—

Laboratório de Sistemas Computacionais: Sistemas Operacionais - Relatório Final

São José dos Campos - Brasil

Setembro de 2018

—

Laboratório de Sistemas Computacionais: Sistemas Operacionais - Relatório Final

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratórios de Sistemas Computacionais: Sistemas Operacionais.

Docente: Prof. Dr. Tiago de Olivera

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Setembro de 2018

Resumo

O relatório em questão apresenta detalhes à respeito implementação de um Sistema Operacional que será utilizado em conjunto com ambos os projetos dos laboratórios de arquitetura e organização de computadores e de compiladores já cursados. Tais detalhes são referentes à gerenciamento de memória, gerenciamento de arquivos, escalonamento de processos e eventuais outros tópicos pertinentes ao tema e ao bom funcionamento do projeto final como um todo.

Palavras-chaves: Sistema, Operacional, Gerenciamento, Escalonamento, Processos.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Motivação

Cada vez mais presentes no dia-a-dia das pessoas, independente da área em que atuem ou trabalhem, computadores são de extrema importância para o funcionamento da sociedade atual como um todo, seja no âmbito de cadastro de serviços, no movimento de transações bancárias ou simplesmente como ferramenta de planejamento.

Seu funcionamento se dá com base em diversos fatores envolvendo elementos de ambos *hardware* e *software* para a execução de um determinado conjunto de instruções sobre um certo dado a fim de se obter um resultado específico. Em outras palavras, tais ações são tomadas a fim da execução de algoritmos, os quais envolvem ações como armazenamento, envio, recebimento e operação de dados, para os quais é necessária a criação de uma unidade de processamento, eventuais memórias, um módulo de entrada e saída, dentre outros.

Uma vez criadas tais unidades de *hardware*, há um programa específico responsável por garantir a comunicação entre elas e o usuário, o Sistema Operacional. Tal programa é gerado por meio de um compilador capaz de traduzir programas de uma linguagem de alto nível para uma de baixo nível, no caso, as instruções do processador.

Assim, o projeto descrito por este relatório visa realizar a integração do processador e do compilador para que assim seja possível, ao final, planejar e implementar o Sistema Operacional.

1.2 Objetivos

1.2.1 Geral

Planejar e desenvolver componentes que oferaçam suporte básico a um Sistema Operacional, os quais integrem o processador previamente desenvolvido e o compilador da linguagem *Cminus*.

1.2.2 Específico

Adapatar as partes do projeto e implementar estruturas de gerenciamento de processos, arquivos de entrada e saída e memória além de adicionar eventuais novos módulos pertinentes ao funcionamento do Sistema Operacional, como um disco rígido e uma BIOS responsável por garantir a inicialização do sistema.

2 Fundamentação Teórica

O Sistema Operacional de uma máquina é o *software* responsável por tratar a comunicação entre os procedimentos executados pelas ações dos usuários e seus elementos de *hardware*. Assim, sua principal função é garantir uma execução eficiente dos programas selecionados pelo(s) usuário(s) de maneira a garantir a integridade do *hardware* e seus dados. Além disso, deve ser capaz de prover funções que garantam acesso à eventuais periféricos de entrada e saída, gerenciamento do sistema de arquivos (criar, deletar, ler e modificar) e maneiras de proteção aos usuários (1).

2.1 Gerenciamento de Processos

Um Sistema Operacional, enquanto operante, realiza inúmeras tarefas, sendo muitas delas não visíveis e/ou perceptíveis ao usuário e desencadeadas com base na necessidade dos programas em execução na máquina. Tais programas, portanto, são chamados de processos, os quais são executados na maior parte das vezes um de cada vez pela CPU. O que de fato faz com que os usuários tenham a impressão de que o computador executa muitos deles ao mesmo tempo é o seu gerenciamento.

Gerenciamento de processos em sistemas operacionais é algo de extrema importância; é por meio dele que são utilizadas maneiras e algoritmos que determinam qual será o próximo processo a ser tratado pela CPU. Assim, formalmente, um processo é um programa que executa uma ação determinada e que pode de alguma forma ser controlado, seja pela aplicação que o criou, pelo próprio Sistema Operacional ou pelo usuário (1).

Uma vez que várias aplicações podem ser executadas ao mesmo tempo, um Sistema Operacional conta sempre com um escalonador de processos, responsável por determinar quais processos serão tratados pela CPU. Alguns dos mais comuns métodos utilizados pelos escalonamentos de processos são (1):

- ***First Come First Served (FCFS)***: Utiliza uma fila única de processos, de forma que o primeiro a chegar será o primeiro a ser atendido.

- ***Shortest Job First (SJF)***: Executa sempre o processo de menor tempo restante pertencente à fila de processos em espera.

- ***Round Robin (RR)***: Utiliza de uma fila circular para executar cada processo em espera por um tempo especificado chamado de *quantum*, de forma que caso um processo não consiga ser executado por completo, este sofre uma interrupção para ceder tempo de CPU para o próximo processo na fila de espera.

