Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus

Introdução

Uma diversidade de plataformas…

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- Telemóveis (e PDAs),

- Tablets e laptops,

- Servidores e mainframes

... para satisfazer diferentes necessidades

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- Utilidades…
  - Envio/recepção de mensagens, chamadas
  - Agendas e gestão de tempo
- Entretenimento
  - Jogos, fotografia, filmes…
- Ferramentas de Produtividade Pessoal
  - Tratamento de texto, folha de cálculo, “PowerPoint”, …
- Serviços bancários
  - Acesso on-line a contas, movimentos, pagamentos, …
- E muito… muito… mais!

... para as quais existem aplicações...

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- Aplicações muito específicas…
  - Envio/recepção de mensagens, chamadas
- Aplicações multimedia
  - Jogos, fotografia, filmes…
- Ferramentas de Produtividade Pessoal
  - "Office" (Word, Excel, PowerPoint,…)
  - Agendas e gestão de tempo
- Aplicações bancárias
  - Acesso on-line a contas, movimentos, pagamentos, …
- E muitas… muitas… mais!

Algumas questões

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- O que são…
  - Aplicações?
  - Programas e Aplicações são a mesma coisa?!
- Sistema de Operação
  - O que é? Para que serve?
  - É uma aplicação? Se não é, qual é/quais são/ as diferenças?
  - Porque há (?tantos?) SOs diferentes?
- Estas e outras questões serão, esperamos, oportunamente respondidas ao longo destas aulas… ☺

Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

Programas:

Do código fonte ao executável…

(recordando AC e ISRC; algumas -poucas- "coisas" novas; complementadas na aula prática)

Programa fonte

rotina.c

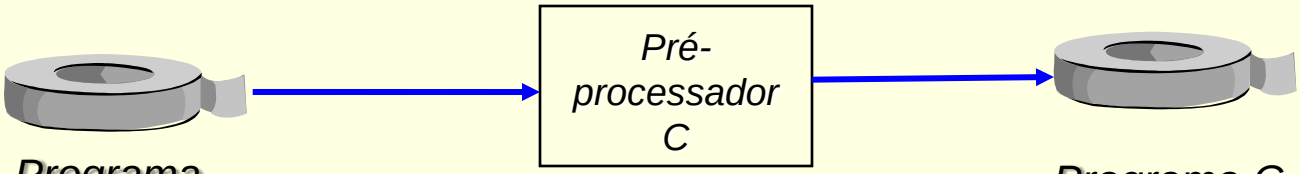
```
#define Pi  3.1415
void abc(void)
{
   float x;
   x= Pi;
}
```

prog.c

```
#define SizeOfInt  4
int main()
{
   int i;
   int *p;
   i= 1;
   p= (int *)malloc(SizeOfInt);
   *p= 2;
   abc();
   printf("Vou terminar\n");
   exit(0);
}
```

O que é um Programa? (visão simplificada)

- 1 a) Produção do Programa fonte “C puro”



Programa C “usual”

*(inclui linguagem **cpp**, i.e., directivas começadas por #)*

<ficheiro>.c

```
#define Pi  3.1415

void abc(void)

{
   float x;
   x= Pi;
}
```

