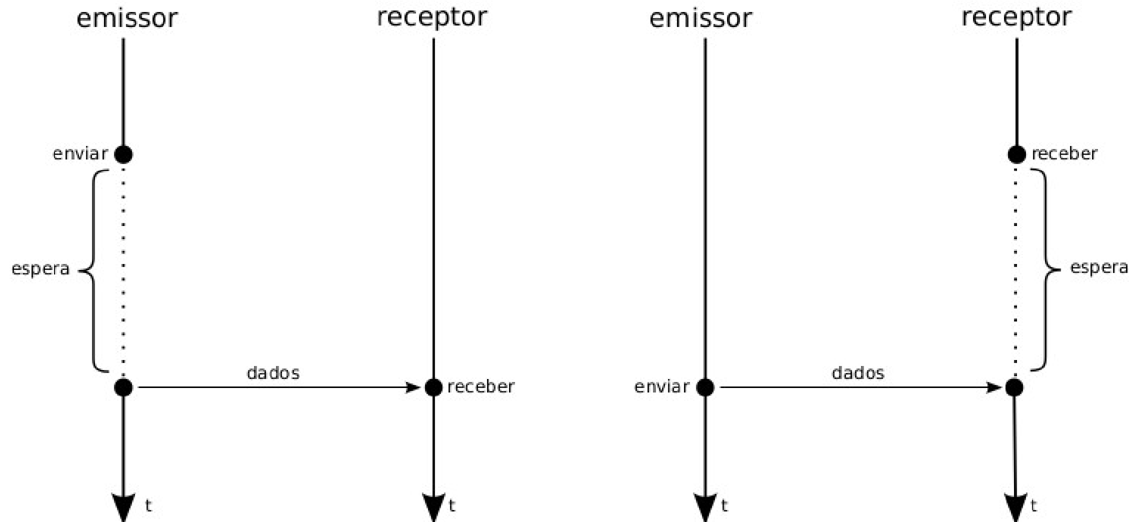
- Compartilhamento de Caixa Postal
  - P_1, P_2, e P_3 compartilham caixa postal A.
  - P_1, envia; P_2 e P_3 recebem.
  - Quem recebe a mensagem?

- Soluções:
  - Permitir que um link esteja associado com no máximo dois processos.
  - Permitir somente a um processo de cada vez executar uma operação de recebimento.
  - Permitir ao sistema selecionar arbitrariamente por um receptor. Remetente é notificado de quem foi o receptor.



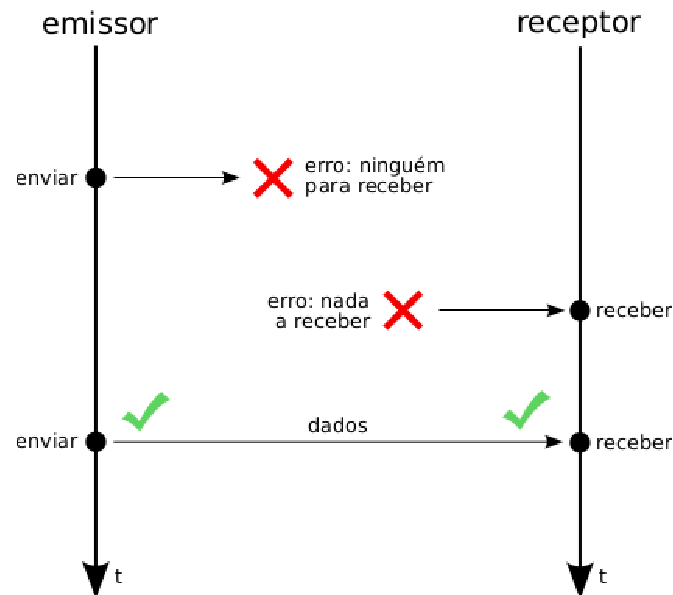
Sincronização

- Troca de Mensagens pode ser bloqueante ou não-bloqueante
- **Bloqueante** é considerado **síncrono**
  - **send Bloqueante** inibe o remetente até que a mensagem seja recebida
  - **receive Bloqueante** inibe o receptor até uma mensagem estar disponível