Os algoritmos citados acima juntamento com os outros tipos de algoritmos de escalonamento atuam de forma a lidar com diversos estados nos quais os processos podem estar, como pode ser visto na Figura 1.

Figura 1 – Estados de Processos

Fonte: Autor

2.2 Gerenciamento de Memória

Uma das principais atuações do Sistema Operacional frente à execução de processos de aplicações é a alocação, realocação e liberação de espaço de memória e seu gerenciamento. Para isso, existem diversas maneiras com as quais o Sistema Operacional pode manter controle sobre seus dados armazenados, sendo algumas delas (1):

- -**Segmentação**: Determina que a RAM seja dividida em blocos de tamanhos variados com base nos processos armazenados e que podem eventualmente ser realocados.

-**Partição**:A memória é dividida em blocos de tamanhos fixos ou variáveis que não podem ser realocados, o que muitas vezes pode gerar segmentação interna devido a um tamanho de bloco muito maior do que o necessário para o armazenamento de um eventual processo.

-**Paginação**: É feita a divisão da memória RAM em blocos de tamanho fixo chamados de páginas, de forma que os processos são divididos em páginas armazenadas na memória principal e em disco; caso um processo eventualmente precise de uma página que ainda não se encontra ne memória principal, o Sistema Operacional deverá realizar a troca (*swapping*) da página que está faltando com a correta presente em disco. Vale ressaltar que podem ainda haver várias maneiras diferentes de se definir um algoritmo de substituição de página, sendo uma delas, por exemplo, o algoritmo de conjunto de trabalho (*Workset*), que define um certo conjunto de páginas mais frequentemente referenciadas à memória

como um conjunto no qual elas devem ser buscadas para uma busca mais diferecionada (1).

2.3 Gerenciamento de Arquivos

O sistema de arquivos de um Sistema Operacional deve oferecer suporte à funções como criação e deleção de arquivos e diretórios juntamente com o gerenciamento de regiões livres de memória e o mapeamento de disco.

2.3.1 Organização Estrutural de Arquivos

Arquivos podem ser estruturados de diversas maneiras (1), sendo três das mais comuns:

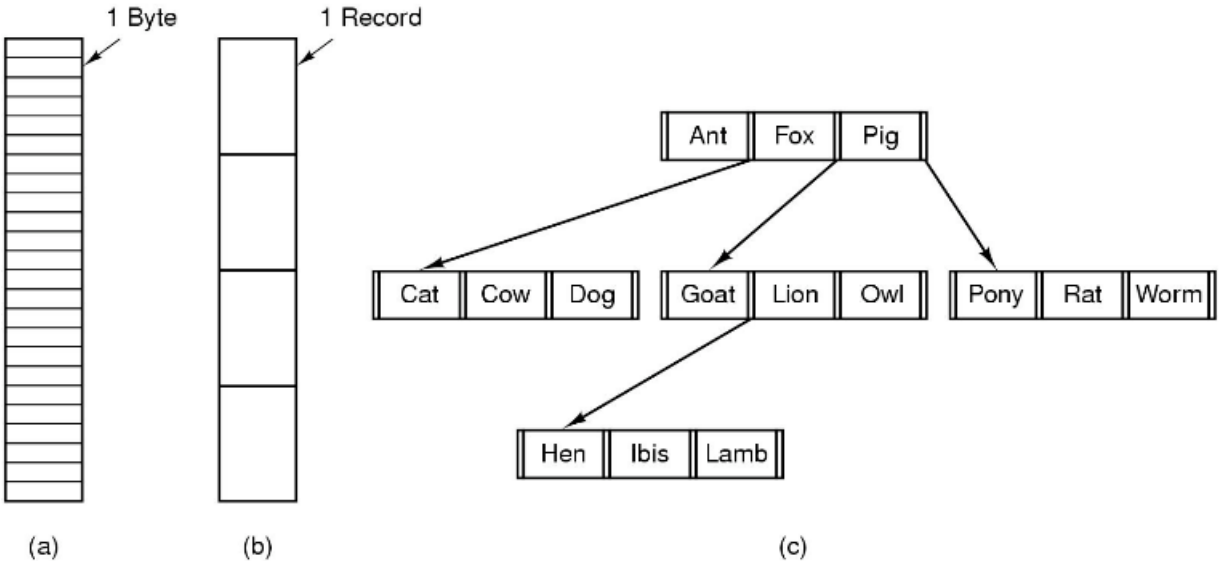
- **Sequência desestruturada de *bytes***: Maneira com a qual o Sistema Operacional não sabe exatamente o que o arquivo contêm ou simplesmente não dá importância ao seu conteúdo, todo significado e informação à respeito de tais arquivos deve ser evidenciado pelas aplicações em nível de usuário. Tal organização permite uma máxima flexibilidade, de forma que qualquer informação pode ser posta em um arquivo.

- **Sequência estruturada de registros**: Ao ser organizado desta maneira, o arquivo passa a ser uma sequência de registros de tamanho fixo, cada um com algum tipo de estrutura interna própria. Tal estrutura permite uma melhor organização com relação à operações de leitura e escrita sobre arquivos uma que vez que na primeira, basta retornar um de seus registros enquanto que na segunda basta sobrepor um já existente ou anexar um novo.

