Rafael Coninck Teigão
Julio Henrique Morimoto

Plano de Testes
BananaKernel*: Um Sistema Operacional Didático*

Curitiba - PR

Junho 2004

Rafael Coninck Teigão
Julio Henrique Morimoto

Plano de Testes
BananaKernel*: Um Sistema Operacional Didático*

Trabalho apresentado pelos alunos do 7º período do curso de Bacharelado em Ciência da Computação para a disciplina Projeto Final I.

Orientador:
Prof. Dr. Carlos Alberto Maziero

Pontifícia Universidade Católica do Paraná

Curitiba - PR

Junho 2004

Sumário

Lista de Tabelas

1 Introdução

A criação de um sistema operacional, além de ser um tópico de pesquisa muito interessante, pois envolve muitos conceitos desenvolvidos em cursos de Ciência e Engenharia de Computação, é também uma oportunidade para criar uma ferramenta de ensino para a disciplina *Sistemas Operacionais*.

Como será visto a seguir, na seção 1.1, este projeto tem como intenção criar um sistema operacional que irá explicar o que está sendo feito internamente, para que o aluno tenha uma maior facilidade de visualização dos processos internos deste tipo de sistema, além de possuir várias funções (vide seção 1.2) para facilitar a modificação e inclusão de novas partes criadas pelo aluno.

Este documento encontra-se dividido em 5 partes, sendo elas:

1. Esta introdução, contendo, também, os objetivos e o escopo do projeto;
2. uma descrição dos objetivos dos testes;
3. as condições dos testes, incluindo o detalhamento do ambiente necessário e de cada teste;
4. a data de conclusão e a assinatura dos responsáveis; e
5. anexos.

1.1 Objetivo

O propósito do sistema é proporcionar uma ferramenta para auxiliar no ensino da disciplina *Sistemas Operacionais*, demonstrando alguns dos mecanismos internos de funcionamento de um SO [1] moderno.

[1] Sistema Operacional

O sistema deve apresentar mensagens explicando partes do funcionamento, deve possuir uma documentação e um desenvolvimento que possibilite aos alunos e professores a criação de novos módulos para substituir ou acrescentar funcionalidades.

1.2 Escopo

Existe uma necessidade no meio acadêmico de um sistema que facilite o ensino de conceitos de SO para alunos, principalmente dos cursos de Bacharelado em Ciência da Computação e Engenharia de Computação.

Este sistema deve possuir uma documentação que explique o funcionamento de conceitos importantes, como gerência de arquivos e memória, por exemplo, e ainda permita que o aluno ou o professor faça substituições em módulos responsáveis pela apresentação desses conceitos.