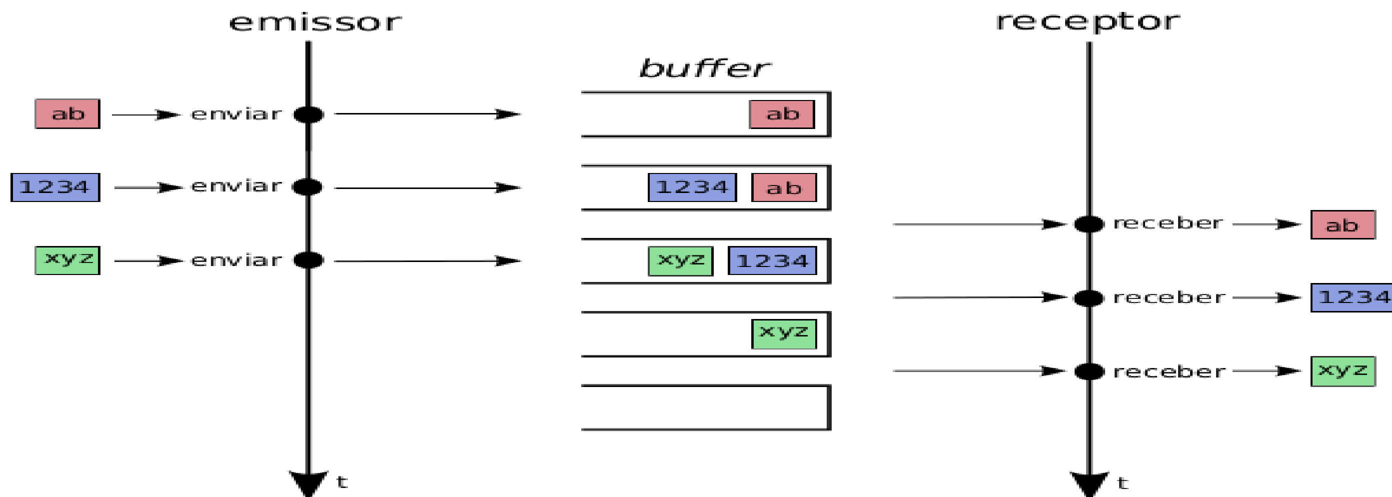
Sincronização

- **Não-Bloqueante** é considerado **assíncrono**
  - **send Não-bloqueante** o remetente envia a mensagem e continua executando
  - **receive Não-bloqueante** o receptor obtém uma mensagem válida ou null



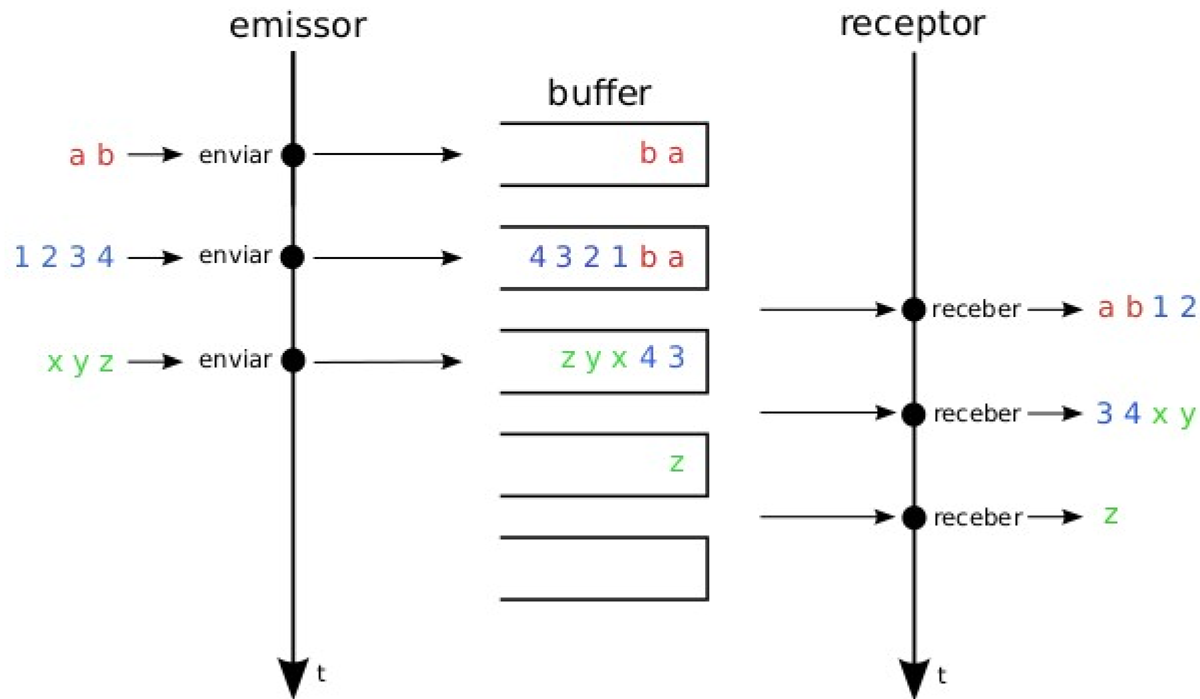
Bufferização

- **Fila de mensagens** associada ao link; implementada em uma dentre três formas.
  1. Capacidade Zero – 0 mensagens
     Remetente deve esperar pelo receptor (*rendezvous*).
  2. Capacidade Limitada – tamanho finito de n mensagens
     Remetente deve aguardar se link está cheio.
  3. Capacidade Ilimitada – tamanho infinito
     Remetente nunca espera.



