Capítulo 4: Processos

SUMÁRIO:

- Noção de processo
- Escalonamento de processos
- Operações sobre processos
- *Comunicação entre processos*

1

Conceito de Processo

- Processo – um programa em execução.

Um processo é mais do que um programa, pois inclui:

- **secção de texto** (ou programa)
- o conteúdo do **contador de programa** (PC, program counter)
- conteúdo dos **registos do processador**
- a **pilha** (stack) do processo com variáveis locais, funções., endereços de retorno.
- **secção de dados** que contém as variáveis globais.
  - Heap, Estatísticas etc...

Objectivo:
Esta noção de processo permite que vários utilizadores estejam a usar vários cópias do mesmo programa por exemplo **gedit, vi, firefox.**

Cada uma destes copias é um processo separado: apenas as **secções de texto** são idênticas, as secções de dados, pilha, tabelas de ficheiros abertos etc. são diferente.

2

Estrutura dum Ficheiro Executável

A estrutura de um arquivo executável depende do sistema operativo que irá carregá-lo na memória. O compilador/linker precisa de produzir um ficheiro em um dos formatos compreendidos pelo sistema operativos onde irá executar.

Formatos executável mais antigos do Unix:

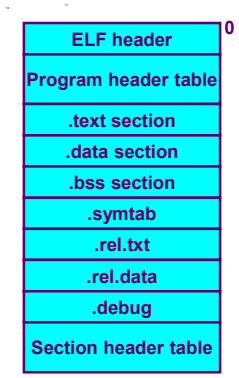
- COFF (Common Object File Format)
- a.out (assembly output) - era o nome do ficheiro de saída do assembler do PDP-7 de Ken Thompson (Unix, B, C...).

- Linux /Unix moderno
  - **ELF** (Executable and Linkable Format).
- O MS Windows
  - **PE** (Portable Executable) - derivado do formato COFF.
- O MAC OS X
  - **Mach-O** (Mach Object) - derivado do formato a.out.

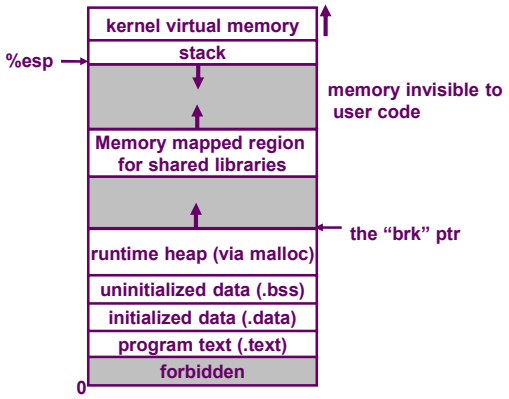
3

Representação dum Processo em Memória

Memoria Secundaria (Disco) → (virtual) Memoria Primaria (RAM)



Programa Compilado Formato ELF



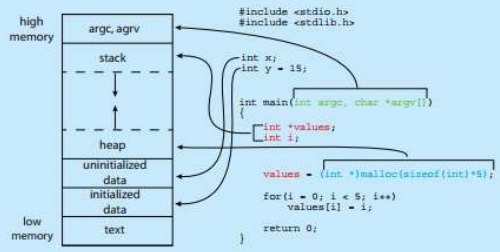
Linux/x86 Process Memory Image

4

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to ***block started by symbol***.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

5

Quiz

Where is Neo? Where is Morpheus? Somewhere in the Matrix. Based on the following code, determine in which segment are the specified variables (on the next slide) allocated.

```
#define BODY_BIT_SIZE 1000000
int A[BODY_BIT_SIZE];
extern void transfer();

void booth(char *xyz)
{
    int i;
    static int neo[BODY_BIT_SIZE];
    int *morpheus = (int *) malloc(sizeof(int) * BODY_BIT_SIZE);
    for (i = 0; i < BODY_BIT_SIZE; i++)
        morpheus[i] = neo[i];
    morpheus[0] = xyz;
    transfer();
}
int main(int argc, char *argv[])
{
    char *xyz = (char *) malloc(sizeof(char) * BODY_BIT_SIZE);
    printf("Hello?\n"); scanf("%s", xyz)
    booth(xyz);
}
```

1.	A[100]	Data	Heap	Stack
2.	i	Data	Heap	Stack
3.	morpheus	Data	Heap	Stack
4.	morpheus[0]	Data	Heap	Stack
5.	neo[10]	Data	Heap	Stack
6.	argc	Data	Heap	Stack
7.	xyz (in booth(...))	Data	Heap	Stack

This Example From:
CS 453: Operating Systems , **Amit Jain**

6

Execution

- Compiling and linking produces an **absolute** program with executable instructions and variables mapped to absolute addresses (i.e beginning with zero)
- The operating system loader maps the programs address space to the allocated primary memory(RAM) addresses
- Then it sets the PC (program counter) register to the first executable instruction (a.k.a. **start/main** entry point).
- The program begins by running a single thread of execution per cpu core.

7

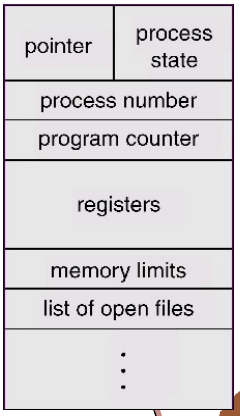
Representação Lógica dum Processo (PCB-Process Control Block)

Representação do processo:

No sistema operativo, um **processo** é representado por um **bloco de controlo do processo** (PCB-process control block).

Informação associada a cada processo:

- **Estado do processo**: ver diagramas de estados.
- **Contador de programa** (PC): indica o endereço da próxima instrução a ser executada.
- **Registos da CPU**: incluem os acumuladores, registos de indexação, ponteiros de pilha, registos gerais, etc.
- **Informação de escalonamento da CPU**: inclui prioridade do processo, ponteiros para filas de escalonamento, etc.
- **Informação de gestão de memória**: inclui valores dos registos de base e de limite, tabelas de paginação (ou tabelas de segmentação).
- **Informação de inventário:** cpu gasto, memoria usada
- **Informação de estado de I/O**: inclui lista de dispositivos I/O afectados ao processo, lista de ficheiros abertos, etc.