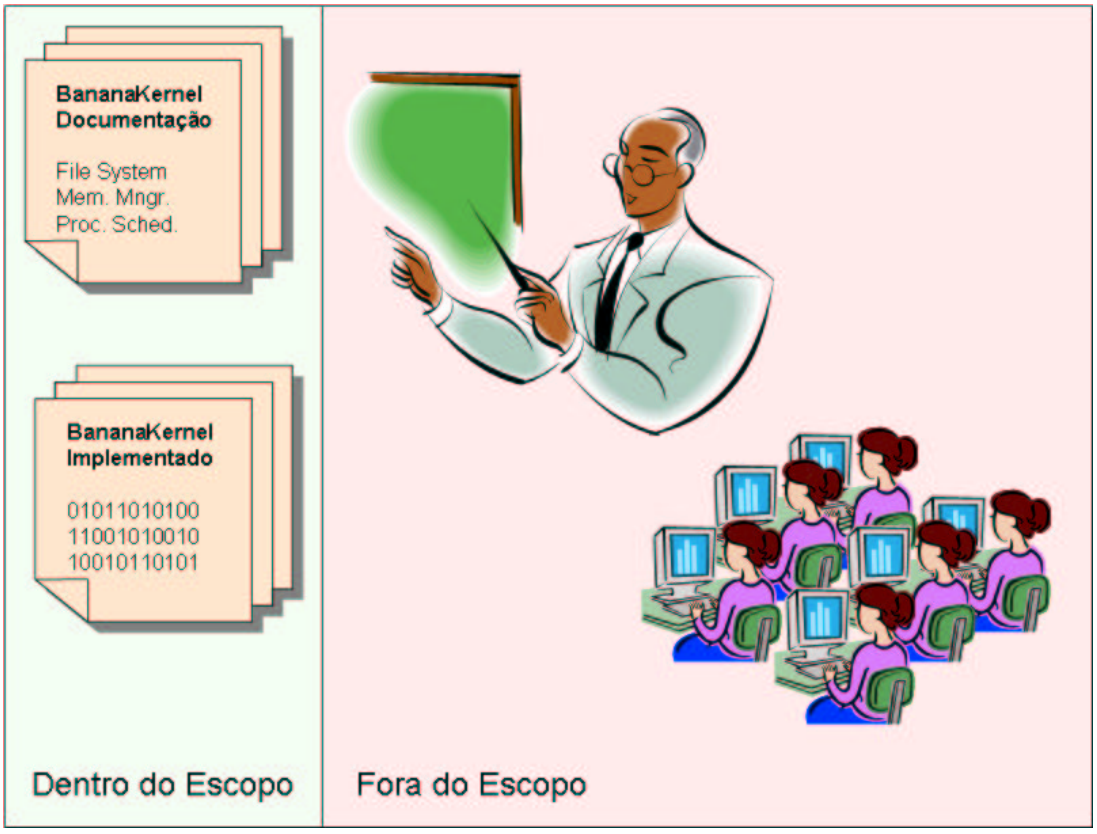

Figura 1: Definição do Escopo do Projeto

O escopo deste projeto, ilustrado na figura 1, cobre a criação de um sistema operacional simples, contendo módulos de gerência de memória (vide Anexo A.1), gerência de arquivos (vide Anexo A.2) e escalonador de processos (vide Anexo A.3), sendo que

esses módulos devem apresentar mensagens na tela do computador explicando o funcionamento interno de cada subsistema (na figura, ilustrado com o título "BananaKernel Implementação").

O projeto cobre, também, a criação de um manual de usuário que explique como incluir ou substituir módulos, além de uma breve introdução à tecnologia e ao método de desenvolvimento utilizado na criação do sistema, e uma função `bnnk_perror()` como descrita em (American National Standards Institute, 1989), responsável pela apresentação das mensagens de erro (na figura, ilustrado com o título "BananaKernel Documentação").

Este projeto não cobre, porém, ensinar aos usuários linguagens de programação, nem pretende montar um curso de sistemas operacionais, como mostra a figura 1 na página precedente. O fornecimento do equipamento necessário para implantação, as demais ferramentas para a execução do sistema ou para condução das aulas e os *software* e sistemas necessários para o desenvolvimento ou modificação de módulos também não estão cobertos (lado direito da figura, mostrando o professor, os alunos e os equipamentos utilizados).

Assume-se, como premissas, que os usuários têm conhecimento e capacidade de criar programas nas linguagens de programação adequadas à alternativa escolhida, bem como capacidade de compreender textos escritos em inglês técnico, além de estarem sendo acompanhados por um professor capacitado da disciplina *Sistemas Operacionais*.

2 Objetivos dos Testes

Os testes descritos neste documento têm como objetivo assegurar que:

- os requisitos funcionais estão sendo cumpridos, ou seja, as funções descritas em (TEIGÃO; MORIMOTO, 2004b) e na seção 1.2 e anexos A.1, A.2 e A.3 deste documento, sendo chamadas em um ambiente adequado e recebendo os parâmetros corretos, executam o código esperado e fornecem a resposta correta em retorno; e
- os requisitos não-funcionais sejam satisfeitos, ou seja, um aluno de um curso universitário de computação, cursando a disciplina *Sistemas Operacionais* e com domínio da linguagem C, possa, utilizando como documentação do *BananaKernel* apenas o **Manual do Usuário** (TEIGÃO; MORIMOTO, 2004a), construir e executar o *kernel* exemplo e alterar e criar módulos.

3 Condições do Teste

Neste capítulo serão descritos o ambiente e os casos de teste para o requisitos funcionais e não-funcionais.

