MINISO – MINISSISTEMA OPERACIONAL[1]

Roland Teodorowitsch[2] <roland.teodorowitsch@gmail.com>

Universidade Luterana do Brasil (Ulbra) – Curso de Ciência da Computação – Câmpus Gravataí
Av. Itacolomi, 3.600 – Bairro São Vicente – CEP 94170-240 – Gravataí – RS

Pontifícia Universidade Católica do Rio Grande do Sul – FACIN – Curso de Engenharia de Computação
Av. Ipiranga, 6681 Prédio 32 Sala 505 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

30 de maio de 2016

RESUMO

Este artigo descreve a implementação de um sistema operacional de pequeno porte, chamado miniSO (minisSistema Operacional), que suporta processos leves (*threads*). São discutidos tópicos relacionados à estrutura básica, compilação e execução, implementação de execução concorrente e estruturas de dados, bem como implementação de chamadas de sistema, primitivas de biblioteca e comandos internos.

Palavras-chave: Sistemas Operacionais; Projeto de Sistemas Operacionais; IBM/PC.

ABSTRACT

Title: *"miniSO – MiniOperating System"*

This paper describes the implementation of a small operating system, called miniSO (minisSistema Operacional or miniOperating System), that supports thread). Topics related to the basic structure, compilation and execution, implementation of concurrent execution and data structures are discussed, and also the implementation of system calls, library primitives and internal controls.

Key-words: *Operating Systems; Operating System Project; IBM/PC.*

1 INTRODUÇÃO

Este artigo descreve a implementação de um um pequeno sistema operacional com suporte a processos leves (*threads*), chamado miniSO (minisSistema Operacional). Este sistema operacional inclui basicamente: um interpretador de comandos (capaz de reconhecer alguns comandos internos); uma pequena biblioteca (que fornece serviços básicos para o interpretador de comandos e para os comandos internos); e um conjunto de chamadas de sistema (que são utilizadas pelas funções da biblioteca).

Este sistema operacional, na sua versão atual, deve ser compilado em um sistema operacional compatível com MS-DOS.

É importante destacar também que o miniSO é um sistema operacional didático, tendo como principal objetivo apresentar uma estrutura simples e que possa ser facilmente entendida por estudantes de graduação. Entre suas principais características estão:

- código-fonte reduzido (com o mínimo possível em linguagem de montagem e a maior parte em Linguagem C);
- facilidade de expansão;
- execução sobre a arquitetura Intel x86;
- uso do modo real do processador (ou seja, não tem suporte para memória virtual nem proteção de memória);
- uso de chamadas do BIOS (*Basic Input Output System*) para acesso a dispositivos;
- suporte a processos leves (*threads*);
- compilação usando ferramentas para MS-DOS;
- setor de inicialização desenvolvido em Bootlib (TEODOROWITSCH, 2000);
- imagem contendo interpretador de comandos e comandos internos embutidos (sem comandos externos na versão atual).

1 Artigo elaborado como material didático para disciplinas da área de Projeto de Sistemas Operacionais e Sistemas de Tempo Real.

2 Professor Assistente da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) e Professor Adjunto da Universidade Luterana do Brasil (Ulbra).

2 ESTRUTURA E CÓDIGO-FONTE DO MINISO

Geralmente o miniSO é distribuído em um compactado que foi criado para ser descompactado em um diretório chamado `c:\pso`. Ao ser descompactado, este arquivo criará a seguinte árvore de diretórios e arquivos:

- `setpath.bat`: deve ser executado a partir de uma janela MS-DOS para que o caminho de busca por executáveis (variável `PATH` do MS-DOS) inclua os utilitários necessários para compilação do miniSO;
- `miniSO/`: diretório que contém os arquivos com o código-fonte do miniSO;
- `bin/`: diretório que contém os utilitários necessários para compilação do miniSO;
- `include/`: diretório que contém os arquivos de inclusão do compilador C;
- `lib/`: diretório que contém as bibliotecas do compilador C.

O miniSO é composto por um setor de inicialização e por um arquivo com a imagem do sistema operacional.

O setor de inicialização (arquivo `boot.bin`), que deverá ser colocado no primeiro setor de um disco flexível de 1,44MB, é responsável pelo carregamento da imagem do sistema operacional (arquivo miniSO.bin).

Os arquivos utilizados para gerar o setor de inicialização e a imagem do sistema operacional são:

- `Makefile`: contém a especificação de como os componentes do miniSO devem ser compilados;
- `boot.bl`: programa em Bootlib utilizado para gerar o setor de inicialização (conforme descrito na Seção 2.1);
- `command.c`: código do interpretador de comandos, incluindo: o próprio interpretador, funções auxiliares e comandos internos;
- `command.h`: definições gerais do interpretador de comandos, incluindo: constantes, tipos e protótipos de funções;
- `init.c`: código inicial do sistema operacional, contém os procedimentos de inicialização, o que corresponde ao código do primeiro "processo" a ser executado;
- `lib.c`: contém a implementação das funções da biblioteca do miniSO, ou seja, das rotinas disponíveis para as "aplicações" que estiverem executando no sistema;
- `lib.h`: contém a definição das funções que fazem parte da biblioteca do miniSO;
- `miniSO.asm`: contém a implementação da rotina básica para atendimento da interrupção de chamadas de sistema e também rotinas necessárias à inicialização do miniSO, compartilhamento de tempo e chaveamento de contexto, em linguagem de montagem;
- `miniSO.bxrc`: contém a definição da máquina virtual que será utilizada para executar o miniSO no emulador Bochs;
- `miniSO.def`: contém definições comuns aos programas em linguagem C e ao código em linguagem de montagem (estas definições são passadas pelo `Makefile` através de parâmetros para o compilador e para o montador);
- `miniSO.h`: definições gerais do miniSO e protótipos de funções internas;
- `scall.c`: contém a implementação das chamadas de sistema em linguagem de alto nível, C.

2.1 O setor de inicialização

O setor de inicialização foi implementado utilizando uma linguagem chamada Bootlib (TEODOROWITSCH, 2000). Com esta linguagem é possível definir um setor de inicialização usando uma linguagem de alto nível e, a seguir, compilar esta definição para gerar uma versão em linguagem de montagem (*Assembly*). Caso a imagem do sistema operacional tenha apenas um setor, pode-se utilizar o programa em Bootlib apresentado na Figura 1 (arquivo `boot.bl`).

```
/* Arquivo: boot.bl */
boot(size=512,signature,segment=0x07c0)
{
  fat(oem="miniSO",volume="VOLUME",disk=FD1440);
  setstack(0x0000,0x7c00);
  setcolor(0x07);
  putstr("Carregando miniSO...");
  readdisk_chs(0x00,0,1,16,1,0x07e0,0x0000);
  jump(0x07e0,0x0000);
}
```

Figura 1 – Programa em Bootlib para carregamento de uma imagem com um setor

No programa da Figura 1, define-se que:

- o tamanho do setor de inicialização será de 512 bytes (`size=512`);
- o setor conterá, nos dois últimos bytes, a assinatura padrão para setores de inicialização (`signature`);
- o setor será colocado no segmento de código 0x07c0 (`segment=0x07c0`);
- o setor será gerado de forma a ser compatível com o sistema de arquivos FAT do MS-DOS: `fat(oem="miniSO",volume="VOLUME",disk=FD1440);`
- a pilha do sistema terá segmento igual a 0x0000 e deslocamento igual a 0x7c00: `setstack(0x0000,0x7c00);`
- a cor para impressão de mensagens será 0x07: `setcolor(0x07);`
- será exibida uma mensagem: `putstr("Carregando miniSO...");`
- será lido um (1) setor do disco 0x00, no cilindro 0, cabeça 1, número relativo de setor 16, que será colocado no endereço de memória formado pelo segmento 0x07e0 e pelo deslocamento 0x0000: `readdisk_chs(0x00,0,1,16,1,0x07e0,0x0000);`
- a execução será iniciada no endereço formado pelo segmento 0x07e0 e pelo deslocamento 0x0000: `jump(0x07e0,0x0000);`

É importante observar que este setor de inicialização exige que a imagem do sistema operacional seja colocada no cilindro 0, cabeça 1, setor relativo 16. Esta posição corresponde exatamente ao primeiro setor da área de dados de um disco de 3 1/2 polegadas, 1,44MB, formatado com o sistema de arquivos FAT-12. Para garantir que a imagem do sistema operacional esteja nesta posição, basta formatar um disco, ou utilizar um disco vazio, e fazer com que o primeiro arquivo copiado para este disco seja o arquivo que contém a imagem do sistema operacional.

Para carregar uma imagem de sistema operacional maior, será necessário modificar o arquivo `boot.bl`. Como o programa gerado pela Bootlib utiliza o BIOS (*Basic Input/Output System*), deve-se levar em consideração que há certas restrições nos serviços disponibilizados pelo BIOS. Para uma imagem do sistema operacional com até três setores, basta aumentar o número de setores na chamada `readdisk_chs()`, por exemplo, para:

```
readdisk_chs(0x00,0,1,16,3,0x07e0,0x0000);
```

Caso o arquivo ocupe mais de três setores, e considerando a posição em que a imagem se encontra, o BIOS não permite que se inicie uma leitura em uma trilha, terminado a mesma leitura em outra trilha. Será necessário, portanto, utilizar duas chamadas `readdisk_chs()`, calculando onde inicia a nova trilha e em que posição de memória este segmento da imagem deverá ser colocado. Por exemplo, para carregar uma imagem com até 21 setores, pode-se utilizar:

```
readdisk_chs(0x00,0,1,16,3,0x07e0,0x0000);
readdisk_chs(0x00,1,0,1,18,0x0840,0x0000);
```

Para carregar uma imagem com até 39 setores, pode-se utilizar:

```
readdisk_chs(0x00,0,1,16,3,0x07e0,0x0000);
readdisk_chs(0x00,1,0,1,18,0x0840,0x0000);
readdisk_chs(0x00,1,1,1,18,0x0a80,0x0000);
```

Caso sejam necessários de 40 até 57 setores, pode-se acrescentar mais uma chamada:

```
readdisk_chs(0x00,2,0,1,18,0x0cc0,0x0000);
```

É importante verificar, a cada compilação, se o setor de inicialização está compatível com o tamanho da imagem do sistema operacional. Caso isto não seja verdadeiro, o resultado será o carregamento parcial do sistema operacional, o que pode gerar resultados imprevisíveis.

Para compilar o programa `boot.bl` e gerar um arquivo chamado `boot.bin` são necessários os passos especificados na Figura 2.

```
bootlib -o boot.asm boot.bl
tasm boot
tlink boot
exe2bin boot boot.bin
```

Figura 2 – Passos para a compilação do setor de inicialização

Para colocar o arquivo `boot.bin` no primeiro setor do disco flexível, pode-se utilizar o programa `bootcopy.exe` que se encontra no diretório de utilitários do arquivo de distribuição do miniSO. A execução deste programa é simples, não necessitando de nenhum parâmetro. Ele sempre tentará transferir o arquivo `boot.bin` para o primeiro setor do disco.

2.2 A imagem do miniSO

A imagem do miniSO é formada basicamente por quatro tipos de código:

- código de inicialização, incluindo rotinas internas e a rotina de atendimento da interrupção do relógio;
- aplicações, incluindo o interpretador de comandos;
- biblioteca;
- chamadas de sistema.

O **código de inicialização**, função `main()` do arquivo `init.c`, corresponde ao primeiro código da imagem a ser executado. Neste código: define-se a pilha do primeiro processo leve (ou seja, do processo que está executando), define-se o registrador DS (*Data Segment*), desvia-se a interrupção 0x22 para uma rotina do núcleo (chamada `miniSO_systemcall`, implementada em `miniSO.asm`, que é responsável pela execução de chamadas de sistema), inicializa-se a tabela de processos (o que é feito na função `miniSO_init_proctable()` de `scall.c`, de forma que o processo leve armazenado no descritor 0 seja o processo corrente), captura-se o endereço da rotina de atendimento da interrupção do relógio, e desvia-se a interrupção do relógio (0x08) para uma rotina do núcleo (chamada `miniSO_clockhandler`, implementada `miniSO.asm`, que é responsável pelo escalonamento e pelo chaveamento de processos). A seguir inicia-se a execução do interpretador de comandos. Após o final da execução do interpretador de comandos (o que ocorrerá quando o usuário executar o comando "quit" ou "exit" na linha de comandos), o sistema é reinicializado chamando-se a função `bootstrap()`.

Este código de inicialização utiliza um conjunto de funções que constitui a biblioteca interna do miniSO. Trata-se de um conjunto de funções que devem ser utilizadas apenas no desenvolvimento do sistema operacional e nunca no desenvolvimento de aplicações. Estas funções, que estão codificadas em linguagem de montagem no arquivo `miniSO.asm`, estão descritas no Quadro 1.

Quadro 1 – Funções implementadas em `miniSO.asm`

Função / Protótipo	Descrição
`void setvect (unsigned interruptno, void interrupt (*isr) (void));`	Define a função `isr` como rotina de atendimento da interrupção `interruptno`.
`void interrupt (*getvect (int interruptno) )(void);`	Obtém o endereço da rotina de atendimento da interrupção `interruptno`.
`void enable (void);`	Habilita as interrupções.
`void disable (void);`	Desabilita as interrupções.
`void far *MK_FP (unsigned seg, unsigned off);`	Retorna o endereço de memória montado a partir do segmento (`seg`) e do deslocamento (`off`) fornecidos.
`unsigned FP_SEG (void far *fp);`	Retorna o segmento de um endereço de memória (`fp`).
`unsigned FP_OFF (void far *fp);`	Retorna o deslocamento de um endereço de memória (`fp`).
`void bootstrap (void);`	Executa o procedimento de inicialização do computador chamando a interrupção 19h do BIOS.
`void init_ds (unsigned ds);`	Define o valor do registrado DS com o valor de `ds`.
`void init_stack (unsigned ss, unsigned sp);`	Define o valor dos registradores SS e SP com os valores de `ss` e sp, respectivamente.