8

Representação Lógica dum Processo (PCB-Process Control Block)

- **O bloco de controlo do processo** (PCB) as vezes chamado "descritor de processo" é a maneira como um SO represente o conceito dum processo
- Um PCB tem que conter um identificador único (valor inteiro chamado "process ID" ou **pid**), este valor pode-se aceder através do *syscall* getpid()

Informação associada a cada processo:

- **Estado do processo**: ver diagramas de estados.
- **Contador de programa** (PC): indica o endereço da próxima instrução a ser executada.
- **Registos da CPU**
- **Informação de escalonamento da CPU**:
- **Informação de gestão de memória**:
- **Informação de inventário**
- **Informação de estado de I/O**.

9

Comutação e Escalonamento de Processos

De forma a maximizar a utilização da CPU, há que ter sempre um processo a correr, i.e. em execução.

Além disso, num dado instante, só um processo pode usar um dispositivo de entrada/saída.

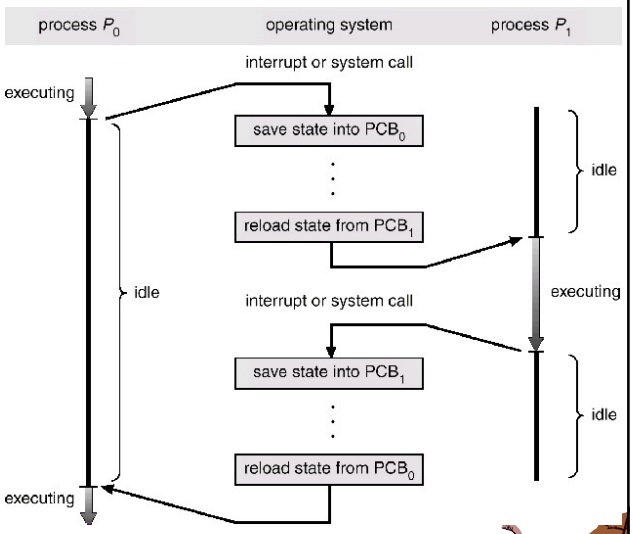
Isto obriga a que haja:

- A **troca** entre os processos em execução
  - Comutação do Contexto
- Processo existem num ***estado***
  - Estados representados por "filas" de processos
- **Escolha** dos processos nas filas e transição entre as filas
  - Escalonamento

10

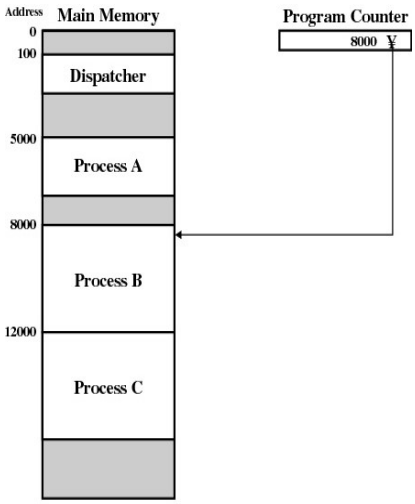
Comutação de Contexto

- A partilha do processador requer um mecanismo de comutação de processos, a que se dá o nome de *comutação de contexto.*
- Comutação entre dois processos faz-se através da:
  - **salvaguarda do estado** do processo que perde a CPU;
  - **restauração do estado** do processo que ganha a CPU.
- Comutação deve ser frequente, mas não exagerada, tem um custo overhead para o sistema.



11

Process/Traces



Estado dum Sistema

TRACE

Process-A	B	C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004	8004	12004
5005	8222	12005
5006	8223	12006
5007	….	12007
5008		….
5009		
…		

5000 endereço inicial do A
8000 endereço inicial do B
12000 endereço inicial do C

12

Um Sistema em Execução

1	5000	27	12004
2	5001	28	12005
3	5002	----------Time out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------Time out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------Time out	
16	8003	41	100
----------I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------Time out	

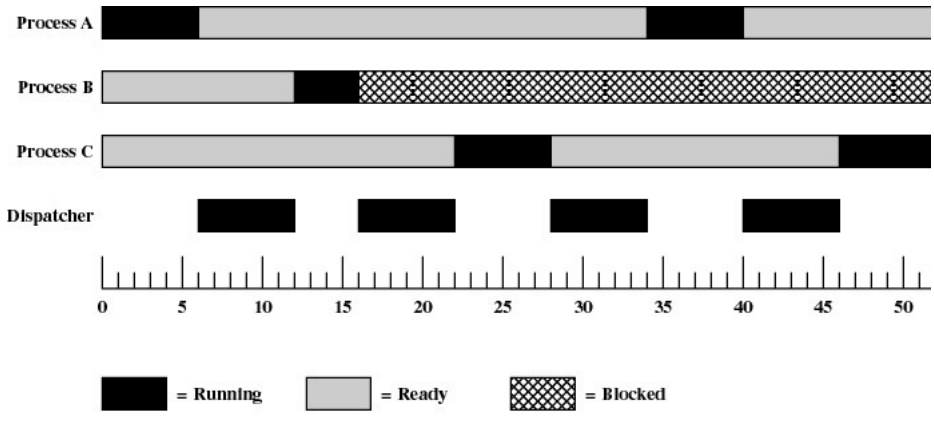
100 = Endereço Inicial do Dispatcher

105 = Enderço Final do Dispatcher

Colunas são os ciclos de instruções e o enderço da instrução a ser executada

13

Exemplo para 3 Processos



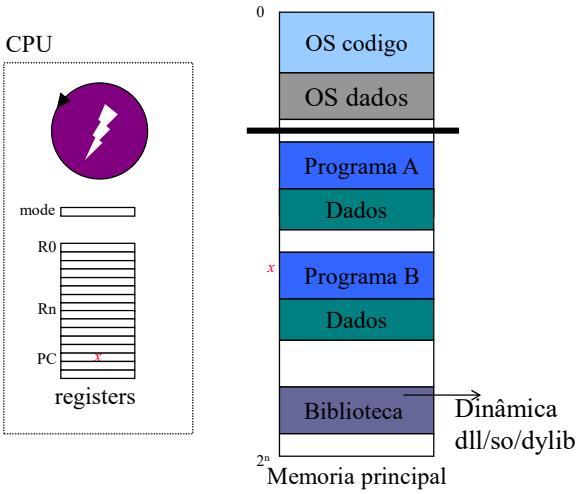
14

Um Kernel Protegido

Modo (register bit) indique se o CPU está a executar em modo *User* ou modo *Protegido*

