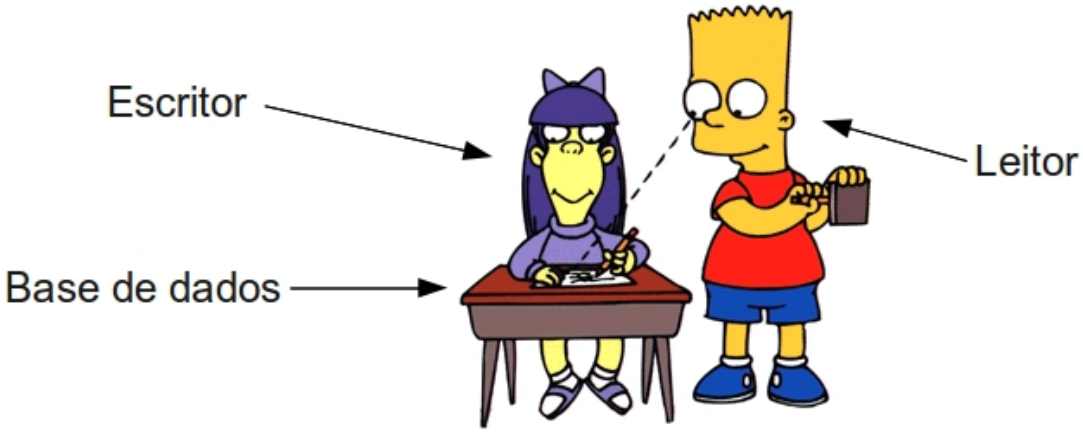
Aula 12 – Problemas Clássicos de Comunicação entre Processos Multiprogramação e Memória

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

17 de outubro de 2014

Leitores e Escritores (Courtois, 1971)

- Modela acessos a uma base de dados
- Um sistema com uma base de dados é acessado simultaneamente por diversas entidades.
- Estas entidades realizam dois tipos de operações:
  - Leitura
  - Escrita



Leitores e Escritores

- Neste sistema é aceitável a existência de diversas entidades lendo a base de dados ao mesmo tempo.

- Porém, se um processo necessita escrever na base, nenhuma outra entidade pode realizar acesso a ela
  - Nem mesmo leitura

Leitores e Escritores

- O que fazer?

Leitores e Escritores

- O que fazer?
- Escritores devem bloquear a base de dados
- Leitores:
  - Se a base estiver desbloqueada:
    - Se for o primeiro leitor a usar a base, deve bloqueá-la, para que nenhum escritor entre
    - Se, contudo, já houver outro leitor lá, basta usar a base
  - Ao sair, os leitores verificam se há ainda outro usando a base
    - Se não houver, desbloqueia a base
    - Se houver, deixa bloqueada

Leitores e Escritores

- Possível solução:

```
typedef int semaphore;                  /* use sua imaginação */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* número de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base();             /* acesso aos dados */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc - 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o último leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read();              /* região não crítica */
     }
}
void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data();              /* região não crítica */
          down(&db);                    /* obtém acesso exclusivo */
          write_data_base();            /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Leitores e Escritores

- Há algum problema com esse procedimento?

Leitores e Escritores

- Há algum problema com esse procedimento?
  - Ele esconde uma decisão tomada por nós
  - Suponha que um leitor acesse a base
  - Enquanto isso, outro leitor aparece, e entra sem problemas.
  - Mais leitores aparecem, entrando na base
  - Agora suponha que um escritor aparece
    - Não poderá entrar, devido aos leitores. Então ele é suspenso
  - Mas não param de chegar leitores

Leitores e Escritores

- E como resolvemos?

Leitores e Escritores

- E como resolvemos?
  - Podemos fazer com que, quando um leitor chegar e um escritor estiver esperando, o leitor é suspenso também, em vez de ser admitido imediatamente
    - Escritores precisam apenas esperar que leitores ativos completem
    - Não precisam esperar por leitores que chegam depois dele
  - Desvantagem:
    - Há menos concorrência $\rightarrow$ menor desempenho

Barbeiro Sonolento

- Uma barbearia possui:
  - 1 barbeiro
  - 1 cadeira de barbeiro
  - N cadeira para clientes esperarem
- Quando não há clientes, o barbeiro senta em sua cadeira e dorme

Barbeiro Sonolento

- Quando um cliente chega, ele acorda o barbeiro
- Quando um cliente chega e o barbeiro estiver atendendo um cliente, ele aguarda sua vez sentado na cadeira de espera
- Quando um cliente chega e não existem cadeiras de espera disponíveis, o cliente vai embora
- O problema é programar o barbeiro e os clientes sem que haja condição de disputa

Barbeiro Sonolento

```
#define BANCOS 5      /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;  /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;     /*para controle da região crítica*/
int esperando=0;      /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);       /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;      /*incrementa o contador de clientes à espera*/
    up(&clientes);    /*acorda o barbeiro se necessário*/
    up(&mutex);       /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();   /*sentado e sendo servido*/
  } else {
    up(&mutex);       /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento

Conta-se os avisos de que clientes estão à espera de atendimento

```
#define BANCOS 5      /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;  /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;     /*para controle da região crítica*/
int esperando=0;      /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);       /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;      /*incrementa o contador de clientes à espera*/
    up(&clientes);    /*acorda o barbeiro se necessário*/
    up(&mutex);       /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();   /*sentado e sendo servido*/
  } else {
    up(&mutex);       /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento

Conta-se os avisos de que clientes estão à espera de atendimento

Ao chegar para trabalhar, se não houver clientes, o barbeiro dorme

```
#define BANCOS 5      /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;  /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;     /*para controle da região crítica*/
int esperando=0;      /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);       /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;      /*incrementa o contador de clientes à espera*/
    up(&clientes);    /*acorda o barbeiro se necessário*/
    up(&mutex);       /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();   /*sentado e sendo servido*/
  } else {
    up(&mutex);       /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento

Conta-se os avisos de que clientes estão à espera de atendimento

Ao chegar para trabalhar, se não houver clientes, o barbeiro dorme

Um cliente que entra na barbearia deve contar o número de clientes à espera de atendimento. Se este for menor que o número de cadeiras, ele ficará; do contrário, sairá

```
#define BANCOS 5       /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;   /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;      /*para controle da região crítica*/
int esperando=0;       /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);        /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;       /*incrementa o contador de clientes à espera*/
    up(&clientes);     /*acorda o barbeiro se necessário*/
    up(&mutex);        /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();    /*sentado e sendo servido*/
  } else {
    up(&mutex);        /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento

Conta-se os avisos de que clientes estão à espera de atendimento

Ao chegar para trabalhar, se não houver clientes, o barbeiro dorme

Um cliente que entra na barbearia deve contar o número de clientes à espera de atendimento. Se este for menor que o número de cadeiras, ele ficará; do contrário, sairá

Qualquer outro cliente (e até o barbeiro) deve esperar a liberação de mutex (proteje até o semáforo com o mutex)

```
#define BANCOS 5       /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;   /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;      /*para controle da região crítica*/
int esperando=0;       /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);        /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;       /*incrementa o contador de clientes à espera*/
    up(&clientes);     /*acorda o barbeiro se necessário*/
    up(&mutex);        /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();    /*sentado e sendo servido*/
  } else {
    up(&mutex);        /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento

clientes e *esperando* são duas variáveis distintas porque entre ler a variável e testar, ela pode ser mudada

Dessa forma protege-se a variável de teste com um mutex.

```
#define BANCOS 5      /*número de cadeiras para os clientes à espera*/
typedef int semaforo;
semaforo clientes=0;  /*número de clientes à espera de atendimento*/
semaforo barbeiros=0; /*número de barbeiros à espera de clientes*/
semaforo mutex=1;     /*para controle da região crítica*/
int esperando=0;      /*clientes estão esperando (não estão cortando)*/

void barbeiro(void) {
  while (TRUE) {
    down(&clientes); /*vai dormir se o número de clientes for 0*/
    down(&mutex);    /*obtém acesso a 'esperando'*/
    esperando--;     /*decresce o contador de clientes à espera*/
    up(&barbeiros);  /*um barbeiro está pronto para cortar o cabelo*/
    up(&mutex);      /*libera 'esperando'*/
    corta_cabelo();  /*corta o cabelo (fora da região crítica)*/
  }
}

void cliente (void) {
  down(&mutex);       /*entra na região crítica*/
  if (esperando < BANCOS) { /*se não houver cadeiras livres, saia*/
    esperando++;      /*incrementa o contador de clientes à espera*/
    up(&clientes);    /*acorda o barbeiro se necessário*/
    up(&mutex);       /*libera o acesso a 'esperando'*/
    down(&barbeiros); /*vai dormir se o número de barbeiros livres for 0*/
    recebe_corte();   /*sentado e sendo servido*/
  } else {
    up(&mutex);       /*a barbearia está cheia; não espere*/
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente$_1$	exec	0	0	0	0
cliente$_1$	exec	0	0	0	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente$_1$	exec	0	0	0	0
cliente$_1$	exec	0	0	0	1
cliente$_1$	exec	1	0	0	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente$_1$	exec	0	0	0	0
cliente$_1$	exec	0	0	0	1
cliente$_1$	exec	1	0	0	1
cliente$_1$	exec	1	0	1	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0
cliente_2	exec	0	0	0	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0
cliente_2	exec	0	0	0	1
cliente_2	exec	1	0	0	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0
cliente_2	exec	0	0	0	1
cliente_2	exec	1	0	0	1
cliente_2	exec	1	0	1	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Barbeiro Sonolento – Exemplo

processo	*ação*	*clientes*	*barbeiros*	*mutex*	*esperando*
–	–	0	0	1	0
barbeiro	bloq	0	0	1	0
cliente_1	exec	0	0	0	0
cliente_1	exec	0	0	0	1
cliente_1	exec	1	0	0	1
cliente_1	exec	1	0	1	1
barbeiro	exec	0	0	1	1
barbeiro	exec	0	0	0	1
barbeiro	exec	0	0	0	0
barbeiro	exec	0	1	0	0
barbeiro	exec	0	1	1	0
cliente_1	exec	0	0	1	0
cliente_2	exec	0	0	0	0
cliente_2	exec	0	0	0	1
cliente_2	exec	1	0	0	1
cliente_2	exec	1	0	1	1
cliente_2	bloq	1	0	1	1

...

```
#define BANCOS 5
typedef int semaforo;
semaforo clientes=0;
semaforo barbeiros=0;
semaforo mutex=1;
int esperando=0;

void barbeiro(void) {
  while (TRUE) {
    down(&clientes);
    down(&mutex);
    esperando--;
    up(&barbeiros);
    up(&mutex);
    corta_cabelo();
  }
}