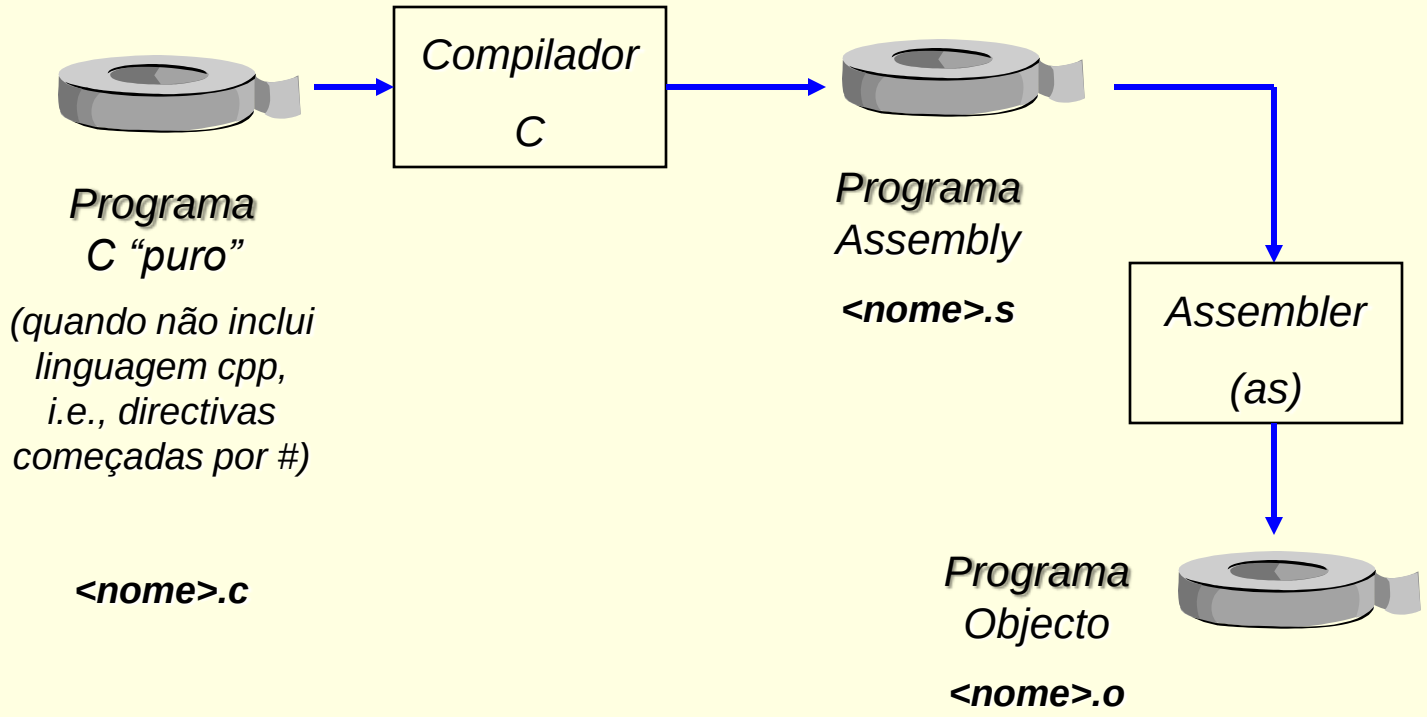
Programa C “puro”

<temporário>.c

```
void abc(void)

{
   float x;
   x= 3.1415;
}
```

O que é um Programa? (visão simplificada)

- 1 b) Produção do Programa Objecto



Programa assembly (1)

- Código da função main

```
        .section .rodata
.LC1:
        .string  "Vou terminar"

        .text
.globl main
        .type    main, @function
main:
        leal     4(%esp), %ecx
        andl     $-16, %esp
        pushl    -4(%ecx)
        pushl    %ebp
        movl     %esp, %ebp
        pushl    %ecx

        subl     $20, %esp                  int i; int *p; …

        movl     $1, -8(%ebp)               i= 1
```