3.1 Ambiente

O ambiente de teste deverá contar com:

- Computador AMD K6-2 500MHz, 32MB RAM, HD 20GB (plataforma escolhida por ser modesta e presente na maioria das instituições de ensino superior no Brasil);
- instalação do sistema operacional Linux 2.4.20-30.9;
- instalação do emulador Bochs 2.1.1;
- instalação da biblioteca OSKit Snapshot 20020317;
- instalação do compilador GCC (GNU Compiler) 3.2.2 e suas bibliotecas e ferramentas associadas; e
- manual do usuário para o *BananaKernel* (TEIGÃO; MORIMOTO, 2004a), em formato impresso.

3.2 Casos de Teste de Requisitos Funcionais

Foi criado um caso de teste para cada um dos 4 grandes módulos do projeto, sendo eles:

1. Gerenciador de Memória;
2. Gerenciador de Arquivos;

3. Escalonador de Processos;

4. Informativos de Erros.

Independente da implementação, simples ou intermediária, os casos de teste deverão verificar a ação correta de todas as funções dentro desses módulos.

Esses casos de teste serão descritos nesta seção.

3.2.1 Caso de Teste do Gerenciador de Memória: #01MM

Identificação: #01MM

Objetivos Específicos: verificar a capacidade do gerenciador de memória de alocar blocos de memória com permissão de escrita, re-alocar blocos já preenchidos com dados, efetuar os cálculos para encontrar o volume disponível de memória, calcular o tamanho de um bloco e liberar blocos alocados.

Descrição: para testar o gerenciador de memória, será utilizada uma função que deverá alocar alguns blocos de memória, de tamanhos variáveis, em seguida escrever dados (seqüências de 1) nesses blocos, re-alocar alguns blocos, mostrar a quantidade de memória disponível, liberar um bloco, mostrar, novamente, o tamanho da memória disponível, e mostrar o tamanho de um bloco. A função que deverá ser utilizada encontra-se no anexo B.

Pré-condições: o sistema deve estar sendo executado em um equipamento cuja memória esteja certificada contra falhas.

Resultados Esperados: este teste considerará este gerenciador satisfatório quando a função retornar 0 (zero), e não satisfatório se retornar um dos códigos de erro da tabela 1.

Tabela 1: #01MM - Códigos de Erro

código	função com erro
-1	`bnnk_malloc()`
-2	`bnnk_realloc()`
-3	`bnnk_free()`
-4	`bnnk_avail()`
-5	`bnnk_getsize()`

3.2.2 Caso de Teste do Gerenciador de Arquivos: #02FS

Identificação: #02FS

Objetivos Específicos: verificar a capacidade do gerenciador de arquivos de manter e alterar uma tabela de alocação de arquivos e permitir que dados sejam escritos em e lidos de arquivos armazenados no sistema de arquivos em disco rígido.

Descrição: para testar o gerenciador de arquivos, será utilizada uma função que deverá abrir um arquivo novo, escrever em seu conteúdo e fechar este arquivo, em seguida, este procedimento deverá ser repetido para um segundo arquivo. Então, ambos os arquivos deverão ser abertos e seu conteúdo, copiado para a memória, porém esta abertura deverá ser feita antes no segundo arquivo, para garantir o bom funcionamento da tabela de alocação. Os dois arquivos serão, então, fechados novamente. A função que deverá ser utilizada encontra-se no anexo C.

Pré-condições: o disco em que esses arquivos estão sendo criados não deve possuir falhas e deve conter espaço suficiente para a criação dos arquivos.

Resultados Esperados: este teste considerará este gerenciador satisfatório quando a função retornar 0 (zero), e não satisfatório se retornar um dos códigos de erro da tabela 2.

Tabela 2: #02FS - Códigos de Erro

código	função com erro
-1	`bnnk_open()`
-2	`bnnk_read()`
-3	`bnnk_write()`
-4	`bnnk_close()`

3.2.3 Caso de Teste do Escalonador de Processos: #03SC

Identificação: #03SC

Objetivos Específicos: verificar a capacidade do escalonador de processos de adicionar e remover processos da fila de processos ativos e alternar a execução entre esses processos.