O arquivo `miniSO.asm` possui também uma série de rotinas que não serão chamadas diretamente nem pelo usuário nem pelo desenvolvedor do sistema operacional. Tratam-se de rotinas que fazem parte da biblioteca do compilador para manipulação de inteiros dos tipos "long" e "long unsigned". O compilador gera, para o modo real dos processadores x86, instruções que utilizam apenas registradores de 16 bits nas operações. Isto significa que operações com inteiros do tipo "long" ou "long unsigned", que utilizam 32 bits, não poderiam ser feitas diretamente colocando o valor destas variáveis em registradores. Algumas operações como multiplicação, divisão, módulo e deslocamento com estes tipos de dados devem ser executadas por rotinas específicas. Quando se utiliza alguma operação destas com algum dos tipos em questão, automaticamente o compilador insere chamadas para um rotinas da biblioteca padrão do compilador C que executam as respectivas operações. No entanto, como nenhuma biblioteca do compilador (que são bibliotecas específicas para MS-DOS) será ligada à imagem do miniSO, isto significa que durante a geração da imagem ocorreria um erro de ligação, pois o compilador insere chamada para rotinas da biblioteca e estas rotinas não estarão presentes. A solução encontrada para evitar este problema foi implementar as mesmas rotinas que o compilador espera utilizar para manipular inteiros longos, com o mesmo nome e com exatamente a mesma semântica, dentro do arquivo `miniSO.asm`. Estas rotinas são: `N_LXMUL@`, `LXMUL@`, `N_LDIV@`, `LDIV@`, `N_LUDIV@`, `LUDIV@`, `N_LMOD@`, `LMOD@`, `N_LUMOD@`, `LUMOD@`, `N_LXLSH@` e `LXLSH@`. O programador que estiver desenvolvendo código para o miniSO não precisa, no entanto, se preocupar em chamar estas rotinas. Isto é feito de forma transparente quando se executam operações com inteiros longos no código em C.

Depois do código de inicialização, o controle é passado para o interpretador de comandos, função `command()` de `command.c`. E a partir deste interpretador de comandos, é possível iniciar a execução de diversos comandos internos.

O interpretador de comandos e seus comandos internos constituem o que se pode chamar de **nível de aplicação** dentro do miniSO. Processos neste nível não utilizam as chamadas de sistema diretamente. Na verdade eles apenas utilizam chamadas da biblioteca padrão do miniSO.

A **biblioteca** padrão do miniSO corresponde a um conjunto de rotinas codificadas no arquivo `lib.c`. O principal objetivo desta biblioteca é evitar que os programas necessitem trabalhar diretamente com interrupções para executar chamadas de sistema.

No nível mais baixo do miniSO estão as **chamadas de sistema**. O ponto de entrada das chamadas de sistema (`miniSO_systemcall`) está implementado em `miniSO.asm`, e as chamadas propriamente ditas estão todas implementadas em `scall.c`. Basicamente a rotina `miniSO_systemcall` recebe no registrador AH o número da chamada de sistema e passa a execução para a função `scall()`, que executa a respectiva chamada.

A Figura 3 mostra todos os arquivos utilizados para gerar a imagem do miniSO, bem como os passos que devem ser seguidos na sua criação. Inicialmente os programas em C (`init.c`, `command.c`, `lib.c` e `scall.c`) devem ser compilados, gerando os respectivos arquivos em linguagem de montagem. A Figura 4 mostra os comandos utilizados para executar esta compilação. As opções de compilação `-1- -S -ms! –TDc –v-` habilitam, respectivamente, geração de código para o processador 8088/8086, geração de código em linguagem de montagem, uso do modelo de memória *Small*, geração executável do tipo ".COM" e geração de código sem possibilidade de depuração. A opção `–D`, utilizada na compilação dos arquivos `init.c`, `lib.c` e `scall.c`, define constantes (veja maiores detalhes sobre estas constantes na Seção 2.2.1).

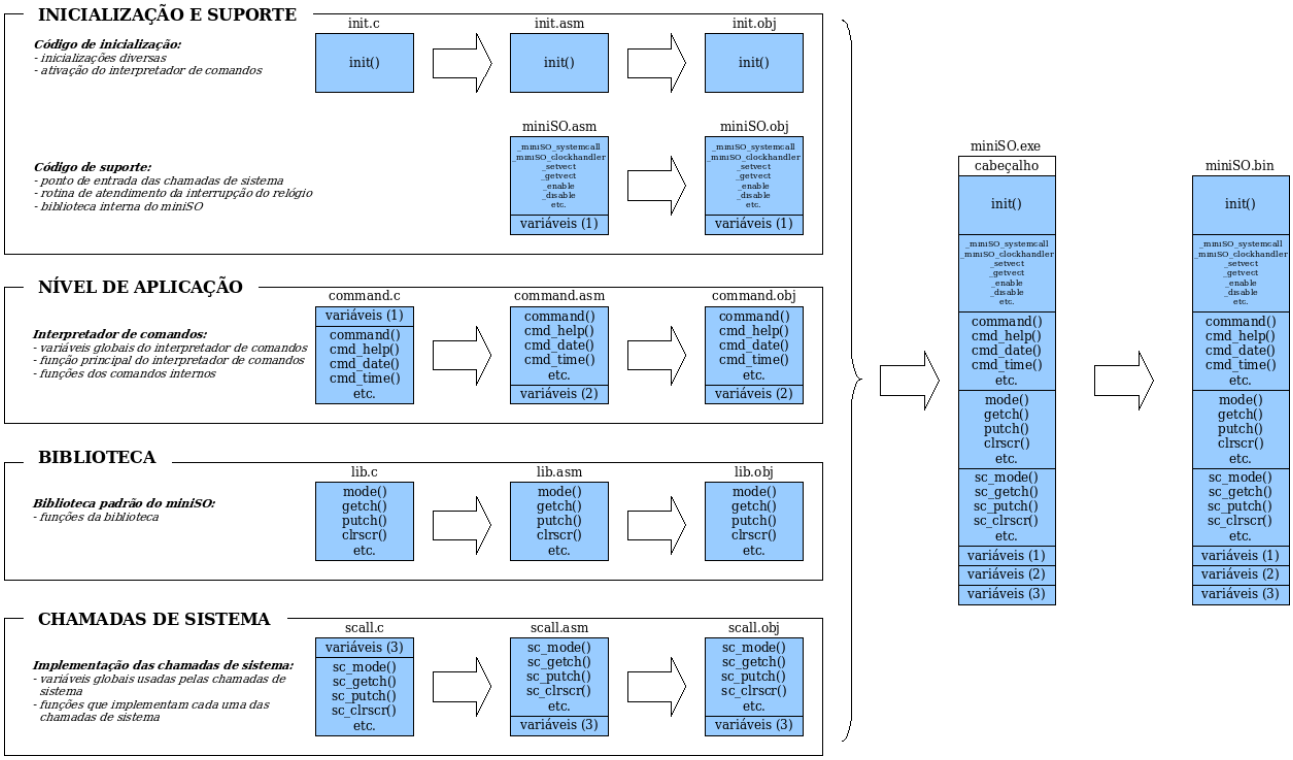

Figura 3 – Formação da imagem do miniSO

```
bcc -1- -S -ms! -TDc -v- -DminiSO_VERSION="$(VERSION)" command.c
bcc -1- -S -ms! -TDc -v- -DminiSO_CODESEGMENT=0x$(CODE) -DminiSO_DATASEGMENT=0x$(CODE)+0x$(DATA)
lib.c
bcc -1- -S -ms! -TDc -v- -DminiSO_CODESEGMENT=0x$(CODE) -DminiSO_DATASEGMENT=0x$(CODE)+0x$(DATA)
scall.c
bcc -1- -S -ms! -TDc -v- -DminiSO_CODESEGMENT=0x$(CODE) -DminiSO_DATASEGMENT=0x$(CODE)+0x$(DATA)
init.c
```

Figura 4 – Comandos utilizados na compilação dos arquivos em C

O próximo passo é executar a montagem de todos os programas em linguagem de baixo nível, ou seja, transformar os arquivos `.ASM` em `.OBJ`. Isto pode ser feito com os comandos mostrados na Figura 5. A opção `/zn` do montador é utilizada para desabilitar informações para depuração. A opção `/d` é utilizada para definição da constante `miniSO_DATASEGMENT` (veja maiores detalhes sobre esta constante na Seção 2.2.1).

```
tasm /zn command,,,
tasm /zn lib,,,
tasm /zn scall,,,
tasm /zn init,,,
tasm /zn /dminiSO_DATASEGMENT=$(CODE)h+$(DATA)h miniSO,,,
```

Figura 5 – Comandos utilizados na compilação dos arquivos em linguagem de montagem

A seguir, gera-se o arquivo executável `miniSO.exe`, que inclui o código de todos os programas que fazem parte do miniSO. O comando para executar a ligação é mostrado na Figura 6. A opção `/s` gera um arquivo chamado `miniSO.map` que contém um mapa detalhado de todos os segmentos (código e dados) do programa `miniSO.exe`.

```
tlink /s init command lib miniSO scall,miniSO,,,
```

Figura 6 – Comando utilizado na ligação do miniSO

Por fim, é necessário remover o cabeçalho do arquivo executável (que contém informações específicas para o sistema operacional MS-DOS, onde a compilação está sendo executada), isto é feito

através do utilitário `exe2bin.exe`, que converte um arquivo `.EXE` em `.BIN`. Este último passo é mostrado na Figura 7.

```
exe2bin miniSO miniSO.bin
```

Figura 7 – Transformação do arquivo executável em binário

2.2.1 A inicialização do registrador DS

A Figura 3, apresentada na seção anterior, mostra como o código da imagem do miniSO é gerado. Basicamente os segmentos de código e dados de cada arquivo são agrupados em dois segmentos únicos, um para código e outro para dados. Isto significa que na imagem haverá um segmento de código seguido de um segmento de dados. O segmento de código deve ser apontado pelo registrador CS (*Code Segment*). Como o setor de inicialização carrega a imagem (`miniSO.bin`) no endereço de memória 07e0:0000 e a seguir executa um salto intersegmento para este endereço, automaticamente CS receberá 07e0 e IP (*Instruction Pointer*) receberá 0x0000. Isto está coerente com o código gerado, uma vez que os deslocamentos dos rótulos associados ao código também iniciam em 0x0000.

No entanto, o registrador DS (*Data Segment*) precisa ainda ser inicializado com o valor do segmento onde os dados se encontram. O valor deste segmento depende diretamente do tamanho do segmento de código, pois o segmento de dados é colocado logo depois do segmento de código. O problema está no fato de que a informação sobre o tamanho do segmento de código (e, naturalmente, sobre o início do segmento de dados) faz parte do cabeçalho do arquivo executável, que é perdido quando o arquivo `miniSO.exe` é convertido para `miniSO.bin`.

Para resolver o problema, deve-se executar a ligação (comando `tlink`) com a opção `/s`, conforme mostrado na Figura 6. Isto faz com que o ligador gere um arquivo chamado `miniSO.map`, que contém informações relacionadas à geração do programa e aos segmentos que compõem o programa. A Figura 8 mostra o início de um arquivo `miniSO.map`.

```
Start  Stop   Length Name               Class

 00000H 0278DH 0278EH _TEXT              CODE
 0278EH 03333H 00BA6H _DATA              DATA
 03334H 036D0H 0039DH _BSS               BSS


Detailed map of segments

 0000:0000 00B8 C=CODE   S=_TEXT          G=(none)  M=INIT.C     ACBP=28
 0000:00B8 0EED C=CODE   S=_TEXT          G=(none)  M=COMMAND.C  ACBP=28
 0000:0FA5 07AD C=CODE   S=_TEXT          G=(none)  M=LIB.C      ACBP=28
 0000:1752 01AF C=CODE   S=_TEXT          G=(none)  M=MINISO.ASM ACBP=28
 0000:1901 0E8D C=CODE   S=_TEXT          G=(none)  M=SCALL.C    ACBP=28
 0278:000E 004E C=DATA   S=_DATA          G=DGROUP  M=INIT.C     ACBP=48
 0278:005C 09C2 C=DATA   S=_DATA          G=DGROUP  M=COMMAND.C  ACBP=48
 ...
```

Figura 8 – Exemplo de arquivo `miniSO.map`

Basicamente, o que este arquivo diz é que: o segmento de código começa em 0x00000 (segmento 0x0000, deslocamento 0x0000) e termina em 0x0278D (segmento 0x0000, deslocamento 0x278D); e que o segmento de dados começa em 0x0278E (segmento 0x0278, deslocamento 0x000E) e termina em 0x03333 (segmento 0x0278, deslocamento 0x0BB3). Em outras palavras, o ligador está informando que o código foi gerado como se o arquivo fosse ser executado a partir do segmento 0x0000 e que os dados estariam a partir do segmento 0x0278. No entanto, a imagem do miniSO é sempre colocada a partir do endereço 0x07e0. O que implica que os dados estejam a partir do segmento: 0x07e0+0x0278. É com este valor que o registrador DS deve ser inicializado. Isto deve ser feito no procedimento de inicialização (função `main()` do arquivo `init.c`), chamando a função `init_ds()`, e também na função `miniSO_clockhandler` (no arquivo `miniSO.asm`), para que esta função possa ter acesso às variáveis globais do miniSO. O valor do segmento de código e de dados foi, desta forma definido em duas constantes, respectivamente, `miniSO_CODESEGMENT` e `miniSO_DATASEGMENT`, ambas definidas dentro do arquivo `miniSO.def`. O valor destas constantes será passado durante a compilação, montagem e ligação para os respectivos

programas. A Figura 9 mostra o conteúdo do arquivo `miniSO.def`, que contém, além da definição do código (`CODE`) e do deslocamento até o segmento de dados (`DATA`), a versão do miniSO (`VERSION`).

```
# ----- Variaveis --------------------------------
# Deslocamento ate o inicio do segmento de dados (hexadecimal)
DATA    = 0278
# Segmento de codigo onde a imagem do miniSO sera colocada (hexadecimal)
CODE    = 07e0
# Versao do miniSO
VERSION = 2007
```

Figura 9 – Definição de constantes no `Makefile`

Para que esta informação seja adequadamente codificada na imagem do miniSO, é preciso, no entanto, após cada alteração feita no código do sistema que implique em alteração no tamanho do código, seguir os seguintes passos:

- compilar o sistema, gerando um arquivo `miniSO.map`, executando: `make`;
- examinar o arquivo `miniSO.def`, identificando onde começa o início do segmento de dados;
- atualizar a definição da constante `DATA` com este valor, no arquivo `miniSO.def`;
- eventualmente limpar os arquivos antigos, executando: `make clean`[3];
- compilar novamente o sistema, com: `make`;
- agora sim, instalá-lo no disco flexível (com `make install`) ou executá-lo a partir do emulador BOCHS (com `make run`).