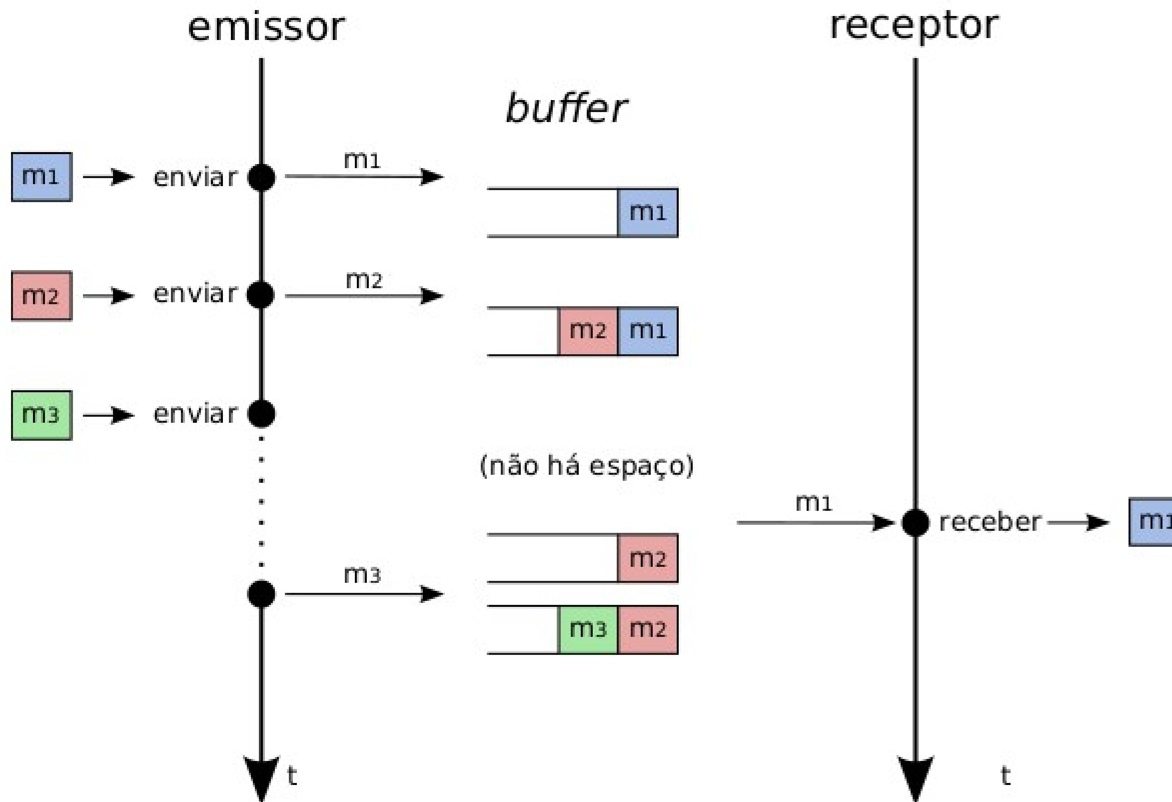
Bufferização

- **Fluxo de mensagens** contínuo associada ao link.



Bufferização

- Comunicação com capacidade de 2 filas.



Exemplos de Sistemas IPC - POSIX

- Memória Compartilhada no POSIX
  - Processo cria primeiro um segmento de memória compartilhado de 4096 bytes

```
segment_id = shmget(IPC PRIVATE, size, S IRUSR | S
   IWUSR);
```

  - Processo que deseja acesso a essa memória compartilhada deve se anexar a ela

```
 shared_memory = (char *) shmat(segment_id, NULL,
0);
```

  - Agora o processo pode escrever na memória compartilhada

```
sprintf(shared_memory, "Olá a todos");
```

  - Imprime a memória compartilhada

```
printf("%s\n",shared_memory);//imprime “Olá a
   todos”
```

  - Quando terminar, um processo pode desanexar a memória compartilhada do seu espaço de armazenamento

```
shmdt(shared_memory);
```

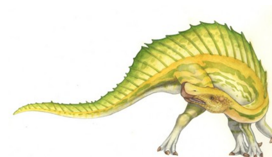
Exemplos de Sistemas IPC - Mach

- Comunicação no Mach é baseado em mensagens
  - Até mesmo chamada de sistemas são mensagens
  - Cada tarefa obtém duas *mailboxes* na criação - *Kernel* e *Notify*
  - Somente três chamadas de sistemas são necessárias para transferência de mensagens

```
msg_send(), msg_receive(), msg_rpc()
```