Descrição: para testar o escalonador de processos, serão criadas três funções, duas para serem os processos, e uma para adicionar e remover os processos. O escalonador deverá alternar entre os dois processos. As funções que deverão ser utilizadas encontram-se no anexo D.

Pré-condições: não se aplica.

Resultados Esperados: este teste considerará este escalonador satisfatório quando as três funções retornarem 0 (zero) e a mensagem que as duas primeiras imprimem aparecerem em blocos de texto alternados, e não satisfatório se retornar um dos códigos de erro da tabela 3, e não alternar entre as funções 1 e 2, indicando um erro na função `bnnk_swapproc()`.

Tabela 3: #03SC - Códigos de Erro

código	função com erro
-1	`bnnk_addproc()`
-2	`bnnk_delproc()`

3.2.4 Caso de Teste do Informativo de Erros: #04PE

Identificação: #04PE

Objetivos Específicos: verificar se o escalonador de processos contém as mensagens corretas para cada código de erro.

Descrição: para testar o informativo de erros, será criada uma função que irá configurar a variável global **errno** e, em seguida, chamar a função `bnnk_perror()`, sucessivamente, para todos os códigos de erro disponíveis. Estes códigos podem ser encontrados no arquivo *errno.h* das implementações simples e intermediárias. Não foi criada, neste momento, uma função completa para executar este teste, pois o conteúdo das mensagens de erro ainda não foi definido. No anexo encontra-se uma versão simplificada desta função.

Pré-condições: não se aplica.

Resultados Esperados: este teste considerará este informativo satisfatório quando todas as mensagens impressas na tela forem iguais àquelas que estão associadas aos respectivos códigos erro.

3.3 Casos de Teste de Requisitos Não-Funcionais

Foram definidos como requisitos não-funcionais a serem testados a abrangência, a facilidade de uso e a clareza do **Manual do Usuário**, considerado a interface geral entre o usuário e o sistema. Para isso, foi criado o caso de teste #05UM.

Estes requisitos foram levantados junto com o orientador do projeto, com o auxilio do artigo[1] (ZANLORENCI; BURNETT, 2001).

3.3.1 Caso de Teste do Manual do Usuário: #05UM

Identificação: #05UM

Objetivos Específicos: verificar a capacidade do Manual do Usuário de expressar correta e claramente os mecanismos de alteração, construção e execução do *kernel* do sistema.

Descrição: para testar o Manual do Usuário, serão escolhidos pelo professor orientador alguns alunos de uma turma da disciplina *Sistemas Operacionais* de um curso universitário de computação da PUCPR, que estejam dispostos a aprender sobre o *BananaKernel*. Estes alunos deverão ter um bom domínio da linguagem C. Será, então, disponibilizado a estes alunos um ambiente conforme a descrição da seção 3.1. Estes alunos deverão conseguir construir e executar o *kernel* exemplo que acompanha o sistema e fazer pequenas modificações em módulos prontos, como mostra a tabela 4.

Tabela 4: Tarefas a Serem Executadas pelos Alunos

#	identificação	descrição	pontos
1	**modbr**	o aluno deve construir e criar a imagem do *kernel* exemplo, move-la para a imagem de disco vazia e executá-la dentro do emulado Bochs	40
2	**modmm**	o aluno deve alterar a função `bnnk_free()` da versão simples para devolver a memória do bloco liberado ao conjunto de memória disponível, quando este bloco for o último alocado	30
3	**modfs**	o aluno deve, utilizando o código da função `bnnk_open()` da versão simples, criar uma função `ls()` que deverá listar todos os arquivos da tabela de alocação	20
4	**modsc**	o aluno deve alterar a função `bnnk_swaproc()` da versão intermediária para envelhecer mais rapidamente os processos na fila de espera	10

[1] O artigo "Requisitos funcionais e não-funcionais, as duas faces da moeda aplicáveis à engenharia de *software*" ajudou a compreender a diferença entre requisitos funcionais e não-funcionais.

Pré-condições: os alunos devem conhecer a base teórica das estruturas básicas de um sistema operacional, como gerenciadores de memória e arquivos e escalonador de processos, além de estarem acompanhados por um professor da disciplina *Sistemas Operacionais* durante este teste, para esclarecerem eventuais dúvidas com relação a essa base teórica.