2.3 Mapa da memória do miniSO

Quando a máquina é ligada, o BIOS carregará o setor de inicialização (de 512 bytes) no endereço 07c0:0000 (ou 0000:7c000 ou 0x07c00). Isto significa que o setor de inicialização ocupará as posições de memória de 0x07c00 até 0x07dff. A seguir este setor de inicialização carrega a imagem do miniSO apartir do endereço 07e0:0000 (ou 0000:7e00 ou 0x07e00). O tamanho da imagem do sistema operacional varia de acordo com a quantidade de código incorporada a ele.

Será também necessário reservar memória para as pilhas de cada um dos processos leves em execução no sistema. Estas pilhas serão alocadas a partir do endereço 0xd000 (constante `miniSO_INITSTACKS` de `miniSO.h`) e terão, cada uma, tamanho igual a 1536 bytes (0x600, constante `SO_STACKSIZE` de `miniSO.h`). Desta forma, o total de memória alocada para as pilhas varia conforme o número de processos leves suportado pelo sistema (o que está definido na constante `miniSO_MAXTHREADS` de `miniSO.h`).

Para determinar o endereço inicial da pilha de determinado processo leve, pode-se utilizar as seguintes fórmulas:

Segmento = miniSO_INITSTACKS+(m iniSO_STACKSIZE>>4)*pcb

Deslocamento = miniSO_STACKSIZE

onde *pcb* é o número do Bloco de Controle de Processo reservado para este processo.

3 Compilação do miniSO

Considerando o que foi dito nas seções que descrevem a estrutura e o código do miniSO, os passos necessários para compilação do miniSO são:

- certificar-se que o diretório de fontes do miniSO está instalado, ou seja, que os fontes estejam no diretório `c:\pso`;
- certificar-se que o comando `c:\pso\setpath.bat` foi executado. Este arquivo define a variável `PATH` para que os comandos necessários à compilação sejam encontrados;
- no diretório `c:\pso\miniSO`, executar `make`;
- executar `more miniSO.map` para verificar qual o tamanho do segmento de código, ou seja, onde inicia o segmento de dados (o próprio comando `make` mostra um relatório com este valor,

3 Este procedimento não é necessário, mas eventualmente podem surgir problemas com datas e horas de alterações de arquivos, principalmente quando se utiliza um ambiente virtual como DOSBOX para compilar e outro como Bochs para executar o miniSO, o que pode fazer com que a compilação não funcione como deveria. A recomendação neste caso, é executar `make clean` e, a seguir, `make`.

dispensando a verificação manual do arquivo `miniSO.map`);
- editar `miniSO.def` para atualizar o valor da variável `DATA` com o valor encontrado em `miniSO.map`;
- quando for necessário, executar `make clean` para apagar os restos da última compilação;
- colocar um disco vazio na unidade de discos flexíveis e executar `make install` ou executar `make run` para rodar o miniSO a partir do emulador Bochs.

4 SUPORTE À EXECUÇÃO CONCORRENTE

Para implementação de concorrência em qualquer sistema operacional, é fundamental conhecer a forma como os processadores tratam as interrupções. Para permitir a execução concorrente de múltiplos processos leves (*threads*) dentro de uma mesma aplicação ou código, o sistema operacional (ou a biblioteca que implementa processos leves) deve desviar a interrupção do relógio para uma rotina de tratamento de interrupções própria. Esta rotina será responsável pelo chaveamento entre os processos leves.

4.1 O controle de interrupções

Quando um programa está sendo executado e o *hardware* detecta algum evento de mais alta prioridade, é gerada uma interrupção. O atendimento da interrupção suspende, geralmente de forma temporária, a execução do programa que está em execução. Uma rotina de tratamento é então executada, e, em seguida, o programa que havia sido interrompido deve continuar a sua execução normalmente.

Na arquitetura Intel x86, plataforma sobre a qual o miniSO é executado, o controle das interrupções é feito usando uma tabela de vetores de interrupções. O número que identifica a interrupção é utilizado como índice nesta tabela de vetores de interrupções para obter o endereço da rotina de atendimento que deverá ser executada.

Os vetores de interrupções ocupam os primeiros 1024 *bytes* da memória (endereços 0000:0000h até 0000:03FFh). Existem 256 entradas nesta tabela, o que permite até 256 tipos diferentes de interrupções. Cada entrada é composta de 2 palavras (4 *bytes*). A palavra mais alta contém o segmento, e a palavra mais baixa o deslocamento (*offset*) do endereço da rotina de atendimento. A Figura 10 mostra o posicionamento da tabela de vetores de interrupções na memória e a composição de um de seus elementos.

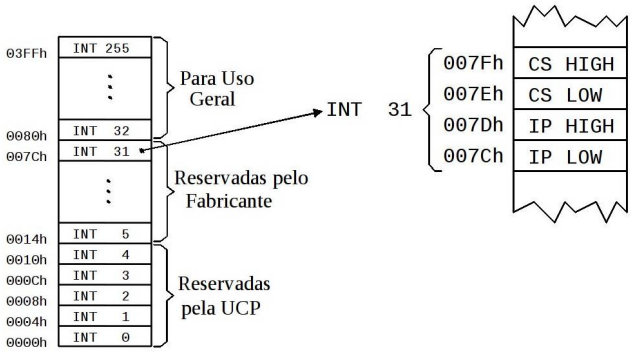

Figura 10 – Tabela de vetores de interrupções

Ao detectar uma interrupção x, o processador:
- termina de executar a instrução atual e avança o registrador IP (*Instruction Pointer*) para a próxima instrução;
- subtrai 2 do registrador SP (*Stack Pointer*) e coloca o registrador de estados (*flags*) nas posições de memória apontadas por SS:SP, ou seja, armazena o registrador de estados na pilha;
- desativa os *bits* de estado IF[4] (*Interrupt Flag*) e TF[5] (*Trap Flag*);
- subtrai 2 de SP e coloca o registrador CS (*Code Segment*) em SS:SP;
- subtrai 2 de SP e coloca IP em SS:SP;
- multiplica x por 4 para descobrir o endereço da posição de memória onde está o endereço da

4 Se IF estiver habilitado, processadores da família x86 atenderão as interrupções de *hardware* mascaráveis. Caso contrário, as interrupções mascaráveis não serão atendidas. Em linguagem de montagem, a instrução STI habilita este *flag* e a instrução CLI o desabilita.

5 TF permite que o processador funcione em modo de passo único (*single-step mode*). Neste modo, geralmente utilizado por depuradores, o processador executa uma instrução e, a seguir, gera uma interrupção do tipo 1. O processador 8086 não tem instruções para ativar ou desativar este *flag* diretamente.

rotina de atendimento da interrupção x;
- executa um desvio para o endereço obtido no passo anterior, carregando este valor nos registradores CS e IP.

A partir deste momento, a rotina de atendimento da interrupção será executada. Esta rotina deve encerrar a sua execução com a instrução `IRET` (*Interrupt RETurn*). A instrução `IRET` efetua um retorno intersegmento, restaurando da pilha o conteúdo dos registradores IP, CS e de estados. A pilha deve conter, em ordem crescente de endereço: IP, CS e o registrador de estados. A sequência de operações executada por `IRET` é:

- carrega o registrador IP com o conteúdo das posições de memória apontadas por SS:SP, soma 2 a SP;
- carrega o registrador CS com o conteúdo das posições de memória apontadas por SS:SP, soma 2 a SP;
- carrega o registrador de estados com o conteúdo das posições de memória apontadas por SS:SP, soma 2 a SP.

É importante observar que apenas o registrador de estados e os registradores CS e IP são automaticamente salvos. Os demais registradores, caso venham a ser alterados, devem ser salvos e recuperados pela rotina de tratamento da interrupção.

4.2 Criação de uma rotina de atendimento de interrupções

Em C, pode-se especificar uma função como sendo uma função de tratamento ou atendimento de interrupções através do modificador `interrupt`. Por exemplo, a declaração da função apresentada na Figura 11 especifica que `clockhandler()` será uma função que poderá ser utilizada para atendimento de interrupções.

```
void interrupt clockhandler()
{
      [...]
}
```

Figura 11 – Uso do modificador `interrupt`

O código gerado para a função `clockhandler()` virá precedido de uma série instruções para salvar todos os registradores na pilha e, consequentemente, será encerrado por uma série de instruções para recuperar estes registradores, além da instrução de retorno de interrupção (`IRET`). A Figura 12 mostra o código em linguagem de montagem gerado pelo compilador para a função da Figura 11.

```
; salva registradores
push ax
push bx
push cx
push dx
push es
push ds
push si
push di
push bp
; faz ds apontar para a área de dados da rotina
mov  bp,offset DADOS_ROTINA
mov  ds,bp
; faz bp apontar para SP
mov  bp,sp
; Reserva espaço na pilha para as variáveis locais da rotina
sub  sp,TAM_DADOS      ; (*) Só aparece quando há variáveis locais
;
; Código da função: ...
;
; Restaura o registrador SP ao seu valor original
mov  sp,bp             ; (*) Só aparece quando há variáveis locais
; recupera o valor dos registradores
pop  bp
pop  di
pop  si
pop  ds
pop  es
pop  dx
pop  cx
pop  bx
pop  ax
iret
```

Figura 12 – Código em linguagem de montagem gerado quando `interrupt` é usado

É importante lembrar que o registrador de estado e os registradores IP e CS serão salvos automaticamente quando a interrupção for ativada e recuperados quando a instrução `IRET` for executada. As instruções da Figura 12 cuja linha de comentário está marcada com "(*)" só aparecerão no código gerado caso a rotina de atendimento de interrupções possua variáveis locais. Deve-se prever também a possibilidade de que as variáveis locais sejam alocadas, através de otimizações de compilação, em registradores e não na pilha como é mais comum.

4.3 Definição de rotinas de tratamento de interrupções

É possível obter o endereço de uma rotina de tratamento de interrupções usando a função `getvect()`, que recebe como parâmetro o número de uma interrupção e retorna o endereço da função de atendimento de interrupções correspondente. O protótipo de `getvect()` e um trecho de código que poderia ser usado para obter o endereço de atendimento da interrupção 0x08 é mostrado na Figura 13.

```
/* Protótipo da função getvect() */
void interrupt (*getvect(int interruptno))();

/* Declaração de uma variável para receber o endereço de uma rotina de atendimento de interrupções
*/
void interrupt (*ender)();

/* Obtém o endereço da rotina de atendimento de interrupções 0x08 */
ender = getvect (0x08);
```

Figura 13 – Uso da função `getvect()`

Para definir o endereço de uma rotina de atendimento de interrupções, deve-se utilizar a função `setvect()`. Esta função recebe como parâmetros o número da interrupção e o endereço da nova rotina de atendimento de interrupções, sem retornar nenhum valor. A Figura 14 mostra o protótipo da função `setvect()` e um trecho de código que poderia ser usado para definir o novo endereço da rotina de atendimento da interrupção 0x08.

```
/* Protótipo da função setvect() */
void setvect(int interruptno, void interrupt (*isr)());

/* Código da nova rotina de atendimento da interrupção 0x21 */
void interrupt nova_int_08h()
{
       [...]
}

[...]
/* Define o endereço de atendimento da interrupção 0x21 */
setvect(0x21,nova_int_08h);
[...]
```

Figura 14 – Uso da função `setvect()`

4.4 Compartilhamento de tempo

A técnica utilizada para implementar processos leves no miniSO é bastante simples. Para ter múltiplas linhas de execução dentro do mesmo código é preciso que o tempo do processador seja dividido de alguma forma entre as linhas de execução ativas. Isto pode ser obtido desviando a rotina de atendimento da interrupção do relógio, de número 0x08, para uma rotina própria do núcleo do miniSO.

Como o controlador de interrupções das máquinas IBM/PC gera interrupções periódicas, é possível aproveitar estas interrupções para executar o chaveamento de processos leves. Os atuais computadores da arquitetura Intel x86 utilizam, como temporizador uma pastilha equivalente ao 8254 (originalmente utilizado com os computadores PC AT). O temporizador está programado para, através da controladora de interrupções, gerar 18,2 vezes por segundo a interrupção 0x08. O MS-DOS, por exemplo, utiliza estas interrupções para atualizar o seu relógio interno.