  - *Mailboxes* necessárias para comunicação, criadas via

```
port_allocate()
```



Exemplos de Sistemas IPC – Windows XP

- Recurso de troca de mensagens é chamado de ***local procedure call* (LPC)**
  - Só funciona entre processos no mesmo sistema
  - Usa portas (como *mailboxes*) para estabelecer e manter canais de comunicação
  - Comunicação funciona da seguinte forma:
    - O cliente abre um manipulador para o objeto porta de conexão do subsistema.
    - O cliente envia uma solicitação de conexão.
    - O servidor cria duas portas de comunicação privadas e retorna o manipulador de uma delas para o cliente.
    - O cliente e o servidor usam o manipulador da porta correspondente para enviar mensagens ou retornos de chamadas e ouvir respostas.

Local Procedure Calls no Windows XP

Cliente

Servidor

Solicitação de conexão

Porta da Conexão

Manipulador

Manipulador

Porta de Comunicação do Cliente

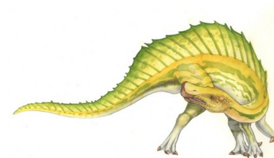
Porta de Comunicação do Servidor

Manipulador

Objeto de Seção Compartilhado (> 256 bytes)

Comunicação Cliente-Servidor

- Sockets
- Pipes
- Chamada a Procedimento Remoto (RPC)
- Invocação Remota de Método (RMI em Java)



Sockets

- Um socket é definido como um ponto final de comunicação
- Concatenação de um endereço IP e porta
- O socket **161.25.19.8:1625** refere a porta **1625** na máquina **161.25.19.8**
- Comunicação ocorre entre um par de sockets

Comunicação com Socket

host *X*
(146.86.5.20)

socket
(146.86.5.20:1625)

web server
(161.25.19.8)

socket
(161.25.19.8:80)

Comunicação com Socket

CLIENTE SOLICITA DATA E HORA

SocketClienteData.java

```
import java.net.*;
import java.io.*;
public class DateClient
{
    public static void main(String[] args) {
        try {
            /* estabelece conexão com o socket do
               servidor */
            Socket sock = new Socket("127.0.0.1",6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
            BufferedReader(new
            InputStreamReader(in));
            /* lê a data no socket */
            String line;
            while ((line = bin.readLine()) != null)
                System.out.println(line);
            /* fecha a conexão com o socket*/
            sock.close();
        }
        catch (IOException ioe) {
          System.err.println(ioe);
        }
    }
}
```

SERVIDOR RESPONDE DATA E HORA

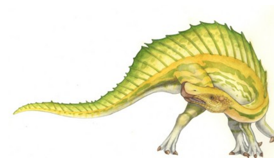
SocketServidorData.java

```
import java.net.*;
import java.io.*;
public class DateServer
{
    public static void main(String[] args) {
        /*tente executar*/
        try {
            /*cria socket na porta 6013*/
            ServerSocket sock = new ServerSocket(6013);
            /* agora escuta conexões */
            while (true) {
                Socket client = sock.accept();
                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);
                /* grava a Data no socket */
                pout.println(new java.util.Date().toString());
                /* fecha o socket e volta */
                /* a escutar conexões */
                client.close();
            }
        }
        /*se não conseguiu, trate o erro*/
        catch (IOException ioe) {/
            System.err.println(ioe);
        }
    }
}
```

Pipes

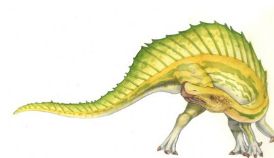
- Agem como canalizações permitindo a comunicação entre dois processos

- Questões
  - A comunicação é unidirecional ou bi-direcional?
  - No caso da comunicação de duas vias, ela é half ou full-duplex?
  - Existe uma relação (ex. Pai-filho) entre os processos comunicantes?
  - É possível usar pipes em uma rede?