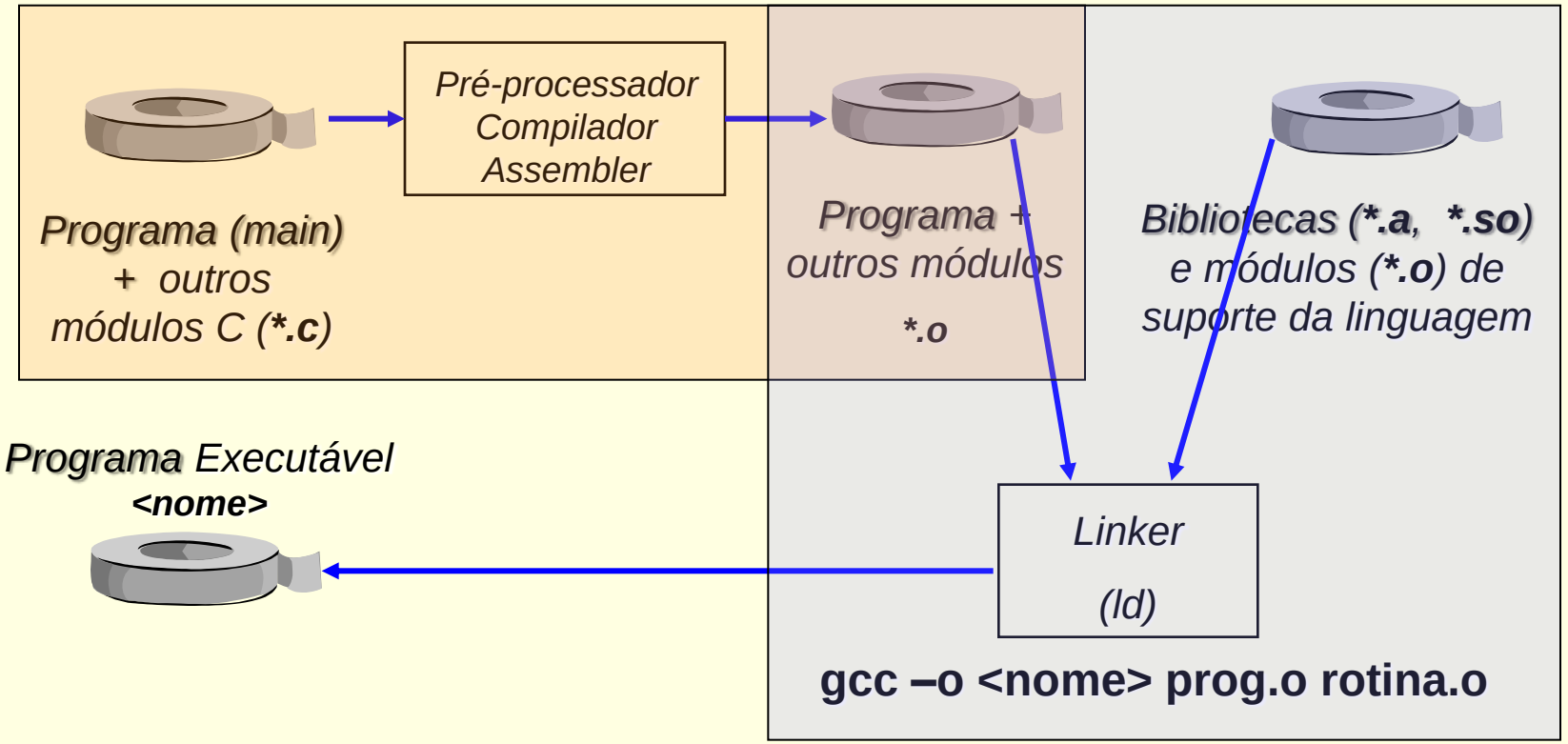
Programa assembly (2)

- Código da função main (continuação)

```
movl    $4, (%esp)
call    malloc                  malloc(4)

movl    %eax, -12(%ebp)         p= malloc(…)

movl    $2, (%eax)              *p= 2

call    abc                     abc()

movl    $.LC1, (%esp)
call    puts                    o compilador decidiu chamar
                                puts em vez de printf

movl    $0, (%esp)
call    exit                    exit(0)
```

O que é um Programa? *(visão simplificada)*

- 2 - Produção do Programa Executável "em vários passos":
  gcc –c prog.c; gcc –c rotina.c; gcc –o prgx prog.o rotina.o



O que é um Programa? (ficheiro executável)

`objdump -D prgx` [Nota: não corresponde ao código C apresentado anteriormente]

```
...
08048374 <main>:
 8048374: 55                                push   %ebp
 8048375: 89 e5                             mov    %esp,%ebp
 8048377: 83 ec 08                          sub    $0x8,%esp
 804837a: 83 e4 f0                          and    $0xfffffff0,%esp
 804837d: b8 00 00 00 00                    mov    $0x0,%eax
 8048382: 83 c0 0f                          add    $0xf,%eax
 8048385: 83 c0 0f                          add    $0xf,%eax
 8048388: c1 e8 04                          shr    $0x4,%eax
 804838b: c1 e0 04                          shl    $0x4,%eax
 804838e: 29 c4                             sub    %eax,%esp
 8048390: c7 45 fc 01 00 00 00              movl   $0x1,0xfffffffc(%ebp)
 8048397: c7 05 dc 95 04 08 02              movl   $0x2,0x80495dc
 804839e: 00 00 00
 80483a1: 8b 45 f8                          mov    0xfffffff8(%ebp),%eax
 80483a4: c7 00 03 00 00 00                 movl   $0x3,(%eax)
 80483aa: c9                                leave
 80483ab: c3                                ret
```

A variável i é guardada no endereço ebp-4, que é um endereço no stack

O código executável do programa é para ser carregado em memória a partir da posição 0x8048374

A variável x é guardada no endereço 0x80495dc

Execução de um Programa (simulação)

(1) IP contém endereço de arranque do programa.