Levando-se estas informações em consideração, os seguintes passos são necessários para implementar o compartilhamento de tempo entre processos leves:

- obter o endereço da rotina de atendimento da interrupção atual do relógio (usando a função `getvect()`), de modo que se possa restaurar o estado da máquina quando a execução do miniSO for encerrada;
- criar uma função para tratamento da interrupção do relógio e desviar o respectivo vetor de interrupção para esta função (usando a função `setvect()`);
- a partir daí todos os processos leves cadastrados receberão uma fatia de tempo, no mínimo correspondente ao intervalo de tempo entre duas interrupções do relógio. Um processo leve executa até que ocorra uma interrupção do relógio. Neste momento a rotina de atendimento salva o contexto do processo leve e restaura o contexto de um novo processo leve. Quando a UCP retorna da interrupção, um novo processo leve será executado;
- antes de retornar da interrupção, pode-se chamar a rotina original de atendimento da interrupção do relógio para garantir o perfeito funcionamento do sistema (caso já haja um desvio anterior cuja execução deve continuar ocorrendo). Também é necessário inicializar a controladora de interrupções. Isto é feito enviando o valor 0x20 para a porta 0x20;
- quando for necessário encerrar a execução do miniSO, basta definir a rotina de atendimento da interrupção do relógio como sendo aquela obtida com a função `getvect()`, no primeiro passo.

4.5 Chaveamento de processos leves

Um dos pontos fundamentais para se ter vários processos sendo executados ao mesmo tempo é que, além do salvamento e recuperação de registradores, cada processo deve ter a sua própria pilha. Se cada processo leve não tivesse a sua própria pilha, as informações que são salvas na pilha em uma chamada de subrotina poderiam ser destruídas ou poderiam se misturar com as informações de outro processo. Portanto, tanto para processos quanto para processos leves, ter uma pilha própria é indispensável.

A pilha de cada processo leve também será utilizada no chaveamento de processos. Quando um processo leve está em execução, os registradores SS e SP apontam para a sua pilha. No momento em que ocorre uma interrupção, automaticamente todos os registradores serão salvos na pilha do processo leve e a rotina de atendimento da interrupção do relógio será executada. Para passar a execução para outro processo leve bastará fazer com que os registradores SS e SP apontem para a pilha de outro processo leve. Ao fim da interrupção, todos os registradores do novo processo leve serão desempilhados e a execução continuará neste

processo leve, até que uma nova interrupção do relógio seja gerada. A Figura 15 ilustra estes passos. Nesta figura é possível visualizar como a execução é passada de um processo leve para outro.

Na Figura 15a, o processo leve 1 está sendo executado e acontece uma interrupção. Automaticamente os registradores de estados, CS e IP são salvos na pilha e a rotina de atendimento da interrupção do relógio (núcleo do miniSO) é executada (Figura 15b). O primeiro passo do núcleo é salvar todos os demais registradores do processo leve na pilha (Figura 15c). Em seguida, o núcleo faz os registradores SS e SP apontarem para a pilha de um novo processo leve (Figura 15d). A escolha do novo processo leve é executada com a ajuda de uma lista de processos prontos. Quando a rotina de atendimento da interrupção encerra a sua execução ela desempilha o primeiro bloco de registradores (Figura 15e). Por fim, quando os registradores de estados, CS e IP são desempilhados, a execução passa para outro processo leve (Figura 15f).

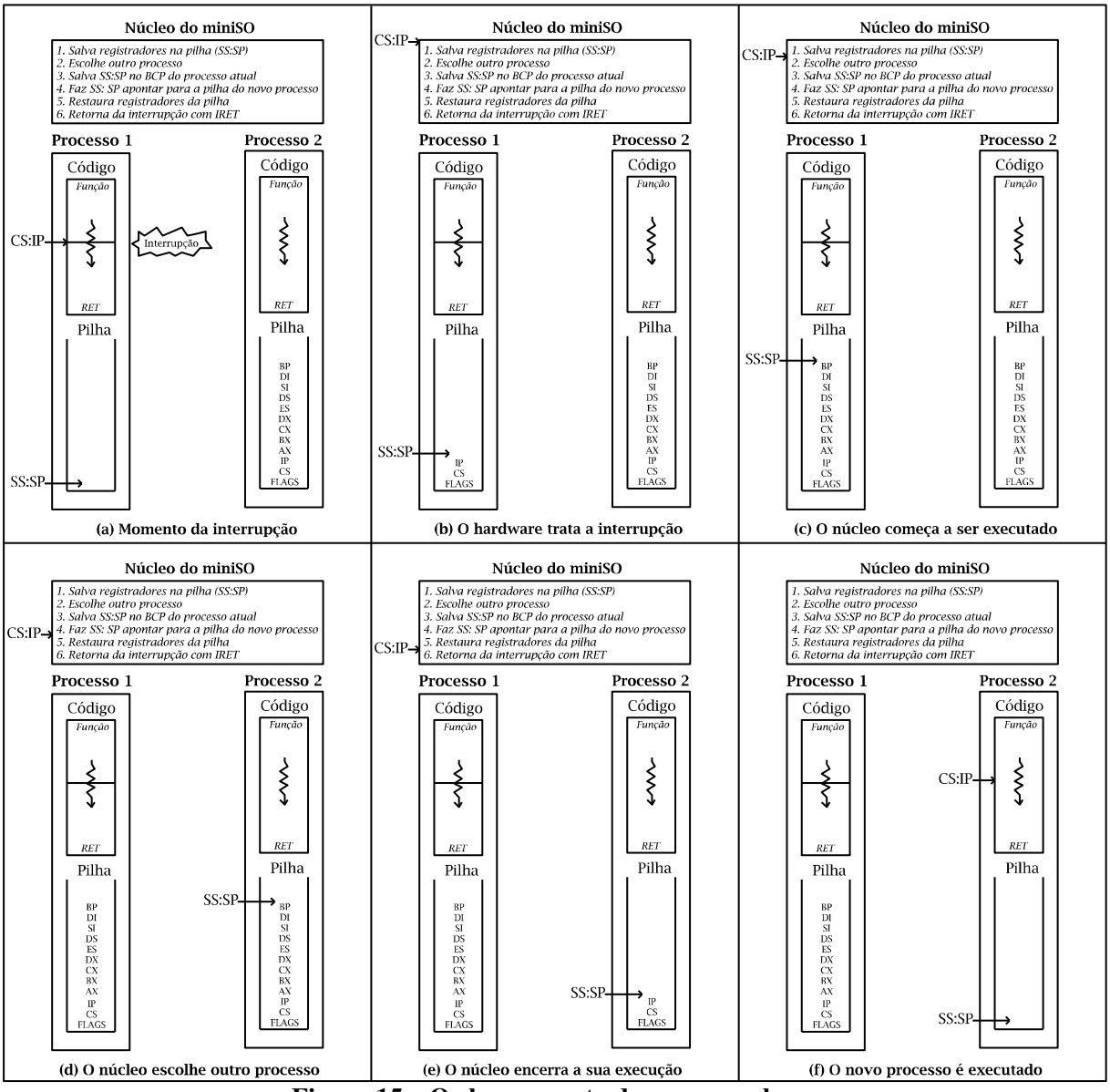

Figura 15 – O chaveamento de processos leves

5 IMPLEMENTAÇÃO DO CONCEITO DE PROCESSO LEVE

Para promover uma compreensão mais detalhada da implementação de processos leves, são discutidos a seguir: as estruturas de dados, os estados de processos, as listas de processos, as chamadas disponíveis para gerência de processos e o diagrama de transição de estados de processos.

5.1 Estruturas de dados para gerência de processos

A implementação de processos leves no miniSO é uma evolução da implementação de um núcleo de concorrência para o sistema operacional MS-DOS, descrito por Teodorowitsch (1999).

Para permitir a execução concorrente de processos, inicialmente, é necessário definir um Bloco de Controle de Processos (BCP), ou seja, um descritor de processo. Este BCP mantém as informações necessárias para que o miniSO possa executar o chaveamento de processos leves. Os BCPs de todos os processos leves são mantidos em um vetor alocado de forma estática. Desta forma, para que o miniSO suporte um número maior de processos leves do que o padrão, é necessário aumentar o tamanho deste vetor. Será necessário gerenciar diversas listas de processos sobre este vetor de BCPs.

Para tornar o código mais claro, foram definidos alguns tipos básicos que serão usados na definição do BCP. Estes tipos estão listados na Figura 16. São eles: `pid_t`, tipo compatível com `int`, para armazenar números de identificação de processos; `pcb_t`, tipo compatível com `int`, para armazenar números de índices da tabela de processos (será usado para implementar listas); `signal_t`, tipo compatível com `unsigned`, para campos relacionados com sinais recebidos e esperados pelo processo.

```
typedef int      pid_t;
typedef int      pcb_t;
typedef unsigned signal_t;
```

Figura 16 – Tipos internos do miniSO

A Figura 17 mostra o BCP definido para os processos do miniSO. Esta definição faz parte do arquivo `miniSO.h`.

```
typedef struct {
  pid_t    pid;
  pid_t    ppid;
  int      status;
  unsigned ss;
  unsigned sp;
  signal_t recvsig;
  signal_t waitsig;
  pcb_t    wait;
  pid_t    waitfor;
  int      exitcode;
  pcb_t    zombies;
  pcb_t    prev;
  pcb_t    next;
} miniSO_PCB;
```

Figura 17 – Definição do BCP do miniSO

Os principais campos de um BCP, tal como mostrado na Figura 17, são:

- número de identificação do processo leve (campo `pid`): é um número único que identifica o processo (não podem haver dois processos com o mesmo número);
- número de identificação do processo-pai (campo `ppid`): é o PID do processo que criou o processo atual;
- estado do processo leve (campo `status`): pode assumir algum dos seguintes valores `FREE`, `READY`, `RUNNING`, `ZOMBIE`, `WAIT` ou `WAITSIG` ou `WAITSEM`.
- registrador de segmento da pilha do processo (campo `ss`): segmento onde está a pilha do processo;
- registrador de deslocamento da pilha do processo (campo `sp`): deslocamento do segmento onde está a pilha do processo;
- sinais recebidos pelo processo (campo `recvsig`): mapa de *bits* correspondente aos sinais que o processo já recebeu. Os dezesseis bits deste campo são utilizados para identificar cada um dos dezesseis diferentes sinais que um processo pode receber. É possível fazer composições com sinais;
- sinais esperados pelo processo (campo `waitsig`): mapa de *bits* correspondente aos sinais pelos quais o processo deseja esperar. Caso o processo ainda não tenha recebido todos os sinais (campo `recvsig`) ela deverá aguardar em uma lista de espera por sinais;

- BCP do processo-pai esperando pelo fim de execução do processo (campo `wait`);
- número de BCP do processo pelo qual se está esperando (campo `waitfor`): este campo é definido pela função `waitpid()`, quando um processo deseja esperar por outro. Ele é utilizado pela função `kill()` para facilitar a exclusão de um processo, quando ele se encontra nesta lista de espera;
- resultado do final de execução de um processo (campo `exitcode`): quando um processo tem sua execução encerrada com `exit()` ou `kill()`, o código de fim do processo terá que ser passado para o processo que executar `wait()` ou `waitpid()`. Esta transferência é feita através desta variável;
- lista de processos-filho em estado zumbi (campo `zombies`): contém uma lista duplamente encadeada e não circular com todos os processos-filho do processo atual, ou seja, que já encerraram a sua execução, porém para os quais o processo atual (pai) ainda não executou `wait()`; ou `waitpid()`;
- "endereço" do processo anterior (campo `prev`): corresponde ao índice do processo anterior da lista à qual o processo em questão pertence. Este campo só é utilizado quando a lista é duplamente encadeada;
- "endereço" do próximo processo (campo `next`): corresponde ao índice do próximo processo da lista à qual o processo em questão pertence.

O vetor correspondente à tabela de processos é declarado da seguinte forma:

```
miniSO_PCB miniSO_thread[miniSO_MAXTHREADS];
```

onde a constante `miniSO_MAXTHREADS` (declarada no arquivo `miniSO.h`) contém, a princípio, o valor 16.

5.2 Estados

Os estados pelos quais um processo ou BCP pode passar no miniSO são:
- `FREE`: corresponde a um BCP não utilizado – processos neste estado são mantidos na lista `miniSO_free`, que é simplesmente encadeada e não circular;
- `READY`: trata-se de um processo que está na lista de prontos, esperando para receber o processador – processos neste estado são mantidos na lista `miniSO_ready`, que é duplamente encadeada e circular;
- `RUNNING`: trata-se do processo que está sendo executado – este processo é o primeiro processo da lista `miniSO_ready`;
- `ZOMBIE`: o processo já encerrou a sua execução, porém o seu processo-pai ainda não executou `wait()` ou `waitpid()` – processos neste estado são mantidos na lista que corresponde ao campo `zombies` do BCP do processo-pai (que é duplamente encadeada e não circular);
- `WAIT`: trata-se de um processo que está esperando pelo final da execução de um de seus processos-filho – processos neste estado são mantidos no campo `wait` do BCP do processo pelo qual estão esperando;
- `WAITSIG`: trata-se de um processo que está esperando por sinais – processos neste estado não permanecem em nenhuma lista;
- `WAITSEM`: trata-se de um processo que está esperando pela liberação de um semáforo – processos neste estado são mantidos na lista de espera do respectivo semáforo.

5.3 Listas de processos e BCPs do miniSO

As listas de processos e BCPs gerenciadas pelo miniSO são as seguintes:
- lista de BCPs livres (`miniSO_free`): trata-se de uma lista simplesmente encadeada (não circular) que contém todos os BCPs que não estão sendo utilizados no momento, ou seja, cujo estado é igual a `FREE`;
- lista de prontos (`miniSO_ready`): é uma lista circular e duplamente encadeada que contém o BCP do processo que está em execução (estado `RUNNING`), e que é o primeiro processo da lista, e também de todos os processos que estão prontos para executar, ou seja, cujo estado é igual a `READY`;
- lista de processos-filho em estado zumbi (campo `zombies` do BCP do processo-pai): é uma lista duplamente encadeada (não circular) e contém todos os processos-filho do processo em questão

(pai) que já encerraram a sua execução, porém para os quais o processo pai ainda não executou `wait()` ou `waitpid()`.