void cliente (void) {
  down(&mutex);
  if (esperando < BANCOS) {
    esperando++;
    up(&clientes);
    up(&mutex);
    down(&barbeiros);
    recebe_corte();
  } else {
    up(&mutex);
  }
}
```

Problemas Clássicos – Links Adicionais

- http://www.anylogic.pl/fileadmin/Modele/Traffic/filozof/Dining%20Philosophers%20-%20Hybrid%20Applet.html
- http://www.doc.ic.ac.uk/ jnm/concurrency/classes/Diners/Diners.html
- http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/deadlock.html
- http://users.erols.com/ziring/diningAppletDemo.html

Gerenciamento de Memória

- Recurso importante
- Tendência atual do software:
  - Lei de Parkinson: "Os programas se expandem para preencher a memória disponível para eles" (adaptação)
  - "Work expands so as to fill the time available for its completion" (Cyril Northcote Parkinson)
- Hierarquia de memória:
  - Cache (como seu gerenciamento normalmente é feito pelo hardware, focaremos nas demais)
  - Principal
  - Disco

Gerenciamento de Memória

- Idealmente os programadores querem uma memória que seja:
  - Grande
  - Rápida
  - Não Volátil
  - De baixo custo
- Infelizmente a tecnologia atual não comporta tais memórias

Gerenciamento de Memória

- A maioria dos computadores utiliza Hierarquia de Memórias que combina:
  - Uma pequena quantidade de memória cache, volátil, muito rápida e de alto custo
  - Uma grande memória principal (RAM), volátil, com centenas de MB ou poucos GB, de velocidade e custo médios
  - Uma memória secundária, não volátil, em disco, com gigabytes (ou terabytes), de velocidade e custo baixos
- Cabe ao SO abstrair essa hierarquia em um modelo útil e então gerenciá-la

Gerenciamento de Memória

- Cabe ao Gerenciador de Memória:
  - Gerenciar a hierarquia de memória:
    - Para cada tipo de memória, gerenciar espaços livres/ocupados
  - Controlar as partes da memória que estão em uso e quais não estão, de forma a:
    - Alocar memória aos processos, quando estes precisarem
    - Localizar dados
    - Liberar memória quando um processo termina; e