Pipes Comuns

- **Pipes comuns** permitem a comunicação no estilo produtor-consumidor
- Produtor escreve em um extremo (o extremo de escrita do pipe)
- Consumidor lê do outro extremo (o extremo de leitura do pipe)
- Pipes comuns são unidrecionais
- Necessitam de relação pai-filho entre os processos comunicantes



Pipes Comuns

WritePai

WriteFilho

ReadPai

ReadFilho

Pipe

pipe

Programa com pipes

criaPipe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define IN 0
#define OUT 1
#define MAXBUFFER 100

int main(){
   pid_t pid;
   int descritoresPipe[2];
   char buffer[MAXBUFFER];
   if(pipe(descritoresPipe)){
    return 1;
   }
   pid=fork();
   if(pid <0) return 1;
   else if(pid>0){   //pai
           close(descritoresPipe[0]); //FECHA pipe de entrada
           write(descritoresPipe[1], "Bem vindo!\n",12);
       sleep(20);
           return 0;
        }
        else{
            close(descritoresPipe[1]);
            read(descritoresPipe[0], buffer, MAXBUFFER);
            printf("%s", buffer);
        sleep(20);
            return 0;
        }
}
```

Exemplo de pipe

cont.c

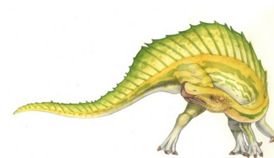
```c
#include <stdio.h>
int main(){
    char letra;
    unsigned int contador = 0;
    while(letra!=EOF){
        letra=getchar();
        if(letra=='c')
            contador++;
    }
    printf("Encontrei %d letras c\n", contador);
    return 0;
}
//CRTL + D para terminar
```

texto.txt

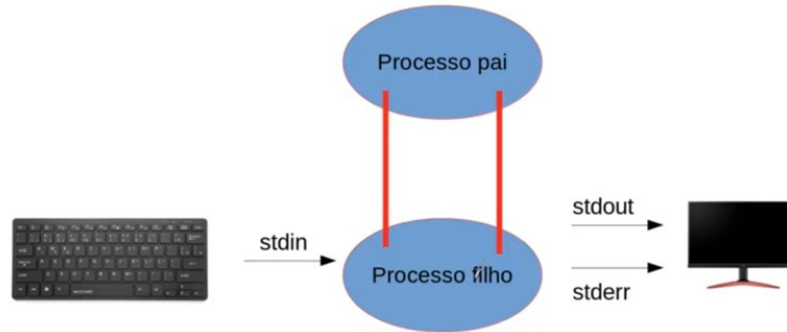
```
Meu texto tem letras c.
Porque eu gosto de letras c.
```

1: ojacques@ojacques-LinuxI7: ~/Dropbox/AULAS ON LINE/SO/FORKS