Além destas listas há ainda os estados de espera por sinais (`WAITSIG`) e pelo fim da execução de um processo-filho (`WAIT`). Este estado último estado é utilizado para tratar a situação em que um processo-pai executa `wait()` ou `waitpid()`, porém ainda não há nenhum filho em estado zumbi.

5.4 Chamadas para gerência de processos

Os serviços para gerência de processos no miniSO estão disponíveis em dois níveis:

- chamadas de biblioteca: que devem ser usadas na implementação das aplicações e que simplesmente repassam os parâmetros recebidos para as respectivas chamadas de sistema; e
- chamadas de sistema: que efetivamente executam o serviço.

5.4.1 Chamadas de biblioteca

Na biblioteca padrão do miniSO, arquivo `lib.c`, há 8 chamadas para gerenciar processos. O arquivo `lib.h` contém a definição dos protótipos destas funções. As funções disponíveis são:

- `fork()`: inicia um processo leve;
- `kill()`: encerra um processo leve especificado;
- `wait()`: espera pelo fim de um processo leve (filho do processo atual);
- `waitpid()`: espera pelo fim de um processo leve (filho do processo atual) especificado;
- `exit()`: encerra o processo leve corrente;
- `getpid()`: retorna o identificador de processo (PID) do processo leve corrente;
- `getppid()`: retorna o PID do processo que criou o processo leve corrente;
- `sendsignal()`: envia um sinal para o processo leve especificado;
- `waitsignal()`: faz com que o processo leve corrente espere por um sinal especificado.

O Quadro 2 apresenta uma descrição mais detalhada de cada uma destas funções.

Quadro 2 – Funções da biblioteca do miniSO para gerência de processos

Função / Protótipo	Chamadas de Sistema utilizadas	Descrição
`pid_t fork (void (*fun)());`	`SC_FORK`	Inicia um processo leve na função `fun`, retornado o identificador de processo (PID) no novo processo. Caso ocorra algum erro, será retornado o valor `miniSO_ERROR`.
`int kill (pid_t pid);`	`SC_KILL`	Encerra o processo leve especificado no parâmetro `pid`. Retorna `miniSO_OK` em caso de sucesso, ou `miniSO_ERROR` em caso de erro.
`pid_t wait (int far *status);`	`SC_WAIT`	Faz com que o processo leve corrente espere pelo final de um processo-filho. Retorna o PID do processo ou `miniSO_ERROR` (em caso de erro). O estado de fim do processo-filho é passado por referência para o parâmetro `status`.
`pid_t waitpid (pid_t pid,int far *status);`	`SC_WAITPID`	Faz com que o processo leve corrente espere pelo final de um processo-filho específico (`pid`). Retorna o PID do processo ou `miniSO_ERROR` (em caso de erro). O estado de fim do processo-filho é passado por referência para o parâmetro `status`.
`void exit (int codfim);`	`SC_EXIT`	Encerra o processo leve corrente com código de fim igual a `codfim`. Nunca retorna.
`pid_t getpid (void);`	`SC_GETPID`	Retorna o identificador de processo (PID) do processo corrente.
`pid_t getppid (void);`	`SC_GETPPID`	Retorna o identificador de processo (PID) do processo que criou o processo corrente.
`int sendsignal (pid_t pid, signal_t signal);`	`SC_SENDSIGNAL`	Envia o sinal `signal` para o processo leve especificado em `pid`. Retorna `miniSO_OK` em caso de sucesso ou `miniSO_ERROR` em caso de erro.
`void waitsignal (signal_t signal);`	`SC_WAITSIGNAL`	Faz com que o processo corrente espere por um sinal especificado em `signal`. Não retorna valor nenhum.

5.4.2 Chamadas de sistema

O ponto de entrada para o código que executa as chamadas de sistema, ou seja, a rotina que atende a interrupção 0x22 (34), é a função `_miniSO_systemcall`, em `miniSO.asm`. Esta função chama a função `scall()`, em `scall.c`, que por sua vez se encarrega de executar a função responsável pela respectiva chamada de sistema, conforme definido na tabela `miniSO_scall_table`. Esta tabela armazena para cada serviço: o número (que deve ser colocado no registrador AH pelas chamadas de biblioteca), o número de parâmetros que o serviço recebe e a função que executará o serviço. Todas as funções que executam serviços tem o prefixo `sc_`, e estão implementadas em C, no arquivo `scall.c`.

As chamadas de sistema ou serviços previstos para gerência de processos são:

- SC_FORK: inicia um novo processo leve;
- SC_KILL: encerra um processo leve especificado;
- SC_WAIT: espera pelo final de execução de um processo-filho qualquer;
- SC_WAITPID: espera pelo final de execução de um processo-filho específico;
- SC_EXIT: encerra o processo leve corrente, fornecendo um código de fim de execução;
- SC_GETPID: retorna o identificador (PID – *Process IDentification*) do processo leve corrente;
- SC_GETPPID: retorna o identificador (PID) do processo-pai do processo leve corrente;
- SC_SENDSIGNAL: envia um sinal para determinado processo leve;
- SC_WAITSIGNAL: faz o processo leve corrente esperar pela chegada de determinado sinal.

O Quadro 3 apresenta um descrição detalhada destas chamadas de sistema.

Quadro 3 – Descrição das chamadas de sistema do miniSO para gerência de processos

Nome/Código em AH	Outros valor(es) recebido(s)	Valor(es) retornado(s)	Descrição
`SC_FORK`	BX = segmento CX = deslocamento	AX = PID ou código de erro	pid_t sc_fork (unsigned cs,unsigned ip); Cadastra o trecho de código apontado por CS e IP como um processo leve, retornando o seu número de identificação (PID) ou um código de erro (`miniSO_ERROR`). Esta função verifica inicialmente se há BCPs disponíveis. Se não houver, retorna um erro. A seguir: desabilita as interrupções, gera um PID único para o novo processo leve, retira um BCP da lista de BCPs livres, faz um ponteiro apontar para a pilha deste processo e inicializa a pilha do novo processo, preenche os outros campos do BCP para o novo processo, coloca o processo novo no final da lista de prontos e habilita as interrupções. Por fim, retorna o PID do novo processo leve.
`SC_KILL`	BX = PID	AX = código de sucesso	int sc_kill (pid_t pid); Encerra a execução de um processo leve especificado, retornando um código de sucesso (`miniSO_OK`) ou erro (`miniSO_ERROR`). Inicialmente esta função desabilita as interrupções e verifica se existe um processo com o número especificado. Se não houver, habilita as interrupções e retorna um código de erro. Em seguida, é necessário verificar se o processo-pai está esperando (em uma chamada `wait()` ou `waitpid()`) pelo processo que está sendo excluído. Isto é feito consultando o campo `wait`, que existe no BCP de cada processo. Se estiver, o processo-pai é recolocado na lista de prontos, sendo que o campo `exitcode` do pai é inicializado com `miniSO_ERROR` (o que corresponde ao código de fim do processo-filho). Em seguida o BCP do processo é excluído da lista em que ele se encontra e colocado na lista de BCPs livres. Por fim, habilitam-se as interrupções e se o processo leve corrente se auto-excluiu, espera-se em um laço infinito. Caso contrário, retorna-se o código de sucesso `miniSO_OK`.
`SC_WAIT`	BX = segmento de `status` CX = deslocamento de `status`	AX = PID do filho que encerrou a execução	int sc_wait (unsigned seg,unsigned off); Faz com que o processo corrente espere pelo final da execução de um de seus processos-filho. Inicialmente desabilitam-se as interrupções e verifica-se se o processo correne tem processos em estado zumbi (lista correspondente ao campo `zombies`). Se não houver nenhum filho em estado zumbi, o processo corrente ficará bloqueado no estado `WAIT`. Quando o processo corrente tiver filhos em estado zumbi ou quando ele for despertado posteriormente do estado `WAIT`, o primeiro processo da lista de zumbis será inserido na lista de BCPs livres. Antes de retornar, atualiza-se o código de fim do processo-filho, por referência, no campo `status`.
`SC_WAITPID`	BX = PID CX = segmento de `status` DX = deslocamento de `status`	AX = PID do filho que encerrou a execução	int sc_waitpid (pid_t pid,unsigned seg,unsigned off); Faz com que o processo corrente espere pelo final da execução de um processo-filho específico. Inicialmente desabilitam-se as interrupções e verifica-se se o PID do processo pelo qual se quer esperar existe e se ele é um processo-filho do processo atual. Caso não exista ou não seja o PID de um processo-filho, retorna-se um código de erro. Se o processo-filho estiver no estado zumbi, ele é removido da lista de zumbis, inserido na lista de BCPs livres e, antes de retornar, atualiza-se o estado do processo-filho, por referência, através do campo `status`. Caso o processo-filho ainda esteja em execução, o processo-pai entra em estado de espera (WAIT). Quando o processo for despertado posteriormente do estado `WAIT`, antes de retornar, atualiza-se o código de fim do processo-filho, por referência, no campo `status`.
`SC_EXIT`	BX = código de fim	Nenhum	void sc_exit (int codfim); Esta função encerra a execução do processo leve corrente, fornecendo um código de fim, que poderá ser obtido pelo processo-pai através de uma chamada `wait()` ou `waitpid()`. Inicialmente, desabilitam-se as interrupções. A seguir, é necessário verificar se o processo-pai está esperando pelo processo que será encerrado em uma chamada `wait()` ou `waitpid()`. Isto é feito consultando o campo `wait`, que existe no BCP de cada processo. Se estiver, o processo-pai é recolocado na lista de prontos, sendo que o campo `exitcode` do pai é inicializado com o valor do parâmetro `codfim`. Em seguida, deve-se verificar se o processo corrente não é o processo 0 e se o processo corrente não é o último. Se for, deve-se imprimir uma mensagem e reinicializar o sistema. O próximo passo é remover o BCP do processo corrente da lista de prontos, colocando-o na lista de BCPs livres. Por fim, habilitam-se as interrupções e espera-se em um laço infinito.
`SC_GETPID`	Nenhum	AX = PID	pid_t sc_getpid (void); Retorna o número de identificação do processo corrente. Simplesmente retorna o valor do campo `pid` do BCP do processo corrente.
`SC_GETPPID`	Nenhum	AX = PID	pid_t sc_getppid (void); Retorna o número de identificação do processo-pai do processo corrente. Simplesmente retorna o valor do campo `ppid` do BCP do processo corrente.
`SC_SENDSIGNAL`	BX = PID CX = sinal	AX = código de sucesso	int sc_sendsignal (int pid, unsigned signal); Esta função envia um sinal para determinado processo. Inicialmente, desabilitam-se as interrupções e verifica-se se existe um processo com o número especificado. Se não existir, retorna-se um código de erro. Caso contrário, atualiza-se o campo de sinais recebidos do processo destino e, se o processo destino estava esperando por sinais, verifica-se se o sinal recebido é suficiente para desbloqueá-lo. Se for, retira-se ele da lista de espera por sinais. recolocando-o na lista de prontos e atualizando os campos necessários no BCP do processo. Por fim, habilitam-se as interrupções e retorna-se o código de sucesso (`miniSO_OK`).
`SC_WAITSIGNAL`	CX = sinal	Nenhum	void sc_waitsignal (unsigned signal); Esta função faz com que o processo corrente espere pela chegada de um sinal. Inicialmente desabilitam-se as interrupções e verifica-se se o processo já não recebeu os sinais pelos quais deseja esperar. Se recebeu, atualizam-se os campos de sinais recebidos (`recvsig`) e sinais esperados (`waitsig`). Caso os sinais recebidos não sejam suficientes para desbloquear o processo, ele deverá ser retirado da lista de prontos e colocado na lista `miniSO_waitsig`. Por fim, habilitam-se as interrupções e se o processo corrente tiver sido colocado na lista de espera por sinais, espera-se em um laço vazio, testando se o estado do processo corrente é igual a `miniSO_WAITSIG`.

5.5 Diagrama de transição de estados de processos

O Quadro 4 mostra a descrição das listas do miniSO, bem como o estado correspondente a cada lista e as chamadas que executam a inserção e remoção de processos nestas listas. As inserções e remoções de processos indicadas nestas listas (ou estados) podem ser utilizadas para montar o diagrama de transição de estados de processos, tal como mostrado na Figura 18. Um arco de um estado para outro na Figura 18 corresponderá a uma exclusão de lista e a uma inserção em lista, no Quadro 4.