    - Tratar do problema do swapping (quando a memória é insuficiente e faz-se necessário o chaveamento entre a memória principal e o disco)

Multiprogramação e Memória

- Multiprogramação melhora o uso da CPU – sem novidade
- Em quanto?

Multiprogramação e Memória

- Multiprogramação melhora o uso da CPU – sem novidade
- Em quanto?
  - Depende do quanto de E/S é feito
  - Suponha que os processos fiquem em execução efetiva apenas 20% do tempo em que reside na memória, quantos processos deveriam estar na memória para ocupar a CPU 100%?

Multiprogramação e Memória

- Multiprogramação melhora o uso da CPU – sem novidade
- Em quanto?
  - Depende do quanto de E/S é feito
  - Suponha que os processos fiquem em execução efetiva apenas 20% do tempo em que reside na memória, quantos processos deveriam estar na memória para ocupar a CPU 100%?
    - Em tese, 5

Multiprogramação e Memória

- Multiprogramação melhora o uso da CPU – sem novidade
- Em quanto?
  - Depende do quanto de E/S é feito
  - Suponha que os processos fiquem em execução efetiva apenas 20% do tempo em que reside na memória, quantos processos deveriam estar na memória para ocupar a CPU 100%?
    - Em tese, 5
  - E qual o problema com essa visão?

Multiprogramação e Memória

- Multiprogramação melhora o uso da CPU – sem novidade
- Em quanto?
  - Depende do quanto de E/S é feito
  - Suponha que os processos fiquem em execução efetiva apenas 20% do tempo em que reside na memória, quantos processos deveriam estar na memória para ocupar a CPU 100%?
    - Em tese, 5
  - E qual o problema com essa visão?
    - Presume que dois processos não estarão esperando, ao mesmo tempo, por E/S

Multiprogramação e Memória

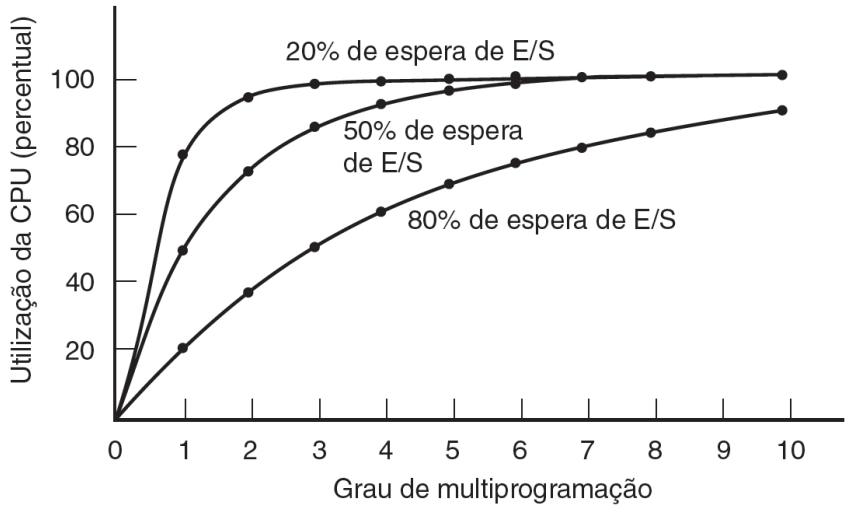
- Olhemos de um ponto de vista probabilístico
  - Suponha que um processo gaste uma fração $0 \leq p \leq 1$ de seu tempo esperando pela finalização de suas solicitações de E/S
    - Então a probabilidade de um determinado processo estar bloqueado por conta de E/S é p
  - Com n processos simultâneos na memória, a probabilidade de todos os n processos estarem esperando por E/S (situação em que a UCP está ociosa) é:

Multiprogramação e Memória

- Olhemos de um ponto de vista probabilístico
  - Suponha que um processo gaste uma fração $0 \leq p \leq 1$ de seu tempo esperando pela finalização de suas solicitações de E/S
    - Então a probabilidade de um determinado processo estar bloqueado por conta de E/S é p
  - Com n processos simultâneos na memória, a probabilidade de todos os n processos estarem esperando por E/S (situação em que a UCP está ociosa) é:
    - P^n

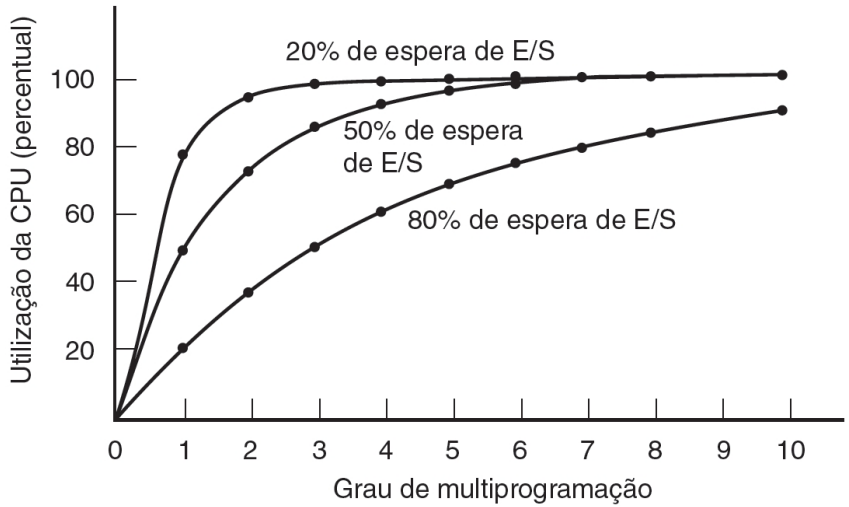
Multiprogramação e Memória

- Considerando a Utilização da CPU como a fração do tempo dos programas em que não estão esperando E/S ao mesmo tempo, temos
  - Utilização da CPU $= 1 - P^n$
    - Probabilidade dela estar sendo usada
  - Isso pressupondo:
    - Independência dos processos (o que não necessariamente ocorre, pois um processo pode ter que esperar outro para poder executar)



Multiprogramação e Memória

- Se processos gastarem 80% em E/S, precisaremos de no mínimo 10 processos na memória para que a ociosidade da CPU seja mantida inferior a 10%
  - 80% são comuns, quando se espera pelo usuário
- Mais memória $\rightarrow$ melhor aproveitamento da CPU
  - Computador "mais rápido" quando há mais memória (mas não linear)

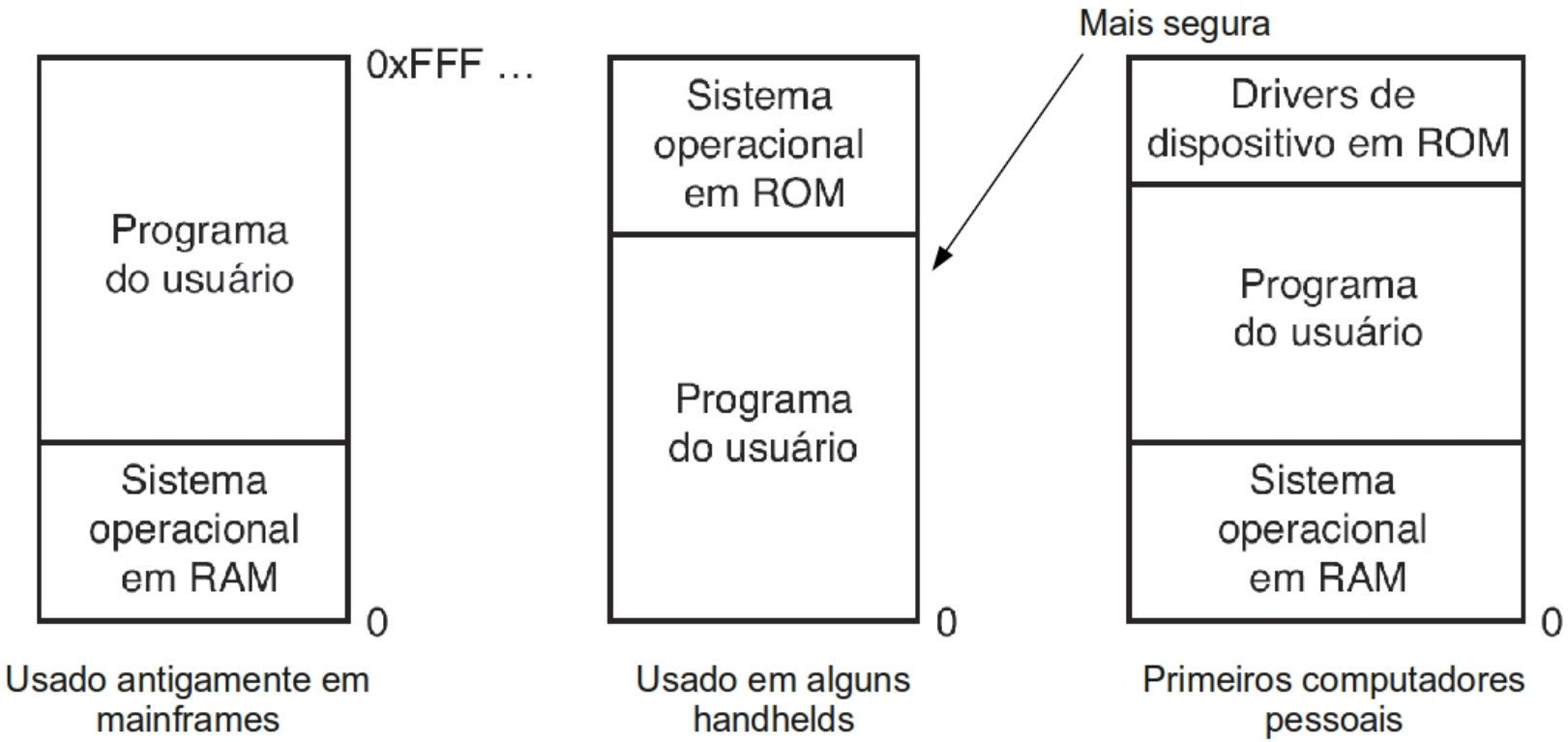


Abstrações de Memória

- Como a memória é vista pelos programas?
  - Sem abstração; ou
  - Espaços de endereços
- Sem abstração (< 1980):
  - A mais simples – programas simplesmente veem toda a memória física
  - Acessam-na diretamente, pelo seu endereço
  - Ex: MOV R1, 1000
    - Mova o conteúdo do endereço de memória 1000 ao registrador R1

Memória – Sem Abstração

- Organização:



À exceção da central, as outras duas possibilitavam que um erro no programa do usuário apagasse o SO

No terceiro modelo (MS-DOS), a porção do sistema na ROM era chamada de BIOS

Memória – Sem Abstração

- Sistemas assim geralmente são monousuários, rodando apenas um processo por vez
  - Do contrário, se um dos processos escrevesse em uma posição de memória, apagaria qualquer outro valor que outro processo tivesse armazenado ali