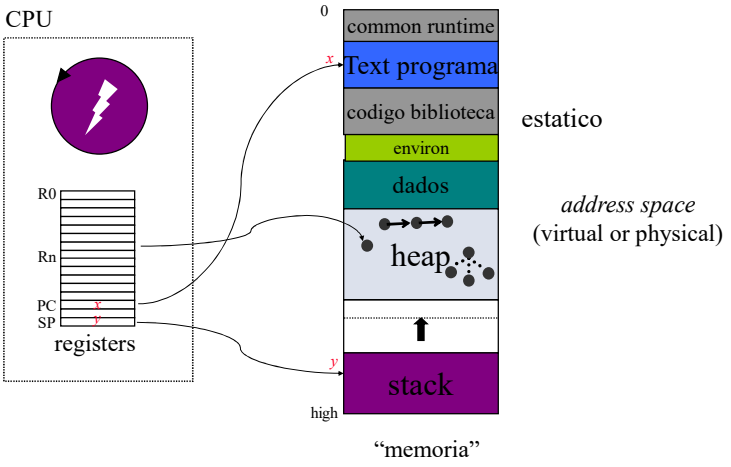
User mode /Kernel Mode

Alguns instruções e acessos a memoria são permitido apenas quando o CPU está em modo protegido



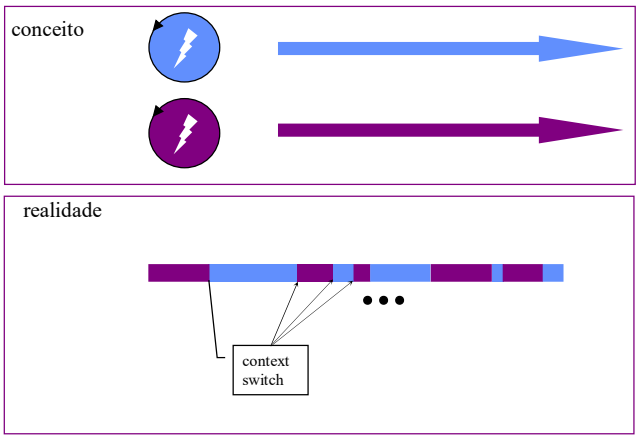
15

Um Programa em Execução



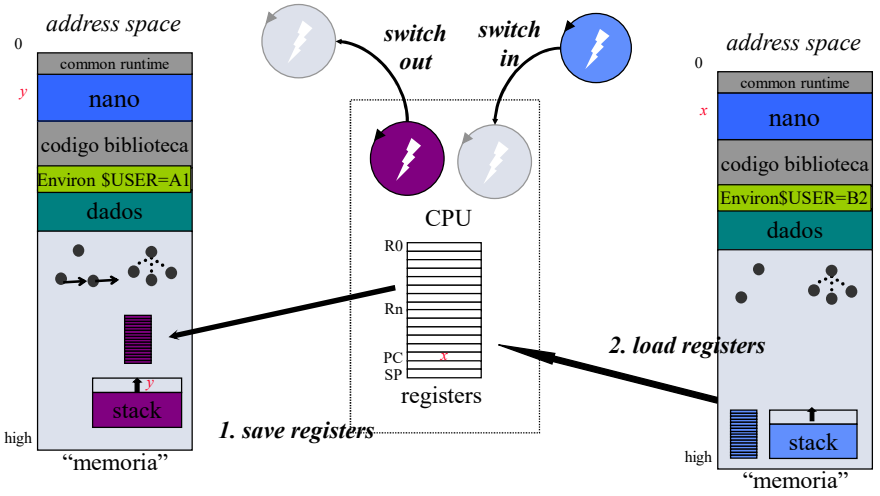
16

Dois Threads/Processos partilhando **um** CPU



17

Processo Context Switch



18

Threads

Até agora, o processo tem um única "linha" de execução

Considere ter vários contadores de programa por processo

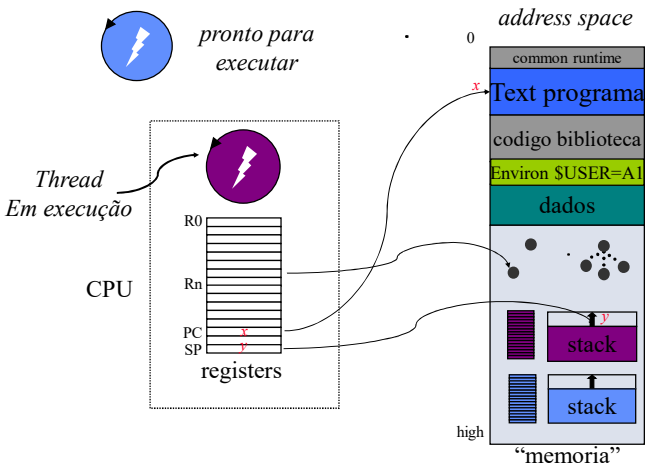
- Vários locais podem ser executados ao mesmo tempo
- Várias "linhas" de controle -> **threads**

Deve então ter armazenamento para os detalhes das threads, vários contadores de programa em PCB etc etc

Ver próximo capítulo

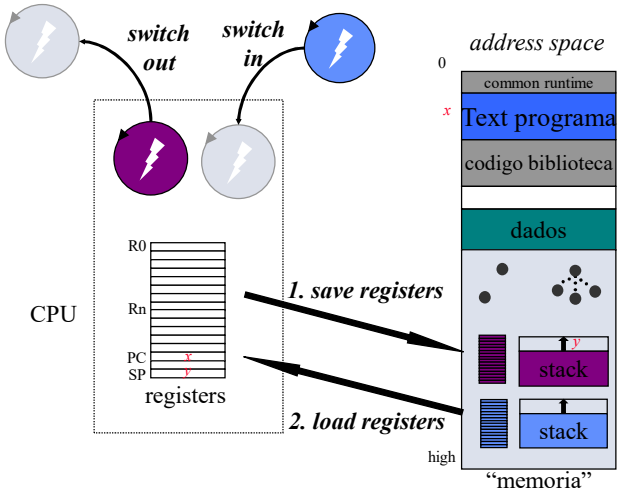
19

Um Programa com duas Threads



20

Thread Context Switch



21

Processos Sumário

- Um processo é um programa em execução
- Execução dum processo é **sequencial**
  - *Pode-se caracterizar um processo pelo seu **trace***
  - *(trace = as instruções executadas)*
- Um processo necessita recursos geridos pelo OS
- O OS execute muitos processos concorrentemente, o CPU é partilhado (multiplexed) entre eles
- O OS fornece mecanismos de manipulação e comunicação entre processos (InterProcess Communication ***IPC***)