Quadro 4 – Relacionamento entre listas, estados e chamadas do miniSO

Lista	Tipo	Estado(s)	Inserção	Exclusão
`miniSO_free`	simplesmente encadeada e não circular	`FREE`	`wait(L)`	`fork(A)`

`miniSO_ready`	duplamente encandeada e circular (primeiro da lista em `RUNNING`, demais em `READY`)	`RUNNING`	`miniSO_clockhandler(K)`	`exit(C)`
				`kill(C)`
				`waitsignal(D)`
				`wait(G)`
				`miniSO_clockhandler(J)`

		`READY`	`fork(A)`
			`sendsignal(E)`
			`kill(H)`
			`exit(H)`
			`miniSO_clockhandler(J)`
				`kill(B)`
				`miniSO_clockhandler(K)`

`miniSO_waitsig`	duplamente encadeada e não circular	`WAITSIG`	`waitsignal(D)`	`sendsignal(E)` `kill(F)`
campo `wait` do BCP	elemento simples	`WAIT`	`wait(G)`	`kill(H,I)` `exit(H)`
campo `zombies` do BCP	duplamente encadeadas e não circular	`ZOMBIE`	`kill(B,C,F,I)` `exit(C)`	`wait(L)`

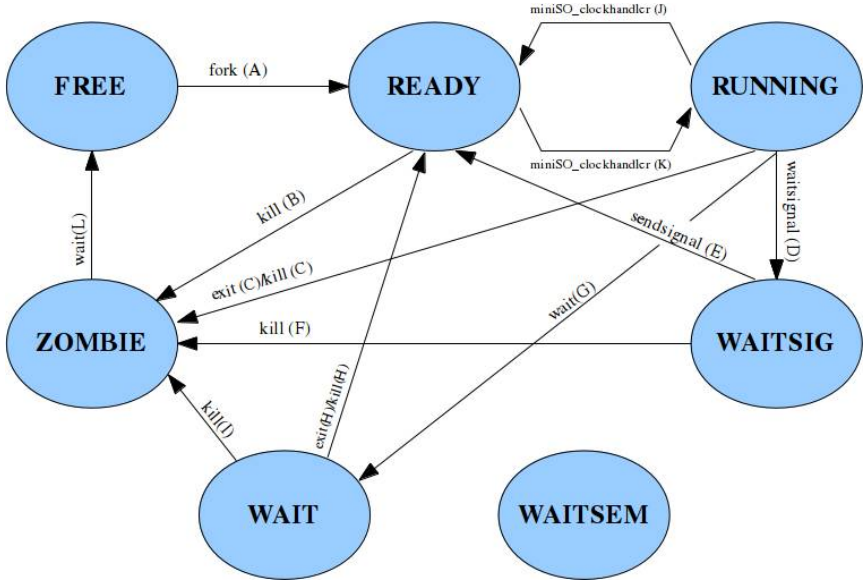

Figura 18 – Diagrama de transição de estados de processos do miniSO

Com relação ao diagrama de transição de estados de processos, mostrado na Figura 18, é importante perceber a diferença entre a transição `kill(I)` e `kill(H)`. `kill(I)` ocorre quando um processo em execução executa esta chamada sobre um processo que está aguardando pelo final da execução de seu processo-filho, enquanto `kill(H)` ocorre quando um processo em execução executa esta chamada sobre o próprio processo filho. No último caso, o processo-filho é encerrado de forma forçada e o processo-pai deve ser recolocado na lista de prontos.

6 Implementação de código para o miniSO

O miniSO é um sistema operacional de pequeno porte desenvolvido para ser utilizado em projetos didáticos. Apesar possuir um número restrito de funcionalidades, o miniSO utiliza os mesmos conceitos que a maioria dos sistemas operacionais modernos. A ampliação de serviços deste pequeno sistema operacional é uma atividade que permite que os alunos de disciplinas de Sistemas Operacionais possam desenvolver trabalhos práticos sem se sentirem desestimulados com o grande tamanho do código-fonte dos sistemas operacionais usados em ambientes de produção.

As etapas básicas para acrescentar uma nova funcionalidade a um sistema operacional podem envolver os seguintes níveis:

- acrescentar um novo comando ou desenvolver uma aplicação que utilize alguma funcionalidade;
- implementar uma nova rotina em uma biblioteca para acessar alguma funcionalidade;
- criar uma nova chamada de sistema no núcleo do sistema operacional que implemente alguma nova funcionalidade.

Na sua versão atual, os comandos do miniSO são comandos internos, que fazem parte do próprio interpretador de comandos. O interpretador de comandos executa no nível de aplicação. Processos nesse nível são normalmente implementados em linguagem de alto nível e usam, portanto, rotinas de bibliotecas na sua implementação.

Bibliotecas não são, em geral, uma responsabilidade do sistema operacional. Ou seja, quem projeta o sistema operacional não precisa necessariamente se preocupar com as bibliotecas que serão disponibilizadas para este sistema operacional. No entanto, no caso específico do miniSO, como não há comandos externos e as aplicações são embutidas na própria imagem do sistema operacional, foi desenvolvida uma biblioteca básica para que se possa desenvolver comandos e programas em linguagem de alto nível.

Por sua vez, acrescentar chamadas de sistema a um sistema operacional envolve uma modificação no núcleo deste sistema.

O restante desta seção descreve as modificações que devem ser feitas no código-fonte do miniSO para acrescentar um comando interno, uma chamada de biblioteca e uma chamada de sistema, com o objetivo de permitir que se possa suspender a execução de um processo e mais tarde continuar a sua execução. As modificações são feitas em etapas, de forma que se possa testar o miniSO após cada etapa, verificando se a alteração está funcionando adequadamente.

As etapas apresentadas nas subseções seguintes consistem em:
- alterações necessárias para a inclusão de dois novos comandos no miniSO;
- implementação das rotinas de biblioteca;
- implementação das chamadas de sistema;
- realização de outras alterações que deverão ser feitas para dar suporte às rotinas para suspensão e reinício de processos.

6.1 Implementação comandos internos

Serão acrescentados dois comandos internos ao miniSO. Estes comandos serão chamados: `stop` e `resume`. Ambos receberão como parâmetro o número do processo que será, respectivamente, suspenso (comando `stop`) ou reiniciado (comando `resume`).

Inicialmente, para acrescentar um comando simples ao miniSO deve-se modificar o valor da constante `MAXCOMMANDS`, definida em `command.h`, aumentando o seu valor em uma unidade para cada novo comando. Na versão atual o valor será aumentado de 23 para 25. Também deve-se acrescentar ao final deste arquivo os protótipos das funções que implementarão os comandos internos. A Figura 19 mostra, em destaque, as alterações que devem ser feitas no arquivo `command.h`.

```
[...]
/* Número máximo de comandos */
#define MAXCOMMANDS 25
[...]
int    cmd_semdestroy (int argc, char far *argv[]);
int    cmd_stop       (int argc, char far *argv[]);
int    cmd_resume     (int argc, char far *argv[]);
```

Figura 19 – Alterações no arquivo `command.h`

O próximo passo é preparar o interpretador de comandos para reconhecer os novos comandos e fornecer o seu código. Inicialmente deve-se acrescentar a definição do comando na inicialização do vetor `commands` em `command.c`. A Figura 20 mostra a versão atual e a Figura 21 mostra a versão alterada. No final de `command.c`, pode-se acrescentar um código provisório para os novos comandos, conforme mostra a Figura 22, apenas para testar se o reconhecimento do novo comando está funcionando adequadamente.

```
static command_t commands[MAXCOMMANDS] = {
  [...]
  "semdestroy"," <semid>  destroi um semaforo",                                  cmd_semdestroy
};
```

Figura 20 – Trecho da inicialização do vetor `commands` em `command.c`

```
static command_t commands[MAXCOMMANDS] = {
  [...]
  "semdestroy"," <semid>  destroi um semaforo",                                  cmd_semdestroy,
  "stop"," <pid>          suspende um processo/thread",                          cmd_stop,
  "resume"," <pid>        reinicia um processo/thread",                          cmd_resume
};
```

Figura 21 – Trecho alterado da inicialização do vetor `commands` em `command.c`

```
int cmd_stop(int argc, char far *argv[])
{
       putstr("Comando stop:\n");
       putstr("- opcao: ");
       if      (argc>=2)
              putstr(argv[1]);
       putch('\n');
       return 0;
}

int cmd_resume(int argc, char far *argv[])
{
       putstr("Comando resume:\n");
       putstr("- opcao: ");
       if      (argc>=2)
              putstr(argv[1]);
       putch('\n');
       return 0;
}
```

Figura 22 – Código provisório dos novos comandos no final de `command.c`

As versões definitivas dos comandos `stop` e `resume` são apresentadas na Figura 23. Estas versões exigem que se implemente as funções `stop()` e `resume()` na biblioteca do miniSO, o que será descrito na próxima subseção. Ambas as funções recebem como argumento o número do processo sobre o qual a operação deve ser aplicada, retornando o valor `miniSO_OK`, em caso de sucesso, ou `miniSO_ERROR`, em caso de erro.

```
int cmd_stop(int argc, char far *argv[])
{
	int t=0;
	pid_t pid=0;

	if	(argc<2) {
		putstr("stop: nenhum parametro foi fornecido\n");
		return miniSO_ERROR;
	}
	pid = atoi(argv[1]);
	t = stop(pid);
	if	(t==miniSO_ERROR)
		putstr("stop: impossivel suspender a thread\n");
	return t;
}


int cmd_resume(int argc, char far *argv[])
{
	int t=0;
	pid_t pid=0;

	if	(argc<2) {
		putstr("resume: nenhum parametro foi fornecido\n");
		return miniSO_ERROR;
	}
	pid = atoi(argv[1]);
	t = resume(pid);
	if	(t==miniSO_ERROR)
		putstr("resume: impossivel reiniciar a thread\n");
	return t;
}
```

Figura 23 – Código final dos novos comandos no final de `command.c`

6.2 Implementação rotinas de biblioteca

A biblioteca do miniSO corresponde a um conjunto de rotinas cujo principal objetivo é facilitar o desenvolvimento de aplicações em linguagem de alto nível. O código desta biblioteca está descrito nos arquivos `lib.c` e `lib.h`.

Para implementar uma rotina para a biblioteca do miniSO basta colocar o seu código em `lib.c` e o seu protótipo em `lib.h`. Conforme especificado na seção anterior, as funções `stop()` e `resume()` receberão como argumento um número de processo. E retornarão `miniSO_OK`, em caso de sucesso, ou `miniSO_ERROR`, em caso de erro. A Figura 24 mostra os protótipos das funções, que devem ser incluídos em `lib.h`.

```
extern int		stop		(pid_t pid);
extern int		resume		(pid_t pid);
```

Figura 24 – Protótipo das funções a serem incluídos em `lib.h`

O próximo passo será fornecer o código da função. Inicialmente, para efeito de teste, recomenda-se o desenvolvimento de um código provisório para testar se os parâmetros estão sendo corretamente recebidos. Este código pode ser, por exemplo, o código da Figura 25, que simplesmente converte os valores inteiros recebidos, exibindo-os.

```
int stop (pid_t pid)
{
       char str[10];

       inttostr(str,pid);
       putstr("PID=");
       putstr(str);
       putch('\n');
       return 0;
}

int resume (pid_t pid)
{
       char str[10];

       inttostr(str,pid);
       putstr("PID=");
       putstr(str);
       putch('\n');
       return 0;
}
```

Figura 25 – Código provisório das funções da biblioteca no final de `lib.c`

O código definitivo para as funções exige a implementação das respectivas chamadas de sistema, passo que será descrito na próxima subseção. Estas chamadas de sistema terão como números de serviço os valores definidos nas constantes `SC_STOP` e `SC_RESUME`. Para executar uma chamada de sistema o número do serviço deve ser colocado no registrador AH. No caso das duas chamadas de sistema em questão, o valor do identificador de processo (PID) é colocado em BX. A execução da chamada de sistema é feita através da instrução de interrupção de software, vetor de interrupção 22h. O código definitivo para as funções de biblioteca está descrito na Figura 26.

```
int stop(pid_t pid)
{
       _BX = pid;
       asm {
              mov ah,SC_STOP
              int 22h
       }
       return _AX;
}

int resume(pid_t pid)
{
       _BX = pid;
       asm {
              mov ah,SC_RESUME
              int 22h
       }
       return _AX;
}
```

Figura 26 – Código definitivo das funções de biblioteca no final de `lib.c`

6.3 Implementação de chamadas de sistema

O código que implementa as chamadas de sistema do miniSO está descrito em três arquivos:

- `miniSO.asm`: contém o ponto de entrada das chamadas de sistema (que redireciona a execução para uma rotina em C no arquivo `scall.c`) e rotinas internas do miniSO;
- `scall.c`: contém a implementação das chamadas de sistema.

O código do arquivo `miniSO.asm`, que contém o ponto de entrada das chamadas de sistema, simplesmente empilha os registradores que podem conter parâmetros para chamadas de sistema, para que estes valores sejam recebidos como argumentos de função em C, e chama a função `scall()` em `scall.c`. Os valores dos registradores devem ser empilhados na mesma posição em que a função em C espera encontrá-los na pilha.

Para criar uma chamada de sistema deve-se, em `scall.h`:

- incrementar a constante `miniSO_NUMSCALL`, que contém o número total de chamadas de sistema;
- definir constantes inteiras que identifiquem de forma única cada chamada de sistema (por exemplo, `SC_STOP` e `SC_RESUME`); e
- apresentar o protótipo das funções que implementarão as chamadas de sistema (por exemplo, `sc_stop()` e `sc_resume()`).