    - Lembre que todos os processos veem a memória toda
  - Toda a memória é alocada à próxima tarefa, incluindo a área do S.O
    - O usuário digita um comando, então o SO copia o programa do disco para a memória e o executa
  - Sistemas do usuário podem danificar o S.O., que deve ser recarregado

Memória – Sem Abstração

- Poderíamos ter multiprogramação assim?

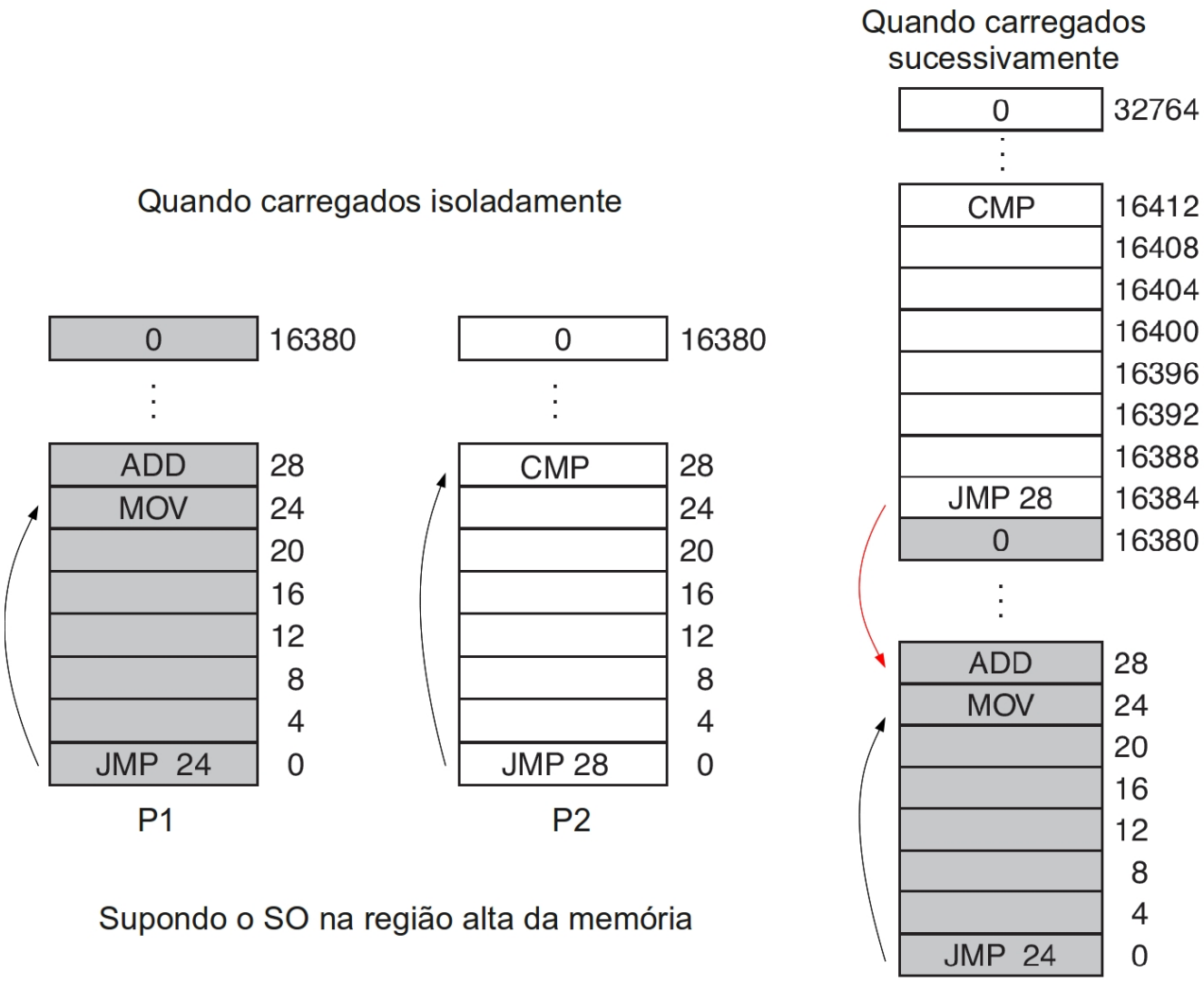
Memória – Sem Abstração

- Poderíamos ter multiprogramação assim?
  - Se o S.O. salvar todo o conteúdo da memória em disco, e então carregar e rodar o próximo programa, sim
    - Basta que haja apenas um programa na memória por vez
    - Este conceito é a base do swapping (mais adiante)
- Mas, sendo os programas pequenos, não poderíamos tê-los na memória ao mesmo tempo?

Memória – Sem Abstração

- Poderíamos ter multiprogramação assim?
  - Se o S.O. salvar todo o conteúdo da memória em disco, e então carregar e rodar o próximo programa, sim
    - Basta que haja apenas um programa na memória por vez
    - Este conceito é a base do swapping (mais adiante)
- Mas, sendo os programas pequenos, não poderíamos tê-los na memória ao mesmo tempo?
  - Embora os programas vejam a memória toda, eles não sabem que outros programas também estão rodando...
  - Teríamos que resolver o problema da violação (ainda que involuntária) de espaço de memória

Violação do Espaço de Memória



Violação do Espaço de Memória

- Uma solução seria adicionar um valor-base a cada endereço usado no segundo processo, enquanto carrega na memória
  - Técnica conhecida como Realocação estática
    - A transformação do endereço ocorre antes da execução começar
  - Caro – podemos ter que fazer muitas substituições
  - Exige distinguir entre o que é endereço e o que é valor numérico, nas diferentes instruções
- Ainda assim, tipo de esquema ainda usado:
  - Smart cards
  - Eletrodomésticos

} O usuário não controla que programas serão carregados

Realocação e Proteção

- Para permitir que múltiplas aplicações estejam na memória, precisamos resolver dois problemas: Proteção e Realocação