22

Exemplo

- **Exemplo de movimentação de dados I/O**
- O programa hello

Diagramas do estado dum Processo

- Um diagrama de estado de processo é usado para caracterizar o comportamento de um processo.
- Um processo pode estar
  - **pronto (ready)**,
  - **em execução (running)**
  - **bloqueado** (**blocked**)– a espera de algo
- Como ocorre a mudança de estado do processo?
- Os estados de pronto e bloqueado podem ser refinados e também podemos introduzir estados para inicialização e terminação.

Modelo de 2 Estados

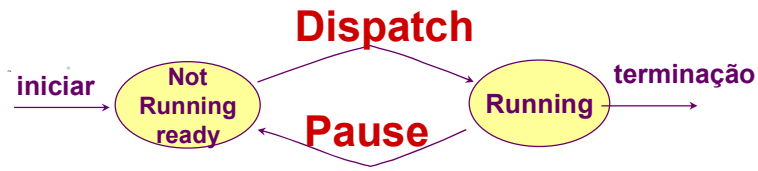


Diagrama de Transição de Estado

- Dispatcher
  - Procedimento que atribui o processador dum processo para outro
  - Previne um processo a ocupar todo o tempo do CPU

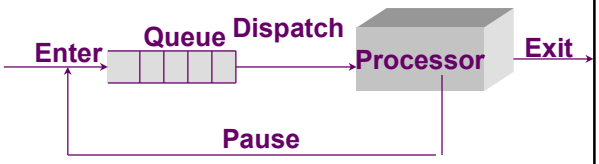


Diagrama da Fila

25

Estado dum Processo
Modelo de 5 Estados

- Um processo em execução muda de estado:
  - **new**: O processo está a ser criado.
  - **running**: Instruções estão a ser executadas.
  - **waiting**: O processo está à espera que algum evento ocorra (receção de sinal ou terminação de alguma operação de I/O).
  - **ready**: O processo está à espera que lhe seja atribuído o processador.
  - **terminated**: O processo acabou de terminar a sua execução.

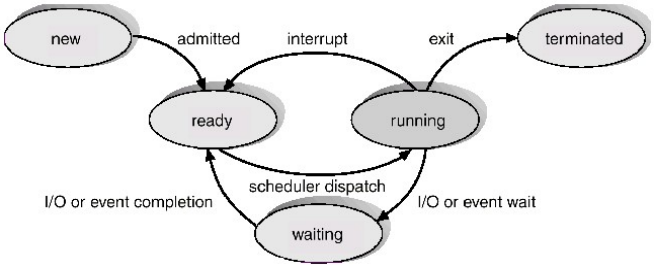


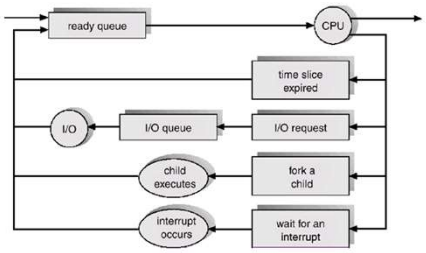
Diagrama de estados dum processo

Notas:
- **Só um processo pode estar no estado *running*.**
- **Pode haver vários processos no estado *ready* e no estado *waiting*.**

26

Diagrama de espera no escalonamento de processos

- Um novo processo é inicialmente colocado na ready queue.
- O processo fica à espera na *ready queue* até ser **selecionado** e despachado para a CPU.
- Durante a sua execução várias coisas podem acontecer:
  - o processo pode ser removido da CPU em consequência duma interrupção do timer (time quantum atinged), o que o faz transitar para a ready queue.

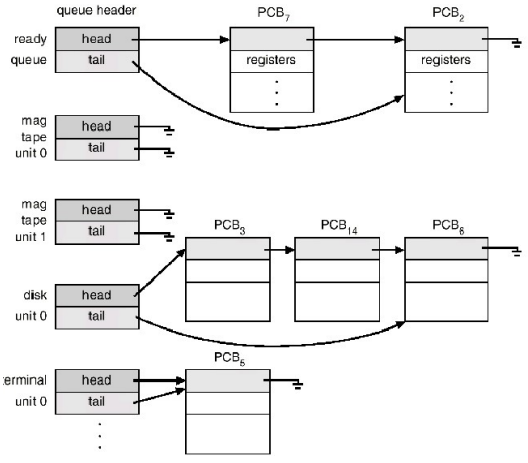


Blocked:
- o processo pode emitir um pedido I/O, e consequentemente ser colocado numa I/O device queue;
- o processo pode criar um novo subprocesso, ficando á espera que ele termine;
- o processo pode estar a espera dum ***interrupt*** qualquer – por exemplo recebeu um sinal para suspender (ctrl-z) e agora espera a sua reiniciação

27

Filas de Escalonamento de Processos

- Ready queue – Esta fila contém os PCBs dos processos residentes em memória que estão no estado ready, i.e. processos que estão prontos e à espera de executar.
- Waiting Queues
  Se um processo espera a conclusão de I/O ou outro dispositivo, é colocado na fila de espera do dispositivo em causa.
- Qualquer recurso pode ter associado uma fila de escalonamento.
- Existe Migração de processos entre várias filas.