```
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$ cat texto.txt | ./cont
Encontrei 2 letras c
(base) ojacques@ojacques-LinuxI7:~/Dropbox/AULAS ON LINE/SO/FORKS$
```



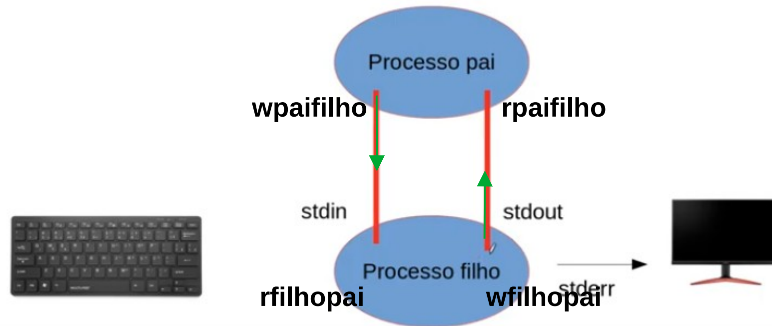
Exemplo de pipe



Usando ***dup2*** para associar entradas e saídas

rpaifilho <==> **stdin** **wfilhopai** <==> **stdout**

Tudo que o pai escrever (sair) no pipepai, o filho pode ler pipefilho



Tudo que o filho escrever (sair) no pipefilho, o pai pode ler pipepai

Exemplo de pipe

*pipeDup.py

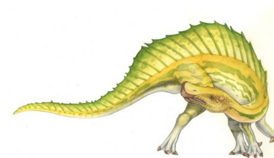
```
1  #!/usr/bin/env python3
2  #Veja video em https://www.youtube.com/watch?v=CXVTdMj-Pqc
3  import os, sys
4  comando = [ "/usr/bin/bc", "-ls" ]
5  rpaifilho, wpaifilho = os.pipe();
6  rfilhopai, wfilhopai = os.pipe();
7  processid = os.fork()
8  if processid:   # Processo pai
9      os.close(rpaifilho);  # Fecha o descritor desnecessário
10     escrita = os.fdopen(wpaifilho, 'w') # reabre como uma stream de escrita
11
12     os.close(wfilhopai); # Fecha o descritor desnecessário
13     leitura = os.fdopen(rfilhopai, 'r') # reabre como uma stream de leitura
14
15     print("Digite uma expressão (quit para sair): ")
16     linha = sys.stdin.readline()
17     while linha != "":
18         if linha == "\n":   # Se apertou enter leia novamente
19             linha = sys.stdin.readline()
20             continue
21         #escreve o que foi teclado
22         escrita.write(linha)
23         escrita.flush()
24         #captura resposta
25         linha = leitura.readline()
26         if linha != "":
27             print("Resposta: %s" % linha)
28         else:
29             break
30
31         print("Digite uma expressão (quit para sair): ")
32         linha = sys.stdin.readline()
33 else:   # Processo filho
34     os.dup2(rpaifilho, sys.stdin.fileno()) # Associa a leitura pai-filho com a entrada padrão
35     os.close(wpaifilho)   # Fecha o descritor desnecessário
36
37     os.dup2(wfilhopai, sys.stdout.fileno()) # Associa a escrita filho-pai com a saida padrao
38     os.close(rfilhopai)   # Fecha o descritor desnecessário
39
40     # Código para substituir a imagem do processo
41     os.execve(comando[0], comando, os.environ) # Substitui a imagem do programa pela calculadora bc
```

Pipes Nomeados

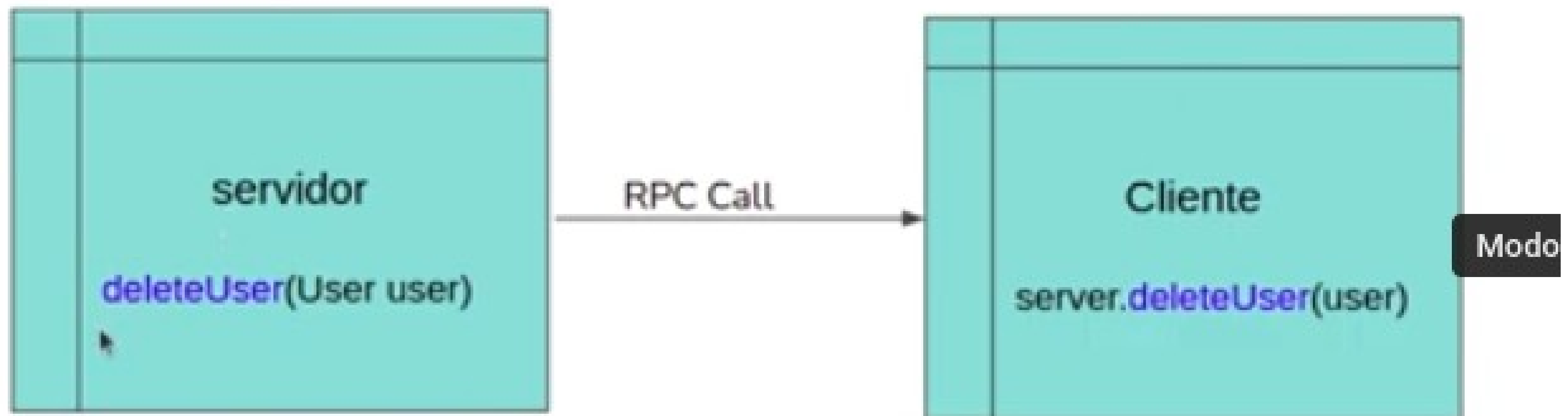
- Pipes Nomeados são mais poderosos que pipes comuns
- Comunicação é bi-direcional