- Proteção:
  - Com vários programas ocupando diferentes espaços da memória é possível acontecer um acesso indevido
    - Como proteger os processos uns dos outros e o kernel de todos os processos?
    - Deve-se manter um programa fora do espaço de outros processos

Realocação e Proteção

- Realocação:
  - Como carregar processos em regiões da memória diferentes das explicitamente definidas em seu código?
  - Quando um programa é linkado (programa principal + rotinas do usuário + rotinas da biblioteca $\rightarrow$ executável) o linker deve saber em que endereço o programa irá iniciar na memória
  - Nesse caso, para que o linker não escreva em um local indevido, como por exemplo na área do SO (ex: 100 primeiros endereços), é preciso de realocação:
    - $\#100 + \Delta \rightarrow$ que depende dos demais processos!!!

Abstração de Memória

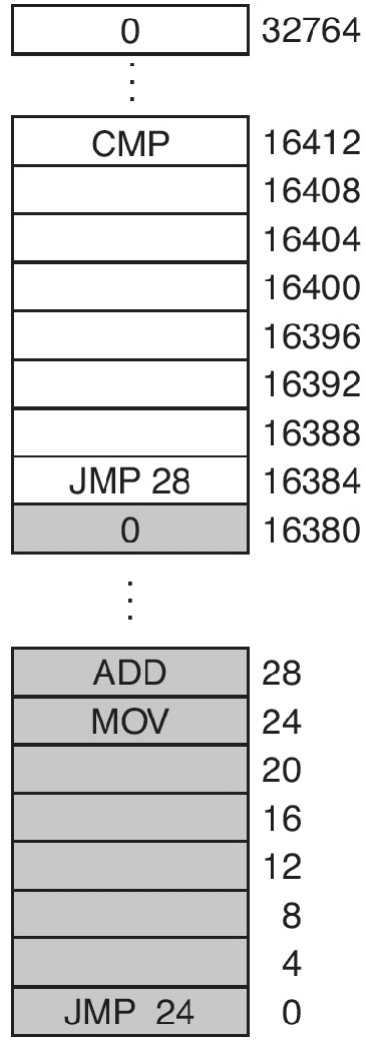
- Multiprogramação fica difícil com endereçamento direto
  - Podemos não ter certeza de onde o programa será carregado na memória
    - As localizações de endereços das variáveis e do código das rotinas não podem ser absolutos
- Solução para ambos os problemas:
  - Uma abstração – o espaço de endereçamento
  - Conjunto de endereços que um processo pode usar para endereçar memória
    - Cada processo tem seu próprio espaço de endereços

Abstração de Memória

- Como dar a cada programa seu próprio espaço de endereços?
  - De modo que o endereço 28 em um signifique uma localização física, na memória, diferente do 28 em outro
- Usando Realocação dinâmica
  - Mapeamento de cada espaço de endereço do processo em uma parte diferente da memória física
  - A transformação do endereço ocorre durante a execução do programa
  - Implementada com 2 registradores → base e limite
    - Protegidos pelo hardware contra modificações pelos usuários
    - Usados no intel 8088 (sem limite, mas com base para segmento de texto e dados)

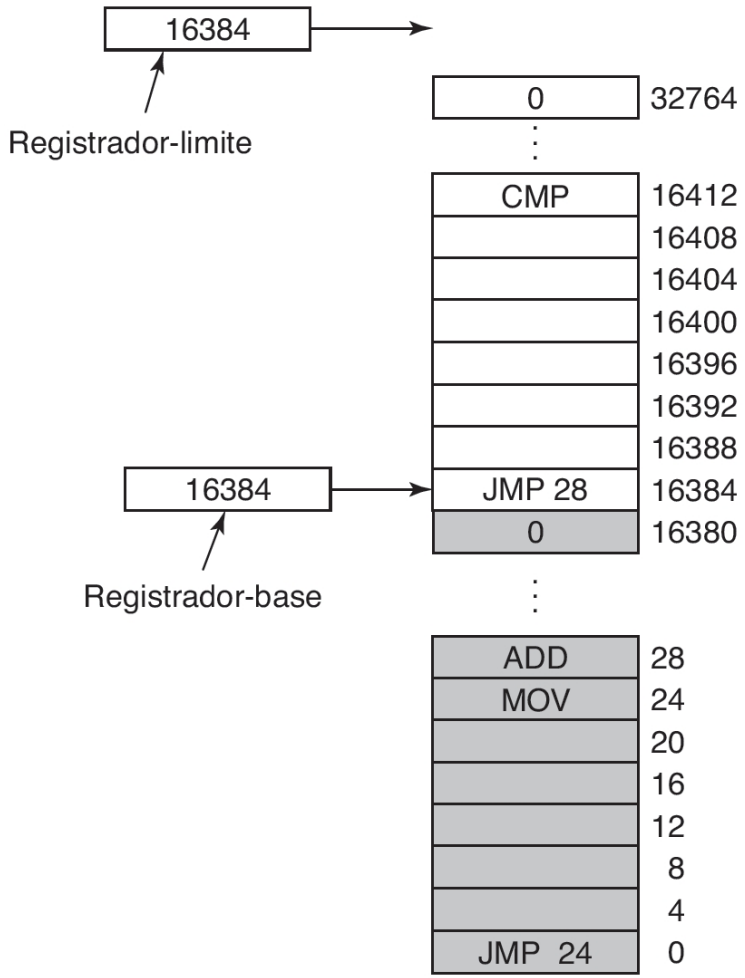
Abstração de Memória

- Quando usamos registradores base e limite, programas são carregados em posições consecutivas da memória