Fila ready e filas de vários dispositivos I/O

28

Escalonadores

- Um processo migra entre várias filas de escalonamento durante o seu tempo de vida.
- O sistema operativo deve **selecionar** processos destas filas com base em algum método ou algoritmo.
- Há vários tipos de escalonadores: vamos caracterizar três tipos:
  - **curto prazo**
  - **longo prazo**
  - **médio prazo**

- **Escalonador de curto-prazo** (ou escalonador da CPU):
  - Seleciona que processos devem ser **executados de seguida** e reserva, consequentemente, a CPU.
  - Escalonador de curto-prazo é invocado com bastante frequência
    - tem de ser muito rápido !

- **Escalonador de medio-prazo** (ou escalonador de memória)
  - Aloca memoria para os processos.
  - Chamado periodicamente ou quando for necessário.

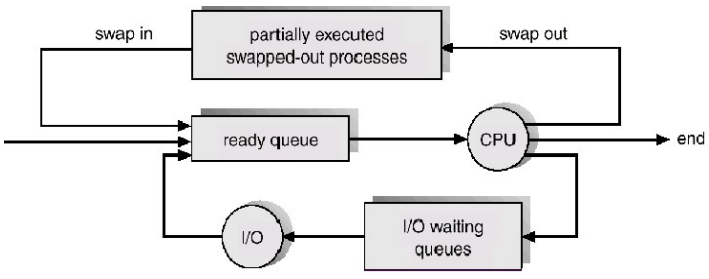
- **Escalonador de longo-prazo** (ou escalonador de processos):
  - Seleciona que processos devem ser levados para a fila ready
  - Escalonador de longo-prazo é invocado com pouca frequência
  - Escalonador de longo-prazo controla o *grau de multiprogramação*.

29

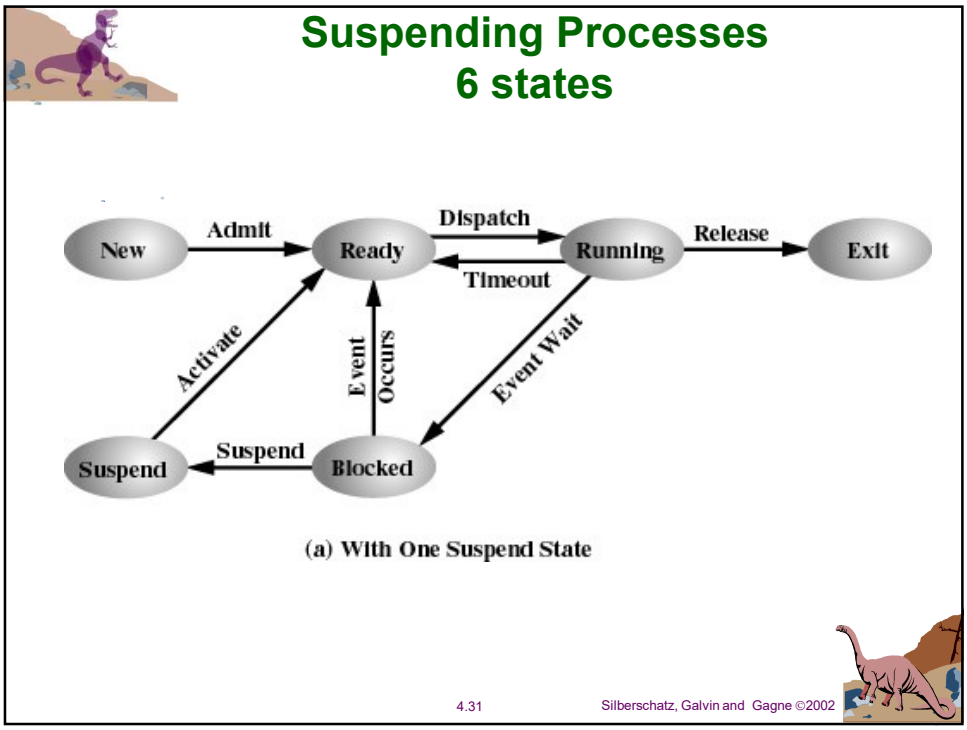
Escalonamento de médio-prazo

Ideia base: swapping

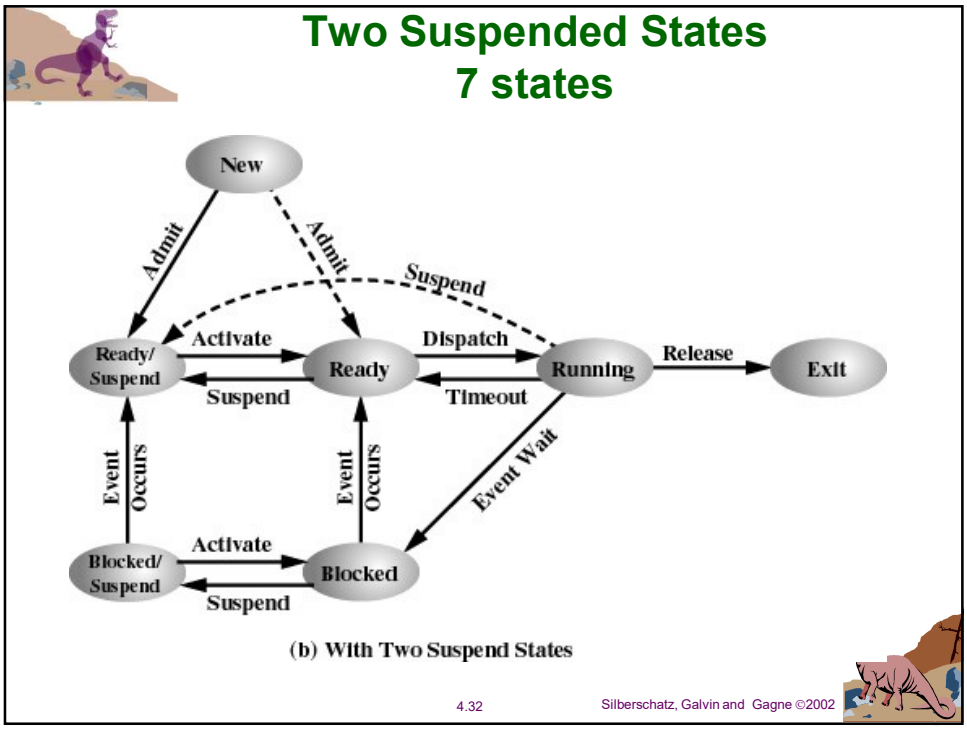
- Remover processos da memória
  - i.e. diminuir o grau de multiprogramação
- Mais tarde, estes processos podem ser reintroduzidos na memória e continuarão a sua execução do ponto onde tinham sido deixados.
- Pode assim conseguir-se uma melhor mistura de processos, ou então libertar memória principal para outros processos.