- Não é necessária relação pai-filho entre processos comunicantes
- Vários processos podem usar os pipes noemados para se comunicarem
- Fornecidos nos sistemas UNIX e Windows

Chamada a Procedimento Remoto

- Chamada a Procedimento Remoto ou *Remote procedure call* (RPC) abstrai chamadas de procedimentos entre processos executando nos sistemas em rede.
- **Stubs** – proxy no lado do cliente para o procedimento real no servidor.
- O stub no lado do cliente localiza o servidor e empacota (*marshall*) os parâmetros.
- O stub no lado do servidor recebe esta mensagem, desempacota os parâmetros e dispara a execução do procedimento no servidor.



Chamada a Procedimento Remoto



Cliente solicita ao Servidor uma rotina deleteUser()
O Cliente não sabe se foi feita uma chamada remota.
Nem o servidor.

Chamada a Procedimento Remoto

Client Stub

Server Stub

Chamada → Ordena os parâmetros → Envia o request → Recebe o request → Retira os parâmetros → Executa o "procedure" retorna → Ordena os resultados → Envia o Response → Recebe o response → Retira os resultados → Retorno da chamada

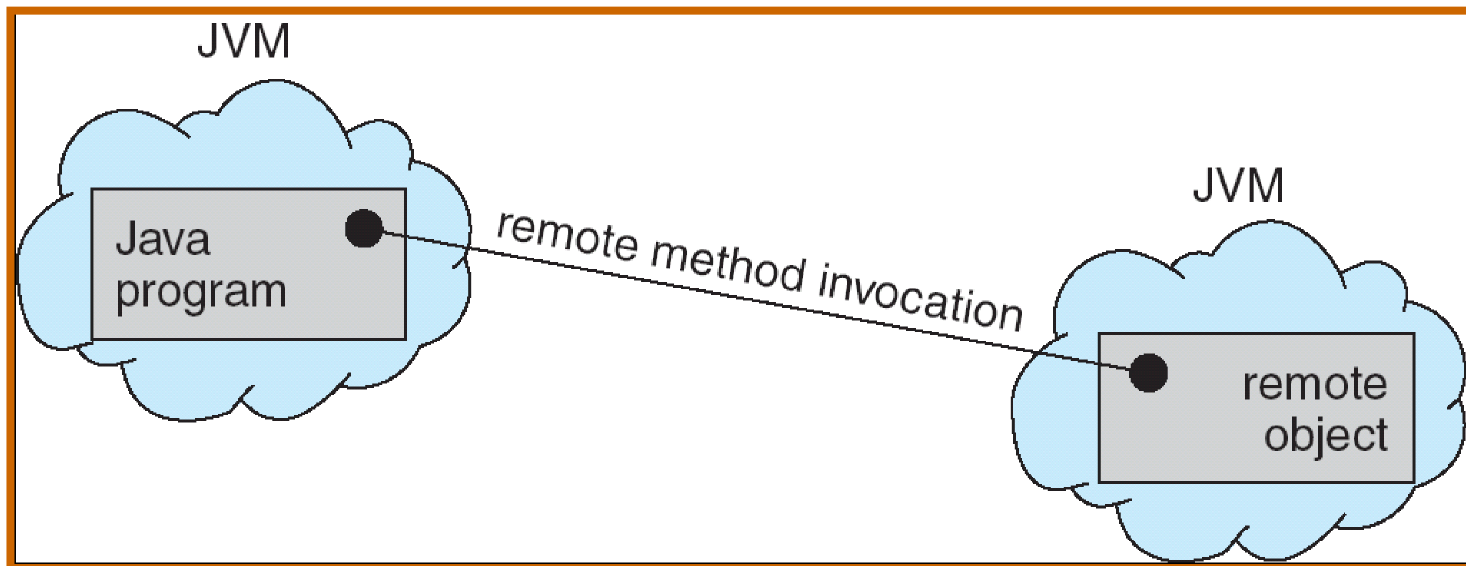
Stub é um código para cada procedimento remoto. Quando o cliente invoca um procedimento, o RPC chama o stub apropriado. Esse stub localiza a porta no servidor e empacota os parâmetros. A porta é definida por um daemon de ponto de encontro (matchmaker).

Execução de RPC

cliente	mensagens	servidor
usuário chama kernel para enviar mensagem RPC ao procedimento *X*		
kernel envia mensagem a matchmaker para encontrar número de porta	De: cliente Para: servidor Porta: matchmaker Re: endereço da RPC *X*	matchmaker recebe mensagem, procura resposta
kernel insere a porta *P* na mensagem RPC do usuário	De: servidor Para: cliente Porta: kernel Re: RPC *X* Porta: *P*	matchmaker responde ao cliente com a porta *P*
kernel envia RPC	De: cliente Para: servidor Porta: porta *P* <conteúdo>	daemon escutando na porta *P* recebe mensagem
kernel recebe resposta e a passa ao usuário	De: RPC Porta: *P* Para: cliente Porta: kernel <saída>	daemon processa solicitação e o envio da saída

Invocação Remota de Método

- Invocação Remota de Método ou *Remote Method Invocation* (RMI) é um mecanismo Java similar a RPC.
- RMI permite a um programa Java executando em uma máquina invocar um método em um objeto remoto.



Invocaçao Remota de Método - Java

IRM com métodos de uma calculadora no Servidor

Calculadora.java × clienteRPC.java × servidorRPC.java ×