A Figura 27 mostra em destaque as alterações a serem feitas em `scall.h`.

```
[...]
#define miniSO_NUMSCALL 27

#define SC_PUTCH        0
[...]
#define SC_SEMDESTROY        24
#define SC_STOP              25
#define SC_RESUME            26
[...]
int    sc_semdestroy  (semid_t s);
int    sc_stop        (pid_t pid);
int    sc_resume      (pid_t pid);
```

Figura 27 – Alterações em `scall.h`

A seguir deve-se modificar o arquivo `scall.c`, atualizando a tabela de chamadas de sistema e fornecendo o código das funções que implementam as novas chamadas de sistema. A alteração da tabela de chamadas de sistema (`miniSO_scall_table`) é mostrada na Figura 28.

```
scall_t miniSO_scall_table[miniSO_NUMSCALL] = {
       { SC_PUTCH,           1, (scfun0_t)sc_putch         },
[...]
       { SC_SEMDESTROY,      1, (scfun0_t)sc_semdestroy    },
       { SC_STOP,            1, (scfun0_t)sc_stop          },
       { SC_RESUME,          1, (scfun0_t)sc_resume        }
};
```

Figura 28 – Alteração da tabela de chamadas de sistema

A Figura 29 apresenta uma versão provisória do código das funções que implementam as novas chamadas de sistema. Recomenda-se acrescentar este código no final de `scall.c`.

```
int sc_stop(pid_t pid)
{
       char str[10];

       disable();
       inttostr(str,pid);
       putstr("PID=");
       putstr(str);
       putch('\n');
       enable();
       return miniSO_OK;
}


int sc_resume(pid_t pid)
{
       char str[10];

       disable();
       inttostr(str,pid);
       putstr("PID=");
       putstr(str);
       putch('\n');
       enable();
       return miniSO_OK;
}
```

Figura 29 – Código provisório das funções que implementam as novas chamadas de sistema

6.4 Outras alterações

Para implementação completa das chamadas de sistema que suspendem e reiniciam processos, também é necessário:

- em `miniSO.h`, definir uma constante `STOPPED` (com valor 6), para representar o estado dos processos suspensos, e uma constante `strSTOPPED`, com a cadeia de caracteres "STOPPED" – a Figura 30 mostra um trecho de `miniSO.h` com estas alterações em destaque;

```
[...]
/* Estados de um processo */
#define FREE       -1     /* O BCP esta' na lista de livres */
#define READY       0     /* O processo esta' na lista de prontos */
#define RUNNING     1     /* O processo e' o primeiro da lista de prontos */
#define ZOMBIE      2     /* O processo esta' no estado "zumbi" */
#define WAIT        3     /* O processo esta' esperando por algum filho */
#define WAITSIG     4     /* O processo esta' na lista de espera por sinais */
#define WAITSEM     5     /* O processo esta' esperando por semaforos */ /*S*/
#define STOPPED     6     /* O processo esta' suspenso */

/* Strings de estados de processos */
#define strFREE    "FREE   "
#define strREADY   "READY  "
#define strRUNNING "RUNNING"
#define strZOMBIE  "ZOMBIE "
#define strWAIT    "WAIT   "
#define strWAITSIG "WAITSIG"
#define strWAITSEM "WAITSEM"
#define strSTOPPED "STOPPED"
[...]
```

Figura 30 – Alterações em `miniSO.h`

- modificar o comando ps (implementado pela função `cmd_ps()` em `command.c`) para que ele exiba a cadeia de caracteres "STOPPED", correspondente ao estado `STOPPED` – a Figura 31 mostra a implementação de `cmd_ps()` com a alterações necessária em destaque;

```
int cmd_ps(int argc, char far *argv[])
{
	int i,l;
	static char str[20];
	extern miniSO_PCB miniSO_thread[];

	argc=argc;
	argv=argv;
	putstr(" BCP   PID   PPID  STATUS   ZLIST  PREV   NEXT\n");
	for    (i=0;i<miniSO_MAXTHREADS;++i)  {
		if     (miniSO_thread[i].status!=FREE)  {
			inttostr(str,i);
			l=strlen(str);
			while  (l++<3)
				putch(' ');
			putstr(str);
			putstr("  ");
			inttostr(str,miniSO_thread[i].pid);
			l=strlen(str);
			while  (l++<5)
				putch(' ');
			putstr(str);
			putstr("  ");
			inttostr(str,miniSO_thread[i].ppid);
			l=strlen(str);
			while  (l++<5)
				putch(' ');
			putstr(str);
			putstr("  ");
			switch(miniSO_thread[i].status)  {
				case FREE:    putstr(strFREE);    break;
				case READY:   putstr(strREADY);   break;
				case RUNNING: putstr(strRUNNING); break;
				case ZOMBIE:  putstr(strZOMBIE);  break;
				case WAIT:    putstr(strWAIT);    break;
				case WAITSIG: putstr(strWAITSIG); break;
				case WAITSEM: putstr(strWAITSEM); break;
				case STOPPED: putstr(strSTOPPED); break;
			}
			putstr("  ");
			inttostr(str,miniSO_thread[i].zombies);
			l=strlen(str);
			while  (l++<5)
				putch(' ');
			putstr(str);
			putstr(" ");
			inttostr(str,miniSO_thread[i].prev);
			l=strlen(str);
			while  (l++<5)
				putch(' ');
			putstr(str);
			putstr("  ");
			inttostr(str,miniSO_thread[i].next);
			l=strlen(str);
			while  (l++<5)
				putch(' ');
			putstr(str);
			putch('\n');
		}
	}
	return 0;
}
```

Figura 31 – Alteração em `cmd_ps()`

- na função `sc_kill()`, em `scall.c`, considerar a exclusão de um processo no estado `STOPPED` – a Figura 32 mostra a nova versão da função `sc_kill()` com a alteração em destaque;

```
int sc_kill(pid_t pid)
{
          pcb_t      pcb,pcbw,last,prevthread,nextthread,pai,i,ant;
          int        sem,desbloqueia_o_pai;

          disable();
          /* Se remover a thread 0 ... */
          if         (pid==0)
                     end_mso("kill(): thread 0 excluida!");
          /* Tem que verificar se existe a thread */
          pcb = get_pcb(pid);
          /* Se nao existe uma thread com este numero, entao retorna */
          if         (pcb==-1)  {
                     enable();
                     return miniSO_ERROR;
          }
          /* Trata os zumbis do processo, colocando-os na lista de livres */
          i = miniSO_thread[pcb].zombies;
          while      (i!= -1)  {
                     miniSO_thread[pcb].zombies = miniSO_thread[i].next;
                     miniSO_thread[i].next   = miniSO_free;
                     miniSO_free             = i;
                     miniSO_thread[i].status = FREE;
                     i = miniSO_thread[pcb].zombies;
          }
          /* Passa os filhos do processo para o avo */
          for        (i=0;i<miniSO_MAXTHREADS;++i)  {
                     if         (miniSO_thread[i].status!=FREE && miniSO_thread[i].ppid == miniSO_thread[pcb].pid)
                                miniSO_thread[i].ppid = miniSO_thread[pcb].ppid;
          }
          miniSO_thread[pcb].exitcode = miniSO_ERROR;
          /* Salva anterior e proximo da thread que sera deletada */
          prevthread = miniSO_thread[pcb].prev;
          nextthread = miniSO_thread[pcb].next;
          /* A thread pode estar em um de varios estados... */
          switch     (miniSO_thread[pcb].status)  {
                     case       READY:
                     case       RUNNING:
                                /* Exclui o nodo da lista de prontos */
                                miniSO_thread[prevthread].next=nextthread;
                                miniSO_thread[nextthread].prev=prevthread;
                                if         (pcb==miniSO_ready)  {
                                           miniSO_delcurthread=1;
                                           if         (nextthread==miniSO_ready)
                                                      miniSO_nextthread = -1;  /* Por enquanto, seria a ultima */
                                           else
                                                      miniSO_nextthread = nextthread;
                                }
                                break;
                     case       WAITSEM:
                                /* Exclui o nodo da lista do semaforo */
                                sem = miniSO_thread[pcb].waitforsem;
                                /* Verifica se eh o primeiro da lista */
                                if         (pcb == miniSO_sem[sem].queue)
                                           miniSO_sem[sem].queue = nextthread;
                                else
                                           miniSO_thread[prevthread].next=nextthread;
                                if         (nextthread != -1)
                                           miniSO_thread[nextthread].prev=prevthread;
                                miniSO_sem[sem].value++;
                                break;
                     case       WAIT:
                     case       WAITSIG:
                     case       STOPPED:
                                break;
                     case       ZOMBIE:
                     default:
                                enable();
                                return miniSO_ERROR;
          }
          /* Coloca o BCP da thread na lista de filhos zumbis do pai */
          pai = get_pcb(miniSO_thread[pcb].ppid);
          if         (pai==-1) { /* Se nao encontrou o pai, */
                     /* coloca o processo na lista de livres */
                     miniSO_thread[pcb].status = FREE;
                     miniSO_thread[pcb].next = miniSO_free;
                     miniSO_free=pcb;
          }
          else  {
                     /* senao, coloca o processo na lista de zumbis do pai */
                     miniSO_thread[pcb].status = ZOMBIE;
                     if         (miniSO_thread[pai].zombies==-1) { /* Se a lista esta vazia */
                                /* Insere no inicio */
                                miniSO_thread[pai].zombies = pcb;
                                miniSO_thread[pcb].prev = -1;
                     }
                     else {
                                /* Senao insere no final */
                                i = miniSO_thread[pai].zombies;
                                while      (miniSO_thread[i].next!=-1)
                                           i = miniSO_thread[i].next;
                                miniSO_thread[i].next = pcb;
                                miniSO_thread[pcb].prev = i;
                     }
                     miniSO_thread[pcb].next = -1;
                     /* Verifica se o pai esta esperando em wait() ou waitpid() */
                     if         (miniSO_thread[pai].status==WAIT) {
                                desbloqueia_o_pai = 1;

                                if         (desbloqueia_o_pai) {
                                           /* Descobre qual a ultima thread */
                                           last = miniSO_thread[miniSO_ready].prev;
                                           /* Acerta links para incluir a thread no final da lista de prontos*/
                                           miniSO_thread[nextthread].prev    = last;
                                           miniSO_thread[nextthread].next    = miniSO_ready;
                                           miniSO_thread[last].next          = nextthread;
                                           miniSO_thread[miniSO_ready].prev  = nextthread;
                                           miniSO_thread[nextthread].exitcode = miniSO_ERROR;
                                           miniSO_thread[nextthread].status  = READY;
                                }
                     }
```

```
        }
        /* REVISAR */
  /* Se o processo-pai esta' esperando em wait, coloca ele na lista de prontos */
  nextthread = miniSO_thread[pcb].wait;
  if (nextthread != -1)  {
     /* Descobre qual a ultima thread */
     last = miniSO_thread[miniSO_ready].prev;
     /* Acerta links para incluir a thread no final da lista de prontos*/
     miniSO_thread[nextthread].prev    = last;
     miniSO_thread[nextthread].next    = miniSO_ready;
     miniSO_thread[last].next          = nextthread;
     miniSO_thread[miniSO_ready].prev  = nextthread;
     miniSO_thread[nextthread].exitcode = miniSO_ERROR;
     miniSO_thread[nextthread].status  = READY;
  }

        enable();
        if         (miniSO_delcurthread) {
                   if         (miniSO_nextthread == -1)
                              end_mso("kill(): nao ha' mais threads executando!");
                   while      (1)
                              ;
        }
        return miniSO_OK;
}
```

Figura 32 – Nova versão da função `sc_kill()`

- implementar, naturalmente, o código completo das funções `sc_stop()` e `sc_resume()` – a Figura 33 mostra a implementação definitiva destas funções.

```
int sc_stop(pid_t pid)
{
       pcb_t pcb,prevthread,nextthread;

       disable();
       /* Tem que verificar se existe a thread */
       pcb = get_pcb(pid);
       /* Se nao existe uma thread com este numero ou se o estado dela não é READY, retorna erro */
       if      (pcb==-1 || miniSO_thread[pcb].status != READY)  {
              enable();
              return miniSO_ERROR;
       }
       prevthread = miniSO_thread[pcb].prev;
       nextthread = miniSO_thread[pcb].next;
       miniSO_thread[pcb].next = -1;
       miniSO_thread[pcb].prev  = -1;
       /* Exclui o nodo da lista de prontos */
       miniSO_thread[prevthread].next=nextthread;
       miniSO_thread[nextthread].prev=prevthread;
       miniSO_thread[pcb].status = STOPPED;
       enable();
       return miniSO_OK;
}


int sc_resume(pid_t pid)
{
       pcb_t pcb,prevthread,nextthread,last;

       disable();
       /* Tem que verificar se existe a thread */
       pcb = get_pcb(pid);
       /* Se nao existe uma thread com este numero ou se o estado dela não é STOPPED */
       if      (pcb==-1 || miniSO_thread[pcb].status != STOPPED)  {
              enable();
              return miniSO_ERROR;
       }
       last = miniSO_thread[miniSO_ready].prev;
       miniSO_thread[pcb].prev = last;
       miniSO_thread[last].next=pcb;
       miniSO_thread[miniSO_ready].prev=pcb;
       miniSO_thread[pcb].next=miniSO_ready;
       miniSO_thread[pcb].status = READY;
       enable();
       return miniSO_OK;
}
```

Figura 33 – Implementação das funções `sc_stop()` e `sc_resume()`

6.5 O novo diagrama de transição de estados de processos do miniSO

O Quadro 5 mostra a nova descrição das listas do miniSO, incluindo o novo estado. O diagrama de transição de estados correspondente pode ser visto na Figura 34.