30

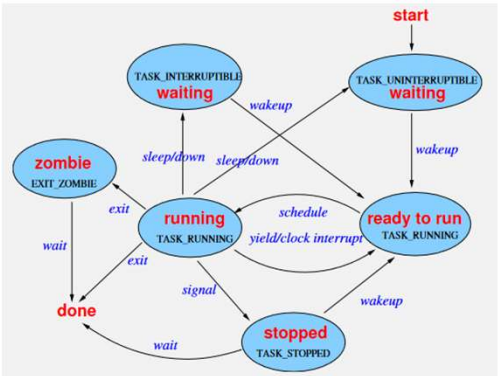

Suspending Processes
6 states
New
Admit
Ready
Dispatch
Running
Release
Exit
Timeout
Activate
Event Occurs
Event Wait
Suspend
Suspend
Blocked
(a) With One Suspend State
4.31
Silberschatz, Galvin and Gagne ©2002

31


Two Suspended States
7 states
New
Admit
Admit
Suspend
Ready/
Suspend
Activate
Ready
Dispatch
Running
Release
Exit
Suspend
Timeout
Event Occurs
Event Occurs
Event Wait
Blocked/
Suspend
Activate
Blocked
Suspend
(b) With Two Suspend States
4.32
Silberschatz, Galvin and Gagne ©2002

32

Linux

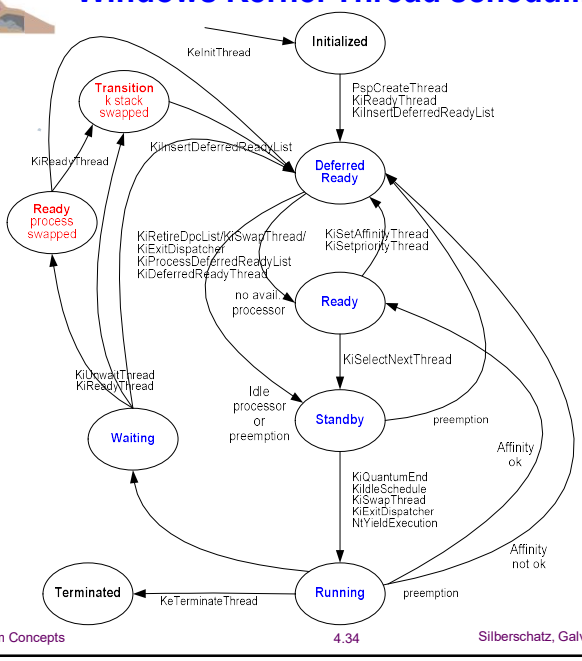


ude / linux / sched.h

```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->__state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */

/* Used in tsk->__state: */
#define TASK_RUNNING                    0x00000000
#define TASK_INTERRUPTIBLE              0x00000001
#define TASK_UNINTERRUPTIBLE            0x00000002
#define __TASK_STOPPED                  0x00000004
#define __TASK_TRACED                   0x00000008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                       0x00000010
#define EXIT_ZOMBIE                     0x00000020
#define EXIT_TRACE                      (EXIT_ZOMBIE |
/* Used in tsk->__state again: */
#define TASK_PARKED                     0x00000040
#define TASK_DEAD                       0x00000080
#define TASK_WAKEKILL                   0x00000100
#define TASK_WAKING                     0x00000200
#define TASK_NOLOAD                     0x00000400
#define TASK_NEW                        0x00000800
#define TASK_RTLOCK_WAIT                0x00001000
#define TASK_FREEZABLE                  0x00002000
#define __TASK_FREEZABLE_UNSAFE         (0x00004000 * I
#define TASK_FROZEN                     0x00008000
#define TASK_STATE_MAX                  0x00010000
```


33

Windows Kernel Thread scheduling states



Windows Server 2003
9 states

34

Informação sobre Processos

Ferrammentas Graficas (GUI's): Gestor de Tarefas / Process Explorer (windows internals) Linux - System Monitor

Ferramentas de Linha de Commando (CLI's) : o comando tasklist (windows)- e os comandos ps e top (Linux)

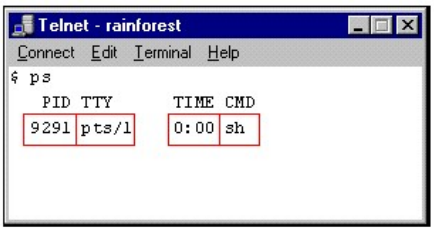
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\TOSHIBA>tasklist