Resultados Esperados: é esperado que a pontuação média dos alunos seja 60 pontos.

4 Responsabilidades

Curitiba, 09 de junho de 2004.

São responsáveis por este Plano de Testes:

Alunos:

Rafael Coninck Teigão

Julio Henrique Morimoto

Orientador:

Prof. Dr. Carlos Alberto Maziero

ANEXO A – Descrição Detalhada do Escopo

Neste anexo estão descritos em detalhes os requisitos dos gerenciadores de memória e arquivos e do escalonador.

A.1 Escopo da Gerência de Memória

O gerenciador de memória deve disponibilizar as funções:

`bnnk_malloc()` responsável pela alocação de memória;

`bnnk_realloc()` responsável pela re-alocação de memória;

`bnnk_free()` responsável pela liberação de uma parte da memória;

`bnnk_avail()` retorna a quantidade de memória disponível; e

`bnnk_getsize()` retorna o tamanho de um volume de memória reservado.

Os nomes, apesar de estarem em inglês, mantém uma coerência com aqueles empregados na linguagem C, e são familiares aos alunos.

Essas funções devem estar disponíveis em duas versões:

- alocação contínua; e
- alocação baseada em listas (TANENBAUM, 2001).

A.2 Escopo da Gerência de Arquivos

O gerenciador de arquivos deve disponibilizar as funções:

`bnnk_open()` abre um arquivo em disco;

`bnnk_read()` faz a leitura de um arquivo aberto no disco para a memória;

`bnnk_write()` escreve em um arquivo aberto; e

`bnnk_close()` fecha o arquivo aberto em disco.

Novamente, os nomes estão em inglês para manter coerência com aqueles utilizados na linguagem C.

Essas funções devem estar disponíveis em duas versões:

- escrita contínua em disco; e
- escrita utilizando o *ext2fs* (CARD; TS'O; TWEEDIE, 1994).

A.3 Escopo do Escalonador de Processos

O escalonador de processos deve fornecer duas funções:

`bnnk_addproc()` inclui um novo processo á lista de processos;

`bnnk_delproc()` remove um processo desta lista; e

`bnnk_swapproc()` muda o contexto para o próximo processo.

Os nomes foram mantidos em inglês para estarem coerentes com as demais funções implementadas.

O escalonador de processos também estará disponível em duas versões:

- escalonador *FIFO*[1], sem prioridade ou envelhecimento;
- escalonador com prioridade e envelhecimento.

[1] *First In - First Out* (primeiro a entrar, primeiro a sair).