(2) O endereço é "emitido" para a memória (no address bus) juntamente com a indicação de que se trata de um READ.

(3) A memória entrega os bytes.

(4) A unidade de controlo descodifica a instrução; se é válida, despacha-a para ser executada.

(5) O IP é incrementado; fica a "apontar" (contém o endereço) da instrução seguinte.

(6) Todo o processo é repetido…


```
…

pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
andl    $-16, %esp
movl    $0, %eax
addl    $15, %eax
addl    $15, %eax
shrl    $4, %eax
sall    $4, %eax
subl    %eax, %esp
movl    $1, -4(%ebp)
movl    $2, x
movl    -8(%ebp), %eax
movl    $3, (%eax)
leave
ret
.size   main, .-main
.comm   x,4,4
```

Execução de um Programa (simulação)

Execução de uma instrução que acede a memória:

`movl $1, -4(%ebp)`

(para a dupla-word (32 bits) que fica localizada quatro posições abaixo da posição apontada pelo ebp é transferido o valor 1). Como?

(1) O endereço efectivo é calculado: é o valor de ebp subtraído de 4.

(2) O endereço efectivo é colocado no address bus.

(3) O valor 1 é colocado no data bus.

(4) É dada uma ordem de WRITE para a memória.

```
                …
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $1, -4(%ebp)
        movl    $2, x
        movl    -8(%ebp), %eax
        movl    $3, (%eax)
        leave
        ret
        .size   main, .-main
        .comm   x,4,4
```