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0         24 K
System                           4 Services                   0     11.744 K
smss.exe                       440 Services                   0        544 K
csrss.exe                      616 Services                   0      3.152 K
wininit.exe                    700 Services                   0        916 K
csrss.exe                      708 Console                    1     13.276 K
services.exe                   756 Services                   0      5.388 K
```

Image Name	Nome do Programa
PID	Process Identifier
Session Name	Services/Console etc
Sesiosn Name	services (0) um user (1) outro user (2) etc
Mem Usage	Utilização da Memoria

35

Informação sobre Processos

Ferramentas de Linha de Comando (CLI's) : e o comando ps (Linux)

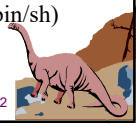


9291 O numero de identificação (PID) deste processo nesta linha

pts/1 Numero de terminal ou janela onde o processo está a correr

0:00 Quantidade de tempo de CPU usado. Neste caso quase nenhum

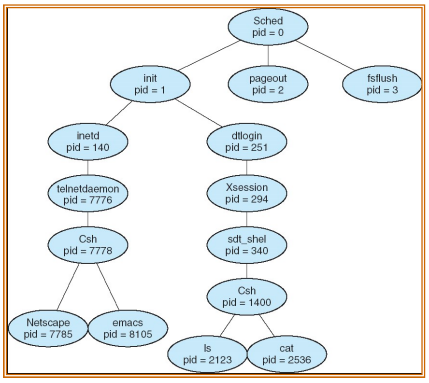
sh O nome do programa a executar, neste caso o Bourne shell (o programa /bin/sh)



36

Operações sobre Processos (criação)

- **Criação** de processos:
  O *processo progenitor* (pai) cria *processos progénitos* (filhos), os quais, por sua vez, criam outros processos, formando uma árvore de processos.
- Modos de execução:
  - Pai e filho(s) executam concorrentemente.
  - Pai espera até que o(s) filho(s) terminem.
- Ocupação da memória
  - O filho duplica o espaço do pai.
  - O filho carrega um novo programa.
- Partilha de recursos:
  - Pai e filhos partilham todos os recursos.
  - Filhos partilham um subconjunto dos recursos do pai.
  - Pai e filhos não partilham quaisquer recursos.
- Criação de processos em UNIX
  - A chamada ao sistema **fork** cria um novo processo
  - A chamada ao sistema **exec** depois dum **fork** é usada para substituir o espaço de memória do processo com um novo programa.

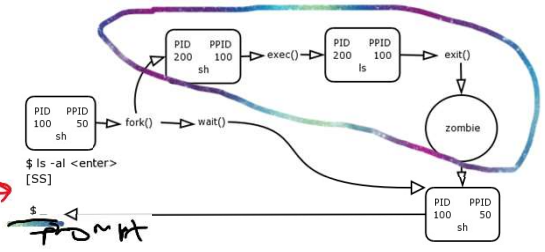


37

Exemplo no Shell

O que acontece quando "ls -al" é escrita ?

block

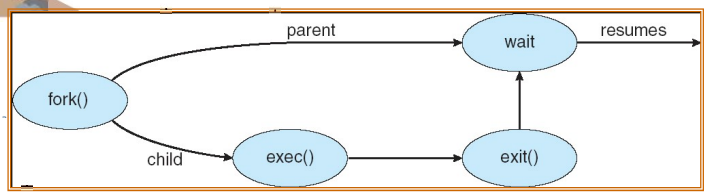


After hitting the enter key after the "ls -al" command, the shell determines that it is an **external** program on the filesystem, namely /bin/ls and not part of the shells **built-in** functionality
Therefore, the shell needs to execute "exec()" this external program.
It does this by first issuing a fork() system call followed by a call to exec() in the new process.

- The fork() system call performs a cloning operation (Copy on Write).
  - The kernel copies an existing process table entry to the next empty slot in the process table.
  - The kernel assigns a new unique PID to this forked process and updates its PPID to reflect the value of the process that forked it.
  - The forker is called the **parent** process and the forked process is called the **child** process.
- The original **parent** process then issues a wait() system call (**unless & was typed** after the command in which case it would be executed in background)
- The child process then issues the exec() to run the new program which cleans out the current program text and loads the new one and resets/clears the stack

38

Processos – Criação - API




```
int  newprocess ()  //outline of shell code
{
    int  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
     }
     else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process  will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
}
```


39

Windows API

```
#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\\mspaint.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);
}
```

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

Creates a new process and its primary thread.
The new process runs in the security context of the calling process.

40

Controlo de Processos - Signals

- Controlo Enviando um Sinal ao Processo pelo Sistema Operativo
  - Ctrl-c kill active process
  - Ctrl-z etc suspend active process
  - Ctrl-alt-delete System Interrupt
- Ferramentas Gráficas - Uso Comun
  - Windows - task manager/process explorer
  - Linux - System Monitor
- Ferramentas de Linha de Comando (para scripts, administradores, programas batch etc)
  - Enviar Custom Signal to PID
    - Windows - taskkill
    - Linux – kill
  - Visualizar jobs, processos e threads
    - jobs, ps, top etc

41

Controlo de Processos

bash

- ps
- Control
  - Ctrl c (envie sinal processo atual)
  - Ctrl z (envie sinal processo atual)
- kill
  - kill -9 pid (ctrl c)
  - kill –code pid
  - kill –l list codes
- top
  - List processes by cpu use

42

Controlo de Processos

Foreground and Background Processes

Logo a seguir um "login", está-se a interagir com o seu shell de login.

Diz-se que o shell está a correr ou executar no "*foreground" (primeiro plano). porquê está no nosso frente e podemos interagir com ele*.

Quando se execute um novo Shell dentro dum primeiro o shell original é posto em "*background" (segundo plano ou plano de fundo) e o novo shell/processos é posto no foreground*.

Quando um processo está em "background" quer dizer que está pronto (waiting) para executar ou até mesmo executando na memória do computador mas não é possível interagir com este processo vai "stdin" apenas por sinais e comandos especiais.

No caso dum shell implica que está num estado de espera, Quando se sai do shell novo o shell original estará obviamente trazido para o "foreground"

jobs

- No Linux, um **job** (tarefa) é um processo (que pode dar origem a outros) que o shell está a gerir.
- & : executar processo ou trabalho em segundo plano
- Control-Z : interrompe o trabalho em primeiro plano e coloca-o em segundo plano como um trabalho interrompido
- jobs : listar tarefas
- kill %n : terminatar job numero n
- fg / bg
  - Mova a tarefa atual para primeiro plano (fg) ou segundo plano (bg) ou especifique o número da tarefa com %n
- at, batch, atq, atrm
  - queue, examine or delete jobs for later execution

Operações sobre Processos (terminação)

Modos de terminação:

- **Terminação normal**. Um processo termina quando acaba a execução da sua última instrução, e pede ao sistema operativo para eliminá-lo via a função **exit** do standard library, a chamada ao sistema **_exit**. Nesta altura:
  - O processo devolve (**return**) eventualmente dados ao seu progenitor (via a chamada ao sistema **wait**).
  - Recursos do processo progénito são libertados pelo sistema operativo.
- **Terminação abrupta**. O processo progenitor pode terminar a execução dos processos filhos através duma chamada ao sistema - **abort**.
  - Filho excedeu os recursos que lhe foram reservados.
  - A tarefa atribuída ao filho não é mais necessária.
  - O pai está a terminar, o que obriga os seus filhos a terminar.

45

IPC

- Comunicação entre Processos (IPC)

46

Comunicação entre Processos (IPC)

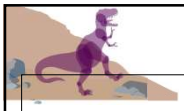
Na cooperação entre processos, estes podem ser classificados como:

- **Processos independentes**. Um processo *independente* não pode afetar nem ser afetado pela execução doutro processo.
- **Processos cooperantes**. Um processo *cooperante* pode afetar ou ser afetado pela execução dum outro processo.
  - Vantagens da cooperação entre processos:
    - **Partilha de informação** (p.ex. ficheiros, variáveis, trincos)
    - **Aceleração da computação** (vários processadores/cores)
    - **Modularidade** (dividir funcionalidades em processos separados)
    - **Utilidade** do ponto de vista do utilizador (p.ex. edição, compilação e impressão em paralelo)
  - Desvantagens : mais complexo, sincronização

"Cooperating Processes" precisam de **interprocess communication** (**IPC**)

- Existem dois modelos de IPC
  - **Message passing**
  - **Shared memory**

47



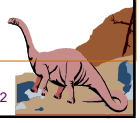
Interprocess Communication

Message Passing

- Mecanismo para os processos comunicarem e sincronizarem suas ações
- Use um sistema de mensagens – os processos se comunicam entre si sem recorrer a variáveis compartilhadas
- IPC oferece duas operações de alto nível:
  - **`send`** (*message*) and **`receive`** (*message*)
- O tamanho da mensagem poderá ser fixo ou variável

Shared Memory

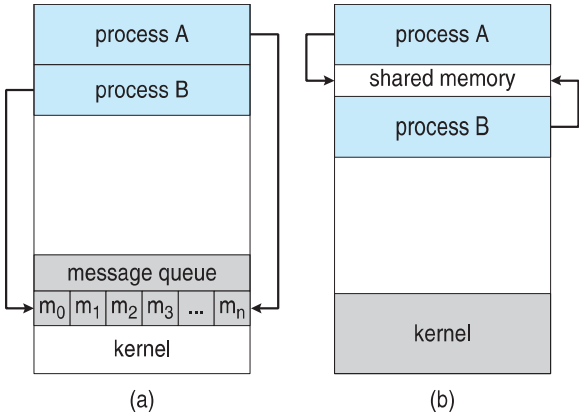
- Área de memória partilhada entre os processos que desejam comunicar
- O protocolo de comunicação é determinado pelos processos do usuário e não pelo sistema operacional.
- A principal questão é fornecer e usar mecanismos que permitam que os processos do usuário sincronizem suas ações no acesso à memória compartilhada.



48

Communications Models

(a) Message passing. (b) shared memory.



Instead of the kernel we can use a third party user library that brokers the exchange (p.ex MPI) on a medium such as the Internet

49

Exemplo de "Mesage Passing" Library: MPI Rotinas com bloqueio

Envio do valor dum inteiro *x* do processo 0 para o processo 1

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  //WHO AM I

if (0==rank) {  //I am process 0
  int x;
  MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
}
else  if (1==rank) { //I am process 1
  int xy;
  MPI_Recv(&xy, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, &status);
}
```