  - Onde haja espaço e sem realocação durante o carregamento
- Quando um processo é executado:
  - O registrador-base é carregado com o endereço físico de onde o programa começa na memória
  - O registrador-limite é carregado com o comprimento do programa



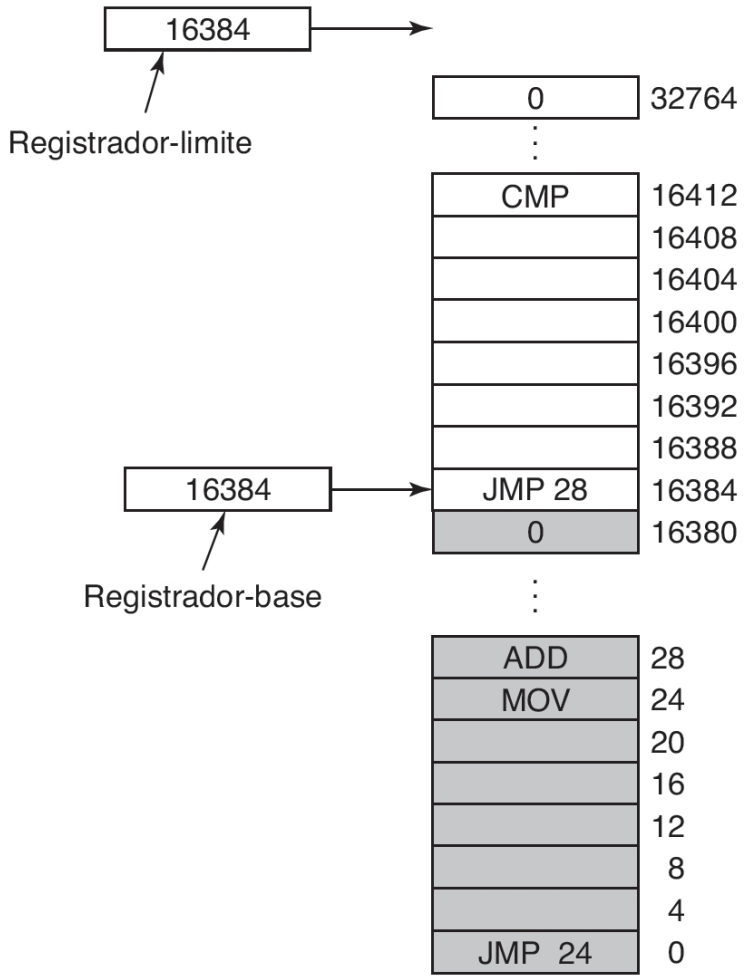
Abstração de Memória

- Ex:
  - Para o primeiro programa:
    - Base: 0
    - Limite: 16.384
  - Para o segundo programa
    - Base: 16.384
    - Limite:
      $32.768 - 16.384 = 16.384$
- Juntos, base e limite definem o espaço de endereçamento do programa



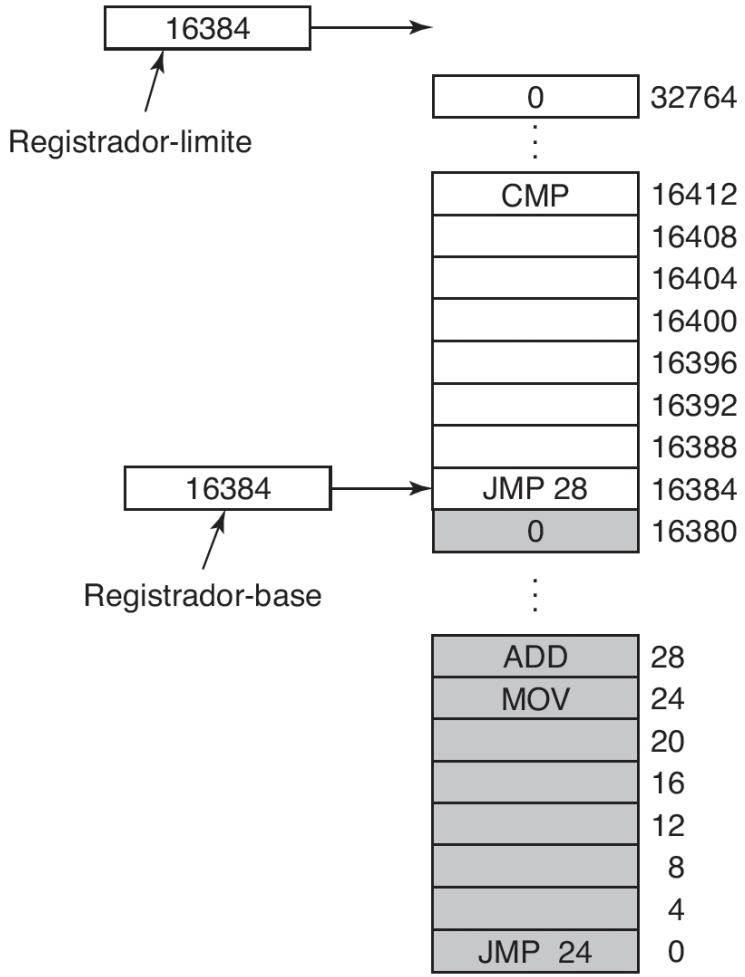
Abstração de Memória

- Toda vez que um processo referencia a memória:
  - A CPU (hardware) automaticamente adiciona o valor base ao endereço, antes de enviar ao barramento da memória
  - Ao mesmo tempo, checa se o endereço referenciado (antes da adição) é maior ou igual que o valor do registrador limite
    - Caso em que um erro é gerado



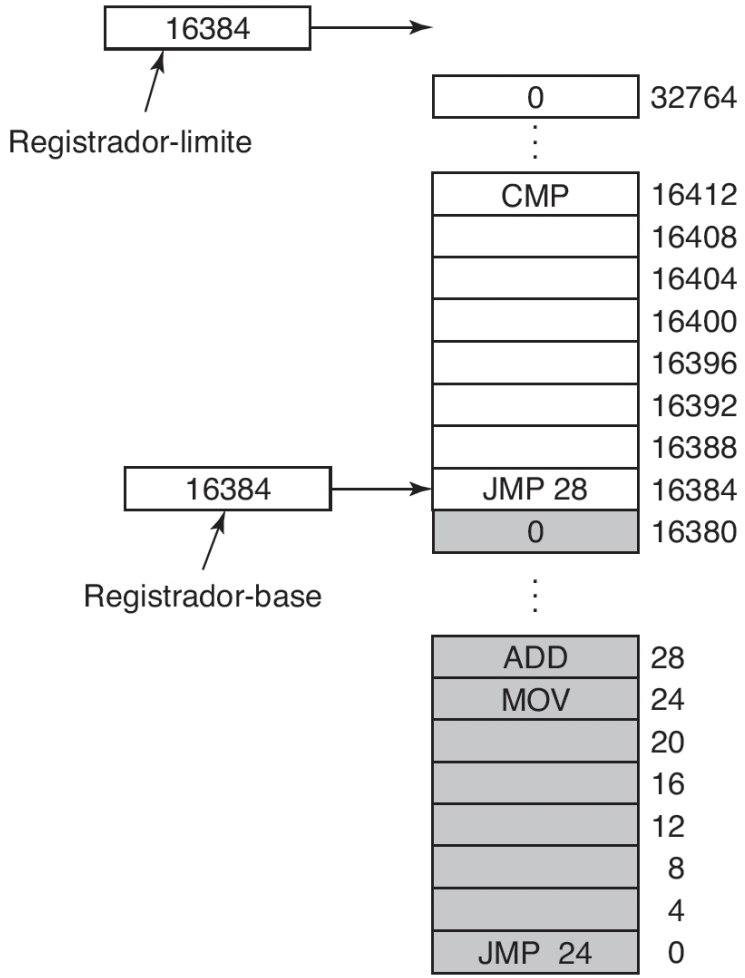
Abstração de Memória

- Ex:
  - No segundo programa, embora o processo execute JMP 28, o hardware trata como se fosse JMP 16.412
- O registrador-base torna impossível a um processo uma remissão a qualquer parte de memória abaixo de si mesmo
  - Previne acessos ao espaço de endereço de outro programa

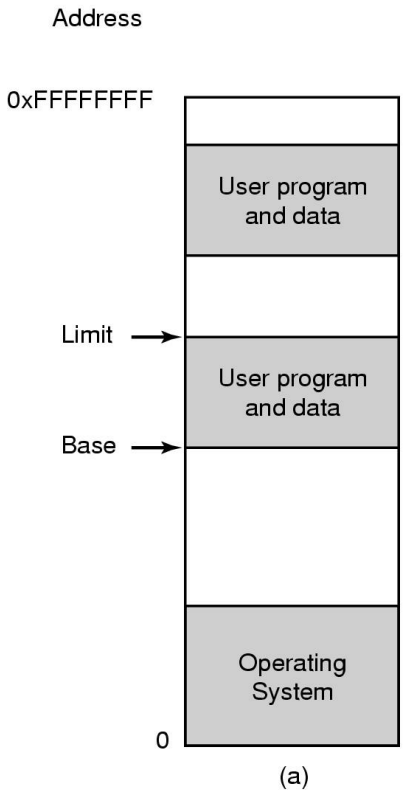


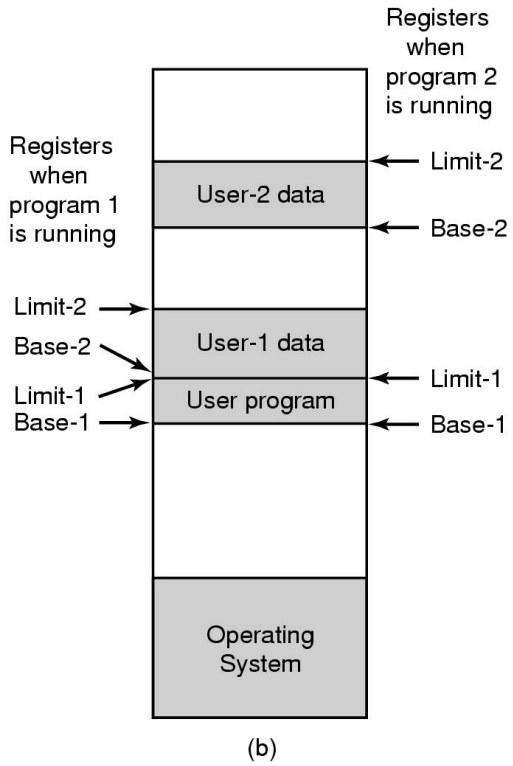
Abstração de Memória

- Quando há a troca de contexto:
  - O escalonador armazena o base e o limite no BCP do processo interrompido
  - Carrega o novo base e limite a partir do BCP do processo que será ativado



Abstração de Memória





Em vez de um par de registradores para dados e texto, o hardware pode usar dois pares: um para o segmento de texto e um para o segmento de dados

Isso faz com que processos que rodem o mesmo programa compartilhem o segmento de texto – ocupam menos memória

Abstração de Memória

- Técnica bastante simples, mas que caiu em desuso
  - Deu lugar a técnicas melhores e mais complicadas
- Desvantagem:
  - Faz uma comparação e soma a cada referência à memória