ANEXO B – Teste #01MM

A seguinte função foi criada para testar as funções disponíveis no sistema, responsáveis pelo Gerenciador de Memória (vide subseção 3.2.1).

```
/* Esta funcao testa o gerenciador de memoria. Retorna 0 em sucesso. */

int
teste_01MM()
{
        char            *memoria[10];
        int             i, j;
        int             disponivel = -1;

        /* teste bnnk_malloc() */

        for (i = 0; i < 10; i++) {    /* para cada posicao do vetor... */
                /*
                 * cria um a posicao de memoria para conter (10*i)+10
                 * caracteres
                 */
                memoria[i] = (char *) bnnk_malloc(sizeof(char) * ((10 * i) + 10));
                if (!memoria[i]) {      /* se nao conseguiu um bloco */
                        return -1;      /* retorna -1 indicando um erro no
                                         * bnnk_malloc() */
                }
        }

        /* teste de escrita */

        for (i = 0; i < 10; i++) {    /* para cada posicao do vetor... */
                /* para cada char no bloco... */
                for (j = 0; (j < ((10 * i) + 10) - 1) || (j == 0); j++) {
                        /* preenche com 1 */
```

```
30                      memcpy(((memoria[i] + j)), "1", sizeof(char));
                }
                memcpy((memoria[i] + j), "\0", 1);  /* preenche a ultima
                                                    * posicao com \0,
                                                    * indicando o fim da
35                                                  * string */
                /* imprime a string */
                printf("Numero de characteres: %i\n", j);
                printf("Caracteres: %s\n", memoria[i]);
        }
40
        /* teste bnnk_realloc() */

        /* realocar os blocos 0, 5 e 9 */
        memoria[0] = (char *) bnnk_realloc(memoria[0], sizeof(char) * 150);
45      memoria[5] = (char *) bnnk_realloc(memoria[5], sizeof(char) * 150);
        memoria[9] = (char *) bnnk_realloc(memoria[9], sizeof(char) * 150);

        /* verifica se os blocos foram realocados */
        /* se nao foram criados */
50      if (!memoria[0] || !memoria[5] || !memoria[9])
                return -2;    /* retorna -2 indicando um erro no
                               * bnnk_realloc() */

        /* imprime esses blocos para garantir que o conteudo foi copiado */
55      printf("Caracteres: %s\n", memoria[0]);
        printf("Caracteres: %s\n", memoria[5]);
        printf("Caracteres: %s\n", memoria[9]);

        /* teste de bnnk_avail() */
60      disponivel = bnnk_avail();
        if (disponivel < 0)    /* se a memoria estah negativa... */
                return -4;    /* retorna -4 indicando um erro no
                               * bnnk_avail() */

65      /* teste de bnnk_free() */
        i = bnnk_free(memoria[5]);    /* libera o bloco na posicao 5 */
        if (i == -1)            /* se bnnk_free() retornou -1 */
                return -3;    /* retorna -3 indicando um erro no
                               * bnnk_free() */
70
        /* teste de bnnk_avail() */
        disponivel = bnnk_avail();
```

```
        if (disponivel < 0)     /* se a memoria estah negativa... */
                return -4;      /* retorna -4 indicando um erro no
                                 * bnnk_avail() */

        /* teste bnnk_getsize() */
        i = bnnk_getsize(memoria[2]); /* recupera o tamanho do bloco na
                                       * posicao 2 */
        if (i != (sizeof(char) * ((10 * 2) + 10)))   /* se o tamanho do bloco
                                                      * eh diferente do
                                                      * tamanho alocado para
                                                      * ele... */
                return -5;      /* retorna -5 indicando um erro no
                                 * bnnk_getsize() */

        return 0;               /* retorna 0 indicando sucesso */
}
```