50

Modos de Envio

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
      - Can possibly register a callback
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**
- Example: The MPI library has implementations for blocking and non-blocking sends and recieves although the semantics may be different to the above

Producer-Consumer Problem

- Paradigma para “cooperating processes” processo “*producer*” produz informação que é consumido pelo processo de “consumer”
  - **unbounded-buffer** no limit on the size of the buffer
  - **bounded-buffer** fixed buffer size

Bounded-Buffer – Shared-Memory Solution

Shared Data Structures

```
#define BUFFER_SIZE 10

typedef struct {          . . . } item;

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

Bounded-Buffer – Producer

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

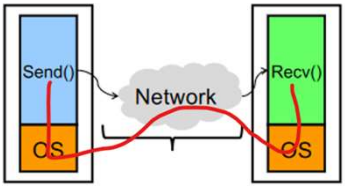
Bounded Buffer – Consumer

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

IPC Methods

Basic Operating System Supported Mechanimss

- Shared Memory Segments (Not Synchronized)
- Pipes and Named Pipes (Not Synchronized)
  - Not that 2 processes reading from same pipe end or two writing to the same pipe end → results in .. chaos
  - One reading and one writing is OK.
  - Best for unidirectional communication
- Message Queues (Synchronized)



Other Methods: Via OS or Libraries

- Network : Low level API -Sockets (Can be Synched & Not Synched)
- Remote Procedure Calls (RPC)
- Remote Method Invocation (Java RMI)

55

IPC Pipes

- A Pipe acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication **uni**directional or **bi**directional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes ?
  - Can the pipes be used over a network ?
  - What about synchronization ?

56

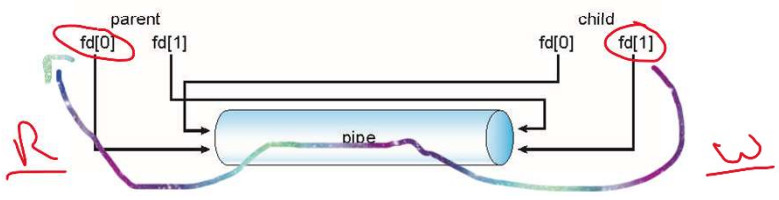
IPC Pipes

- A Pipe acts as a conduit allowing two processes to communicate
- Ordinary pipes
  - Cannot be accessed from outside the process that created it.
  - Typically, a parent process creates one or more half duplex pipes and uses them to communicate with a child process that it created.
  - Half duplex.
- Named pipes
  - Can be accessed without a parent-child relationship.
  - Also half duplex.
  - More powerful. More dangerous ☺
- Sockets
  - full duplex network pipes

57

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
  - Producer writes to one end (the write-end of the pipe)
  - Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore **unidirectional**
- Require **parent-child (fork) relationship** between processes



- Nota : EM Windows são chamados “anonymous pipes”

58

Named Pipes

Named Pipes are **more powerful** than ordinary pipes

- No parent-child relationship is necessary between the communicating processes
- Several processes can use the **named** pipe for communication
- Provided on both Linux and Windows systems
- The Named pipe is a FILE (LINUX) or OBJECT (Windows)
  - i.e we can open/read the file/object

59

FIM

- FIM

60