```
public class Calculadora {

    public int adicao(int x, int y) {
        return x + y;
    }

    public int subtracao(int x, int y) {
        return x - y;
    }

    public int multiplicacao(int x, int y) {
        return x * y;
    }

    public double divisao(int x, int y) {
        return x / y;
    }
}
```

https://www.youtube.com/watch?v=slT__Iw9SdY&t=39s

IRM com Java

clienteRPC.java

```java
import java.net.URL;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class ClienteRPC {

    //DEFINE A URL DO SERVIDOR
    private static final String URL_SERVIDOR = "http://localhost:8185";
    private XmlRpcClient cliente;

    public ClienteRPC() {
        try {
            //configura o cliente para que ele possa se conectar ao servidor
            XmlRpcClientConfigImpl configuracaoCliente = new XmlRpcClientConfigImpl();
            configuracaoCliente.setServerURL(new URL(URL_SERVIDOR));
            //seta a configuração no cliente
            cliente = new XmlRpcClient();
            cliente.setConfig(configuracaoCliente);
        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception);
        }
    }

    public int somar(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.adicao", parametros);
        return resultado;
    }

    public int subtrair(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.subtracao", parametros);
        return resultado;
    }

    public int multiplicar(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.multiplicacao", parametros);
        return resultado;
    }

    public double dividir(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Double resultado = (Double) cliente.execute("Calc.divisao", parametros);
        return resultado;
    }
    public static void main(String[] args) throws Exception {
        ClienteRPC x = new ClienteRPC();
        System.out.println("O Resultado da soma é: " + x.somar(1, 2));
    }
}
```

servidorRPC.java

```java
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;

public class ServidorRPC {

    private ServidorRPC() {
        try {
            // Cria um servidor web na porta 8185
            WebServer ws = new WebServer(8185);
            XmlRpcServer servidor = ws.getXmlRpcServer();
            // Adiciona um novo "handler" ao PHM
            PropertyHandlerMapping phm = new PropertyHandlerMapping();
            phm.addHandler("Calc", Calculadora.class);
            // Define um handler no servidor
            servidor.setHandlerMapping(phm);
            // inicia o servidor
            ws.start();
            System.out.println("Servidor iniciado com sucesso!");

        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception);
        }
    }

    public static void main(String[] args) {
        new ServidorRPC();
    }
}
```

Calculadora.java

```java
public class Calculadora {

    public int adicao(int x, int y) {
        return x + y;
    }

    public int subtracao(int x, int y) {
        return x - y;
    }

    public int multiplicacao(int x, int y) {
        return x * y;
    }

    public double divisao(int x, int y) {
        return x / y;
    }
}
```

Calculadora.class recebeu alias de “Calc”

Invocação Remota de Método

servidorRPC.java

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;

public class ServidorRPC {

    private ServidorRPC() {
        try {
            // Cria um servidor web na porta 8185
            WebServer ws = new WebServer(8185);
            XmlRpcServer servidor = ws.getXmlRpcServer();
            // Adiciona um novo "handler" ao PHM
            PropertyHandlerMapping phm = new PropertyHandlerMapping();
            phm.addHandler("Calc", Calculadora.class);
            // Define um handler no servidor
            servidor.setHandlerMapping(phm);
            // inicia o servidor
            ws.start();
            System.out.println("Servidor iniciado com sucesso!");

        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception);
        }
    }

    public static void main(String[] args) {
        new ServidorRPC();
    }
}
```

Calculadora.java

```
public class Calculadora {

    public int adicao(int x, int y) {
        return x + y;
    }

    public int subtracao(int x, int y) {
        return x - y;
    }

    public int multiplicacao(int x, int y) {
        return x * y;
    }

    public double divisao(int x, int y) {
        return x / y;
    }
}
```

Invocação Remota de Método

clienteRPC.java

```
import java.net.URL;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class ClienteRPC {

    //DEFINE A URL DO SERVIDOR
    private static final String URL_SERVIDOR = "http://localhost:8185";
    private XmlRpcClient cliente;

    public ClienteRPC() {
        try {
            //configura o cliente para que ele possa se conectar ao servidor
            XmlRpcClientConfigImpl configuracaoCliente = new XmlRpcClientConfigImpl();
            configuracaoCliente.setServerURL(new URL(URL_SERVIDOR));
            //seta a configuração no cliente
            cliente = new XmlRpcClient();
            cliente.setConfig(configuracaoCliente);
        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception);
        }
    }

    public int somar(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.adicao", parametros);
        return resultado;
    }

    public int subtrair(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.subtracao", parametros);
        return resultado;
    }

    public int multiplicar(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Integer resultado = (Integer) cliente.execute("Calc.multiplicacao", parametros);
        return resultado;
    }

    public double dividir(int x, int y) throws Exception {
        Object[] parametros = new Object[]{new Integer(x), new Integer(y)};
        Double resultado = (Double) cliente.execute("Calc.divisao", parametros);
        return resultado;
    }
    public static void main(String[] args) throws Exception {
        ClienteRPC x = new ClienteRPC();
        System.out.println("O Resultado da soma é: " + x.somar(1, 2));
    }
}
```

Calculadora.java

```
public class Calculadora {

    public int adicao(int x, int y) {
        return x + y;
    }

    public int subtracao(int x, int y) {
        return x - y;
    }

    public int multiplicacao(int x, int y) {
        return x * y;
    }

    public double divisao(int x, int y) {
        return x / y;
    }
}
```

Fim do Capítulo 3