ANEXO C – Teste #02FS

A seguinte função foi criada para testar as funções disponíveis no sistema, responsáveis pelo Gerenciador de Arquivos (vide subseção 3.2.2).

```
/* Esta funcao testa o gerenciador de arquivos. Retorna 0 em sucesso */

int
teste_02FS()
{
        int             arquivo1 = 0, arquivo2 = 0;
        char            buffer[25];
        int             i = 0;

        /* ARQUIVO 1 */

        /* testa bnnk_open() */

        arquivo1 = bnnk_open("teste1"); /* abre o arquivo 1 */

        if (!arquivo1) {        /* se nao abrio o arquivo 1 */
                printf("Problema ao abrir arquivo 1\n");
                return -1;      /* retorna -1 indicando erro no bnnk_open() */
        }

        /* testa bnnk_write() */

        /* escreve os 25 bytes da string no arquivo 1 */
        i = bnnk_write(arquivo1, "teste #02FS - arquivo 1\0", 25);

        if (i != 25) {          /* se o numero de bytes escritos for
                                 * diferente de 25, que deve ser retornado... */
                printf("Problema ao escrever arquivo 1\n");
                return -3;      /* retorna -3 indicando erro no bnnk_write() */
```

```
30      }

        /* testa o bnnk_close() */

        i = bnnk_close(arquivo1);
35      if (i != 0) {           /* se nao conseguiu fechar... */
                printf("Problema ao fechar arquivo 1\n");
                return -4;      /* retorna -4 indicando erro no bnnk_close() */
        }

40      /* ARQUIVO 2 */

        /* testa o bnnk_open() */

        arquivo2 = bnnk_open("/tmp/teste2"); /* abre o arquivos 2 */
45
        if (!arquivo2) {        /* se nao abrio o arquivo 2 */
                printf("Problema ao abrir arquivo 2\n");
                return -1;      /* retorna -1 indicando erro no bnnk_open() */
        }
50
        /* testa o bnnk_write() */

        /* escreve os 25 bytes da string no arquivo 2 */
        i = bnnk_write(arquivo2, "teste #02FS - arquivo 2\0", 25);
55
        if (i != 25) {          /* se o numero de bytes escritos for
                                 * diferente de 25, qu e deve ser
                                 * retornado... */
                printf("Problema ao escrever arquivo 2\n");
60              return -3;      /* retorna -3 indicando erro no bnnk_write() */
        }

        /* testa o bnnk_close() */

65      i = bnnk_close(arquivo2);
        if (i != 0) {           /* se nao conseguiu fechar... */
                printf("Problema ao fechar arquivo 2\n");
                return -4;      /* retorna -4 indicando erro no bnnk_close() */
        }
70
        /* ARQUIVOS 1 & 2 */
```

```
        /* testa o bnnk_open() */

75      arquivo2 = bnnk_open("/tmp/teste2"); /* abre o arquivos 2 */

        if (!arquivo2) {        /* se nao abrio o arquivo 2 */
                printf("Problema ao abrir arquivo 2\n");
                return -1;      /* retorna -1 indicando erro no bnnk_open() */
80      }

        arquivo1 = bnnk_open("teste1"); /* abre o arquivo 1 */

        if (!arquivo1) {        /* se nao abrio o arquivo 1 */
85              printf("Problema ao abrir arquivo 1\n");
                return -1;      /* retorna -1 indicando erro no bnnk_open() */
        }

        /* testa o bnnk_read() */
90
        i = bnnk_read(arquivo1, buffer);        /* le o arquivo 1 para o
                                                 * buffer */

        if (i != 25) {          /* se o numero de bytes lidos for diferente
95                               * de 25, qu e deve ser retornado... */
                printf("Problema ao ler arquivo 1\n");
                return -2;      /* retorna -2 indicando erro no bnnk_read() */
        }
        printf("Conteudo do arquivo 1: %s\n", buffer);
100
        i = bnnk_read(arquivo2, buffer);        /* le o arquivo 2 para o
                                                 * buffer */

        if (i != 25) {          /* se o numero de bytes lidos for diferente
105                              * de 25, qu e deve ser retornado... */
                printf("Problema ao ler arquivo 21\n");
                return -2;      /* retorna -2 indicando erro no bnnk_read() */
        }

110     /* testa o bnnk_close() */

        i = bnnk_close(arquivo1);
        if (i != 0) {           /* se nao conseguiu fechar... */
                printf("Problema ao fechar arquivo 1\n");
115             return -4;      /* retorna -4 indicando erro no bnnk_close() */
```

```
        }

        i = bnnk_close(arquivo2);
        if (i != 0) {           /* se nao conseguiu fechar... */
120             printf("Problema ao fechar arquivo 2\n");
                return -4;      /* retorna -4 indicando erro no bnnk_close() */
        }

        return 0;               /* retorna 0 em sucesso */
125 }
```

ANEXO D – Teste #03SC

A seguinte função foi criada para testar as funções disponíveis no sistema, responsáveis pelo Escalonador de Processos (vide subseção 3.2.3). Esta função irá adicionar dois processos, compostos pelas funções descritas abaixo desta.

```
/*
 * Esta funcao adicona os dois processos na lista de processos e, depois de
 * um tempo, os remove
 */

int
teste_03SC()
{
        int             funcao1 = 0, funcao2 = 0;
        int             i = 0;

        funcao1 = bnnk_addproc(teste_03SC_funcao1); /* chama bnnk_addproc()
                                                     * com a estrutura do
                                                     * processo contendo a
                                                     * funcao 1 */

        if (!funcao1)           /* se nao incluiu o processo */
                return -1;      /* retorne -1 indicando um erro no
                                 * bnnk_addproc() */

        funcao2 = bnnk_addproc(teste_03SC_funcao2); /* chama bnnk_addproc()
                                                     * com a estrutura do
                                                     * processo contendo a
                                                     * funcao 2 */

        if (!funcao2)
                return -1;      /* retorne -1 indicando um erro no
                                 * bnnk_addproc( ) */
```

```
30      for (i = 0; i < 2000; i++)   /* aguarda o final dos processos */
            printf("Funcao 0 - Aguardando o Final do Processos!\n");

        funcao1 = bnnk_delproc(funcao1);    /* remove a funcao 1 */

35      if (funcao1 == -1)  /* se deu erro na remocao */
            return -2;      /* retorne -2 indicando um erro no
                             * bnnk_delproc() */

        funcao2 = bnnk_delproc(funcao2);    /* remove a funcao 2 */
40
        if (funcao2 == -1)  /* se deu erro na remocao */
            return -2;      /* retorne -2 indicando um erro no
                             * bnnk_delproc() */

45      return 0;           /* retorna 0 indicando sucesso */
}
```