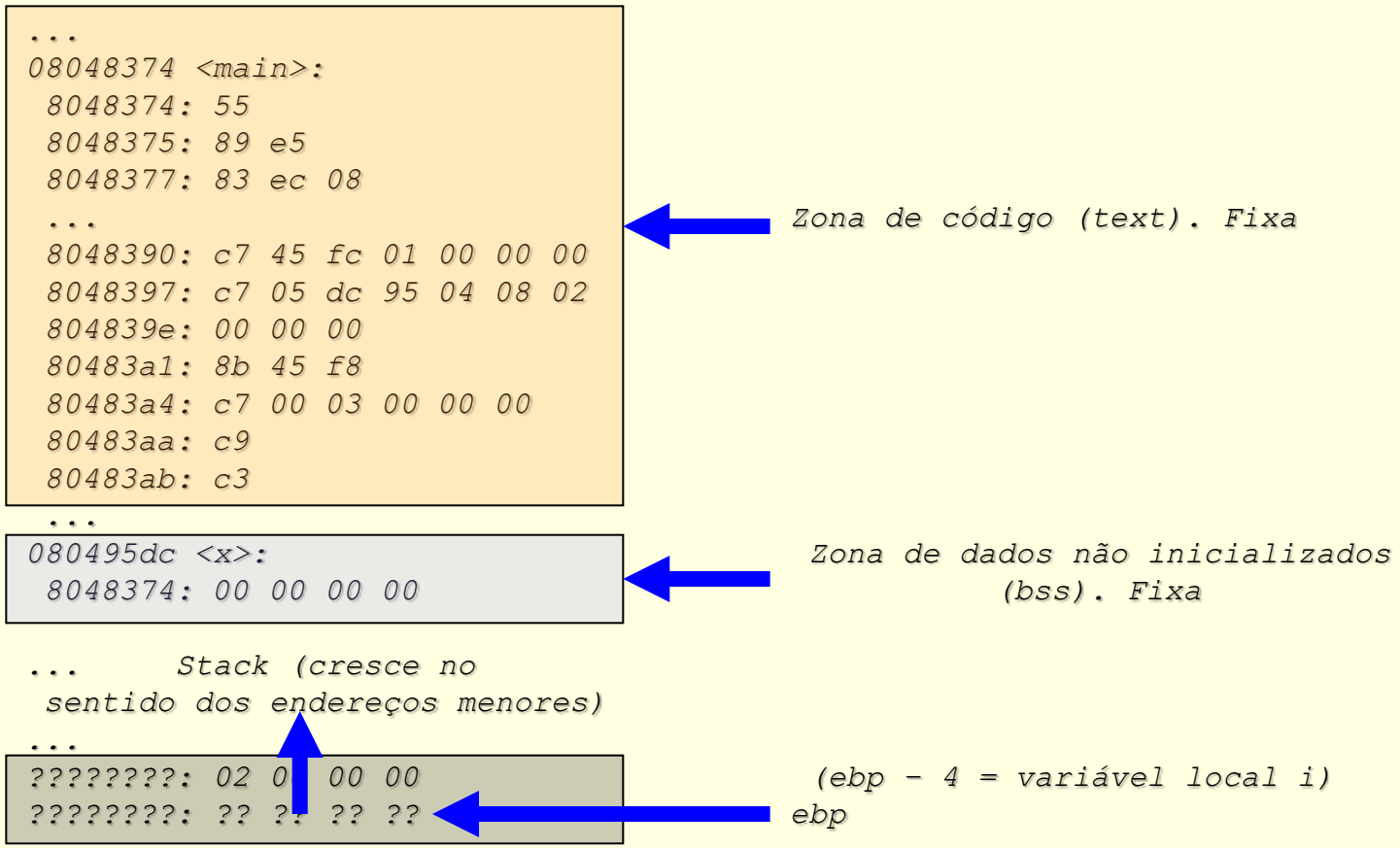
Execução de um Programa (simulação)


```
…
08048374 <main>:
 8048374: 55                            push   %ebp
 8048375: 89 e5                         mov    %esp,%ebp
 8048377: 83 ec 08                      sub    $0x8,%esp
 804837a: 83 e4 f0                      and    $0xfffffff0,%esp
 804837d: b8 00 00 00 00                mov    $0x0,%eax
 8048382: 83 c0 0f                      add    $0xf,%eax
 8048385: 83 c0 0f                      add    $0xf,%eax
 8048388: c1 e8 04                      shr    $0x4,%eax
 804838b: c1 e0 04                      shl    $0x4,%eax
 804838e: 29 c4                         sub    %eax,%esp
 8048390: c7 45 fc 01 00 00 00          movl   $0x1,0xfffffffc(%ebp)
 8048397: c7 05 dc 95 04 08 02          movl   $0x2,0x80495dc
 804839e: 00 00 00
 80483a1: 8b 45 f8                      mov    0xfffffff8(%ebp),%eax
 80483a4: c7 00 03 00 00 00             movl   $0x3,(%eax)
 80483aa: c9                            leave
 80483ab: c3                            ret
```

O IP "percorre" estes endereços (i.e., injecta-os no bus de endereços) para que o CPU "receba" as instruções…

Injectando este valor no bus de endereços acede-se à variável x

Na memória: as várias Zonas do Programa
(deduzidas a partir do programa executável)



TPC: Perguntas sobre o programa *(do slide 15)*

- *Como é que se vê que o programa,*
  - *usa os registos de 32 bits do CPU?*
  - *usa operações que manipulam 32 bits de cada vez quando acedem à memória?*
  - *usa endereços de 32 bits?*
  - *é para correr numa arquitectura "little-endian"?*

TPC: Perguntas sobre a execução do programa

- *Como é que se sabe quantos bytes têm de ser "trazidos" da memória, durante a fase de fetch para "trazer" a próxima instrução para "dentro" do CPU? (slide 15)*
- *Como é que se processa a "incrementação do IP" citada no slide 13 (passo 5)? O IP incrementa de 1? De 2? De outro valor?*
- *No slide 13, entre os passos (2) e (3), quanto tempo decorre? Isto é, quanto tempo leva a memória a responder a um pedido de READ?*

Conclusão…

- *O que sabemos (recordamos):*
  - *Um processador executa programas em linguagem-máquina*
  - *Um programa, para ser executado tem de estar carregado em memória (RAM)*

  e ainda que
  - *Os programas são inicialmente armazenados em dispositivos de memória persistente – discos, memórias flash – sob a forma de ficheiros*

- *O que ☺ intuitivamente ☺ sabemos:*
  - *A gestão dos ficheiros em disco é tarefa do SO*
  - *Para executar um programa interagimos com o SO (click, enter, comando…)*
  - *Para carregar um programa em memória é preciso que haja espaço livre…*