Quadro 5 – Relacionamento entre listas, estados e chamadas do miniSO

Lista	Tipo	Estado(s)	Inserção	Exclusão
`miniSO_free`	simplesmente encadeada e não circular	`FREE`	`wait(L)`	`fork(A)`

`miniSO_ready`	duplamente encandeada e circular (primeiro da lista em `RUNNING`, demais em `READY`)	`RUNNING`	`miniSO_clockhandler(K)`	`exit(C)`
				`kill(C)`
				`waitsignal(D)`
				`wait(G)`
				`miniSO_clockhandler(J)`

		`READY`	`fork(A)`
			`sendsignal(E)`
			`kill(H)`
			`exit(H)`
			`miniSO_clockhandler(J)`
				`kill(B)`
				`miniSO_clockhandler(K)`

`MiniSO_waitsig`	duplamente encadeada e não circular	`WAITSIG`	`waitsignal(D)`	`sendsignal(E)` `kill(F)`
campo `wait` do BCP	elemento simples	`WAIT`	`wait(G)`	`kill(H,I)` `exit(H)`
campo `zombies` do BCP	duplamente encadeadas e não circular	`ZOMBIE`	`kill(B,C,F,I)` `exit(C)`	`wait(L)`
miniSO_stopped	duplamente encadeada e não circular	`STOPPED`	`stop(M)`	`resume(N)` `kill(L)`

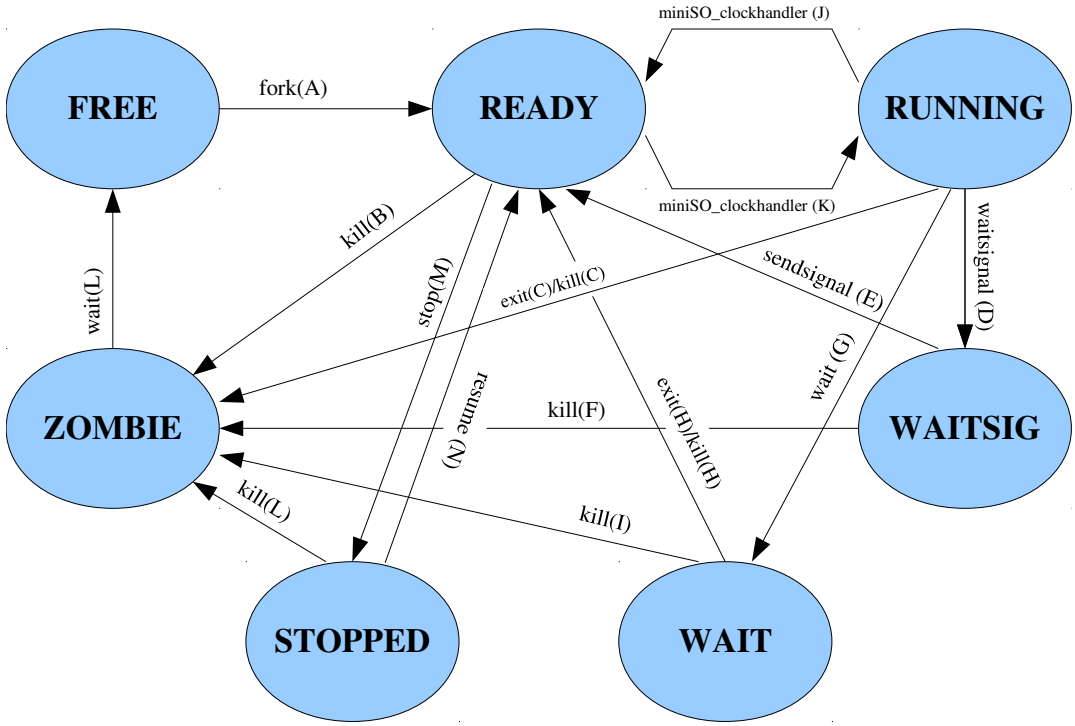

Figura 34 – Novo diagrama de transição de estados de processos do miniSO

7 Caminho percorrido pelas informações dentro do miniSO

Quando o usuário inicia a execução de um comando no miniSO e passa algum parâmetro para este comando, esta informação passa por várias etapas até ser processada no nível de chamada de sistema. O caminho percorrido pela informação é o seguinte:

- o interpretador de comandos processa a linha de comandos e envia cadeias de caracteres para a função que executará o comando (passagem normal de parâmetros para funções, em C);
- a função que executa o comando processa as cadeias de caracteres, eventualmente convertendo-as para outros tipos de dados, e passa estas informações como parâmetros para funções da biblioteca, em C;
- quando uma chamada de biblioteca precisa executar uma chamada de sistema, ela coloca os

parâmetros que recebe em registradores e executa uma interrupção de software;
- a rotina de atendimento da interrupção de software das chamadas de sistema (em linguagem de montagem) coloca os valores dos registradores na pilha, criando o contexto necessário para executar uma chamada de função em C, a partir do código em linguagem de montagem;
- o código da função que executa a chamada de sistema recebe os parâmetros e executa o processamento.

8 Implementação de comandos internos

Para implementar um comando interno, os seguintes passos devem ser seguidos:
- incrementar o valor da constante `MAXCOMMANDS` (que contém o número de comandos internos), definida em `command.h`;
- criar o protótipo da função que executará o comando interno, no final de `command.h` (a função deverá retornar um valor inteiro e receber dois parâmetros: `int argc` e `char far *argv[]`, da mesma forma que as funções para os outros comandos);
- acrescentar a definição do comando na inicialização do vetor `commands` em `command.c` (seguindo o mesmo padrão dos outros comandos internos cadastrados neste vetor);
- em `command.c`, implementar o novo comando.

9 Implementação de chamadas de biblioteca

Para implementar uma chamada de biblioteca basta criar o protótipo da chamada em `lib.h` e acrescentar a sua implementação em `lib.c`. Há chamadas de biblioteca que não executam chamadas de sistema (como, por exemplo, `strlen()`) e chamadas que executam chamadas de sistema (como, por exemplo, `putchar()`).

A Figura 35 mostra um exemplo de função de biblioteca que executa uma chamada de sistema. Neste código, `_BX` corresponde a uma forma de acessar o registrador BX de forma direta em C (este registrador é utilizado para passagem do parâmetro pid para a chamada de sistema). A primitiva `asm{}` permite a inserção de código em linguagem de montagem no código em C. Dentro desta primitiva, AH é inicializado com o número da chamada de sistema (`SC_RESUME`) e executa-se a interrupção de software 0x22. O resultado da execução da chamada de sistema é retornado no registrador, que é retornado pela função da biblioteca.

```
int resume(pid_t pid)
{
  _BX = pid;
  asm {
    mov ah,SC_RESUME
    int 22h
  }
  return(_AX);
}
```

Figura 37 – Função de biblioteca que executa uma chamada de sistema

10 Implementação de chamadas de sistema

Para criar uma chamada de sistema deve-se:
- incrementar a constante `miniSO_NUMSCALL` (que contém o número total de chamadas de sistema), em `scall.h`;
- definir uma constante inteira que identifique de forma única a chamada de sistema (iniciando com `SC_`, tal como as outras constantes com a mesma função), em `scall.h`;
- apresentar o protótipo das funções que implementarão as chamadas de sistema (iniciando com, `sc_`, tal como os outros protótipos das outras chamadas de sistema), em `scall.h`;
- atualizar a tabela de chamadas de sistema (`miniSO_scall_table`), em `scall.h`, acrescentando uma entrada para a nova chamada de sistema – esta entrada deve incluir: número da chamada, número de parâmetros que ela recebe e função que executa a chamada de sistema;
- acrescentar em `scall.c` o código da função que implementa a chamada de sistema.

11 Conclusão

Este artigo descreveu o desenvolvimento de um sistema operacional de pequeno porte criado para ser utilizado em disciplinas de projeto de sistemas operacionais. Um dos principais objetivos deste projeto foi manter o código relativamente pequeno para diminuir, desta forma, as dificuldades que alunos possam encontrar para compreender a sua estrutura e o seu funcionamento. Assim torna-se mais fácil iniciar o desenvolvimento de novos componentes para este sistema – o que naturalmente poderia ser utilizado como trabalho neste tipo de disciplina.

Também é um dos objetivos deste sistema operacional fornecer uma estrutura básica em cima da qual possam ser implementadas novas funcionalidades, tais como: arquivos executáveis externos (ou seja, processos e não apenas processos leves), novas chamadas de comunicação entre processos (semáforos, mensagens, etc.), novas chamadas para manipulação de arquivos e subdiretórios, gerenciamento de memória, etc.

O miniSO foi desenvolvido com o objetivo de ser usado como plataforma de testes em disciplinas de Sistemas Operacionais. O tamanho pequeno de seu código facilita a visualização de sua estrutura e a compreensão de seu funcionamento interno. Por outro lado, ele possui um conjunto de funcionalidades básicas que podem ser estendidas como trabalhos de disciplinas.

Esta aula apresentou, através de um exemplo prático, um roteiro para a implementação de comandos internos, rotinas de biblioteca e chamadas de sistema para um sistema operacional de pequeno porte chamado miniSO. O exemplo utilizado foi a suspensão e reinício de processos neste sistema operacional.

12 Exercícios

1. Considerando uma tabela de processos com 10 entradas e que a lista de prontos é formada pelos processos 10 (BCP=1), 20 (BCP=2), 30 (BCP=3) e 50 (BCP=5); que os processos 40 (BCP=4) e 70 (BCP=7) estão esperando por sinais; e o processo 0 (BCP=0) executou waitpid(10,&status). Mostre como a tabela de processos do miniSO representaria esta situação. Todos os demais BCPs (6, 8 e 9, exatamente nesta ordem) estão na lista de BCPs livres.

2. Qual opção ou quais das opções abaixo sobre o miniSO está ou estão corretas? Some o valor das alternativas corretas e preencha o campo valor.
   (1) O miniSO **não** utiliza o BIOS.
   (2) Há chamadas de sistema no miniSO que utilizam o BIOS e chamadas que não utilizam.
   (4) O nível de aplicação dentro do miniSO é codificado em linguagem de alto nível pura e usa chamadas de biblioteca.
   (8) As chamadas de biblioteca executam as chamadas de sistema.
   (16) O ponto de entrada das chamadas de sistema do miniSO está implementado em C e a implementação de cada chamada de sistema está em linguagem de montagem.
   Valor: ( )

3. Qual opção ou quais das opções abaixo sobre o controle de interrupções no miniSO está ou estão corretas? Some o valor das alternativas corretas e preencha o campo valor.
   (1) O miniSO utiliza duas interrupções: 0x8, interrupção do relógio; e 0x22, para implementação de chamadas de sistema.
   (2) A interrupção do relógio é desviada para permitir o controle de tempo de utilização de acesso a disco.
   (4) A frequência padrão da interrupção do relógio é de 18,2 vezes por segundo.
   (8) A chamada getvect(x), implementada no miniSO, obtém o endereço da rotina de atendimento da interrupção x.
   (16) Para definir a rotina de atendimento de interrupções utiliza-se a função putvect(x,rotina).
   Valor: ( )

4. Qual opção ou quais das opções abaixo sobre o chaveamento de contexto no miniSO está ou estão corretas? Some o valor das alternativas corretas e preencha o campo valor.
   (1) Quando uma interrupção é percebida pelo processador, automaticamente, os registradores de

estados, CS e IP são salvos na pilha.
(2) Os registradores SS e SP apontam para a pilha na memória, que é utilizada para salvamento temporário de registradores.
(4) Para passar a execução de um processo para outro, o miniSO tem que salvar os registradores de estados, CS e IP no BCP (Bloco de Controle de Processo) do processo atual e restaurar estes registradores do BCP do novo processo.
(8) As instruções POP e PUSH são utilizadas, respectivamente, para empilhar e desempilhar registradores na rotina de atendimento da interrupção do relógio.
(16) Variáveis locais da rotina de atendimento da interrupção do relógio são alocadas juntamente com as variáveis globais, ou seja, acessadas através do registrador DS.
Valor: ( )

5. Qual opção ou quais das opções abaixo sobre as listas de processos e sobre os estados do miniSO está ou estão corretas? Some o valor das alternativas corretas e preencha o campo valor.
(1) A lista de prontos é simplesmente encadeada e circular.
(2) A lista de BCPs livres é simplesmente encadeada e não circular.
(4) O processo que está em execução é o primeiro da lista de prontos e o seu estado é RUNNING.
(8) Processos esperando por sinais (que executaram waitsignal), permanecem numa lista de espera duplamente encadeada e não circular chamada miniSO_waitsig.
(16) Quando um processo espera pelo final da execução de um processo-filho seu estado é WAITSIG.
Valor: ( )

6. Qual opção ou quais das opções abaixo sobre as chamadas para gerência de processos e sobre as listas do miniSO está ou estão corretas? Some o valor das alternativas corretas e preencha o campo valor.
(1) A chamada fork() retira um processo da lista de BCPs livres e o insere na lista de prontos.
(2) A inserção na lista de prontos é sempre feita no final da lista.
(4) Remoções da lista de prontos devem ser sempre feitas a partir do início da lista.
(8) A chamada kill() encerra um processo, removendo-o de qualquer lista e inserindo-o na lista de BCPs livres.
(16) A chamada sendsignal() pode retirar um processo da lista de espera por sinais (miniSO_waitsig) e inseri-lo na lista de prontos.
Valor: ( )

7. Elaborar dois comandos, duas chamadas de biblioteca e duas chamadas de sistema. para o miniSO. Os comandos deverão se chamar: initcnt e nextcnt. Nenhum dos dois deverá receber qualquer parâmetro. A implementação de initcnt simplesmente chama a função de biblioteca initcnt(). A implementação de nextcnt chama a função de biblioteca nextcnt() e imprime o valor retornado por esta função. As chamadas de biblioteca terão a seguinte sintaxe:

```
void initcnt();
unsigned nextcnt();
```

Estas chamadas simplesmente executam as chamadas de sistema SC_INITCNT e SC_NEXTCNT. As chamadas de sistema devem utilizar os próximos valores de constantes disponíveis em scall.h. A implementação delas deve utilizar uma variável estática unsigned e deve ser a seguinte:

```
static unsigned cnt = 0;

void sc_initcnt()
{
      cnt = 0;
}

unsigned sc_nextcnt()
{
      return (cnt++);
}
```

Resultado esperado: a cada vez que o usuário executar o comando nextcnt, aparecerá o próximo valor do contador na tela; se for chamado o comando initcnt, o valor será zerado e nextcnt começará em zero novamente.

REFERÊNCIAS

TEODOROWITSCH, Roland. **NC – Uma Biblioteca para Gerência de Threads no Sistema Operacional MS-DOS**. Canoas: [s.n.], 1999. (Apostila da disciplina Projeto de Sistemas Operacionais, Curso de Ciência da Computação, Universidade Luterana do Brasil).

TEODOROWITSCH, Roland. **Bootlib**: uma linguagem para definição de setores de inicialização. In: XXVI CONFERENCIA LATINOAMERICANA DE INFORMÁTICA, Atizapán de Zaragoza: Tec de Monterrey, 2000.