Esta função, que descreve o processo 1, imprime "Função 1 - Hello World!" na tela durante 1000 (mil) repetições.

```
/* Esta funcao imprime "Funcao 1 - Hello World!" na tela */

void
teste_03SC_funcao1()
5 {
        int             i = 0;

        for (i = 0; i < 1000; i++) {
            printf("Funcao 1 - Hello World!\n");
10      }
}
```

Esta função, que descreve o processo 2, imprime "Função 2 - Goodbye World!" na tela durante 1000 (mil) repetições.

```
/* Esta funcao imprime "Funcao 2 - Goodbye World!" na tela */

void
teste_03SC_funcao2()
{
        int             i = 0;

        for (i = 0; i < 1000; i++) {
                printf("Funcao 2 - Goodbye World!\n");
        }
}
```

ANEXO E – Teste #04PE

A seguinte função foi criada para testar as funções disponíveis no sistema, responsáveis pelo Informativo de Erros (vide subseção 3.2.4).

```
/*
 * Esta eh uma versao simplificada da funcao que irah testar os codigos de
 * erro
 */

void
teste_04PE()
{
        int             i = 0;

        for (i = 1; i <= ERRO_MAXIMO; i++) { /* para todos os valores de
                                              * erro disponiveis no
                                              * sistema, comecando e um e
                                              * indo ateh o final */
                errno = i;      /* configura o errno */
                bnnk_perror(); /* chama o bnnk_perror() para mostrar a
                                * mensagem completa */
                /*
                 * deve-se imprimir, aqui, cada uma das mensagens de erro
                 * encontradas codificadas no sistema, de acordo com a
                 * variacao de i, para que se possa comparar com o valor
                 * impresso por bnnk_perror()
                 */
        }
}
```

Referências Bibliográficas

American National Standards Institute. *American National Standard Programming Language C, ANSI X3.159-1989.* 1430 Broadway, New York, NY 10018, USA, December 1989.

CARD, R.; TS'O, T.; TWEEDIE, S. Design and implementation of the second extended filesystem. In: STATE UNIVERSITY OF GRONIGEN. *Proceedings of the First Dutch International Symposium on Linux.* 1994. Disponível em: <http://e2fsprogs.sourceforge.net/ext2intro.html>. Acesso em: 28 de abril 2004.

TANENBAUM, A. Memory management with linked lists. In: VRIJE UNIVERSITEIT. *Modern Operating Systems.* 2^a. ed. [S.l.]: Prentice Hall, 2001. cap. 4.2.2, p. 200–201.

TEIGÃO, R. C.; MORIMOTO, J. H. BananaKernel*: Um Sistema Operacional Didático - Manual do Usuário.* [S.l.], 2004. Disponível em: <http://bananakernel.sourceforge.net/manual/>. Acesso em: 20 de maio de 2004.

TEIGÃO, R. C.; MORIMOTO, J. H. *Projeto Lógico* - BananaKernel*: Um Sistema Operacio nal Didático.* [S.l.], maio 2004. Disponível em: <http://bananakernel.sourceforge.net/docs/logico.pdf>. Acesso em: 20 de maio de 2004.

ZANLORENCI, E. P.; BURNETT, R. C. Requisitos funcionais e não-funcionais, as duas faces da moeda aplicáveis à engenharia de software. 2001. Disponível em: <http://www.pr.gov.br/batebyte/edicoes/2001/bb115/requisitos.htm>. Acesso em: 17 de maio de 2004.

Índice Remissivo