- **Sequência estruturada por índices**: Nesse tipo de organização, o arquivo é constituído de uma árvore de registros, não necessariamente todos do mesmo tamanho, de forma que cada um desses possui um índice ou campo-chave de posição fixa no registro. Assim, a árvore é organizada com base em tais chaves para que buscas por chave específicas se tornem mais rápidas e eficientes.

Na imagem Figura 2 podem ser vistos em a,b e c, respectivamente representações das estruturas citadas acima:

Figura 2 – Estruturas de Arquivos



Fonte: (1)

2.4 Operações de Entrada e Saída

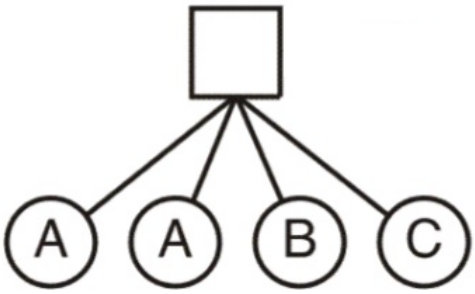
O sistema de arquivos do Sistema Operacional deve ser capaz de realizar um conjunto de ações que permita que tanto suas aplicações quanto seus usuários realizem operações de I/O, sendo elas, por exemplo, a entrada de um nome de uma pasta a ser criada por um usuário ou uma imagem mostrada em um periférico de vídeo por meio de um programa em execução.

2.5 Sistemas de Diretórios

Para que seja possível de se controlar os arquivos, os Sistemas Operacionais possuem também diretórios ou pastas, que em muitos tipos de sistemas também são arquivos.

- **Sistemas de Diretório Único**: Organização mais simples possível de um sistema de diretórios; manter todos os arquivos em um único diretório, chamado de diretório-raiz. Uma representação desse tipo de organização pode ser visto na Figura 3.

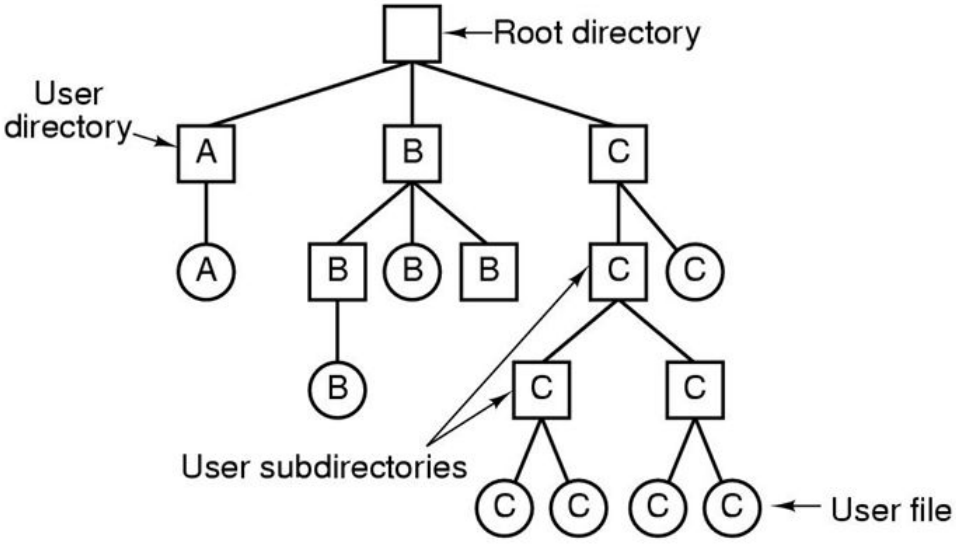
Figura 3 – Estrutura de Diretórios com Diretório Único



Fonte: (1)

- **Sistemas de Diretórios Hierárquicos**: Para uma melhor organização de arquivos em sistemas mais modernos com dezenas de milhares deles, a distribuição dos diretórios por meio de uma hierarquia ou árvore de diretórios se torna mais eficiente. Nela, há a distribuição de diretórios e sub-diretórios para uma melhor organização dos arquivos frente à diferentes aplicações de usuários do sistema. Um modelo dessa representação pode ser visto na Figura 4

Figura 4 – Estrutura de Diretórios Hierárquica



Fonte: (1)

2.6 Gerenciamento de Dispositivos de Entrada e Saída

O Sistema Operacional deve ser capaz de realizar o controle sobre dados relacionados à entrada e saída, de forma a oferecer *drivers* e *buffers* nos quais tais dados possam ser escritos, armazenados e então processados.

Assim, existem à princípio três diferentes maneiras de se realizar o controle de I/O de um sistema, sendo elas (1):

- **Entrada e Saída Programada (*Programmed I/O*)**: O processador é quem tem controle sobre os *buffers* nos quais serão realizadas a entrada e saída de dados, de forma que este também obtém controle sobre eventuais interrupções necessárias para que a ação requisitada ocorra corretamente.

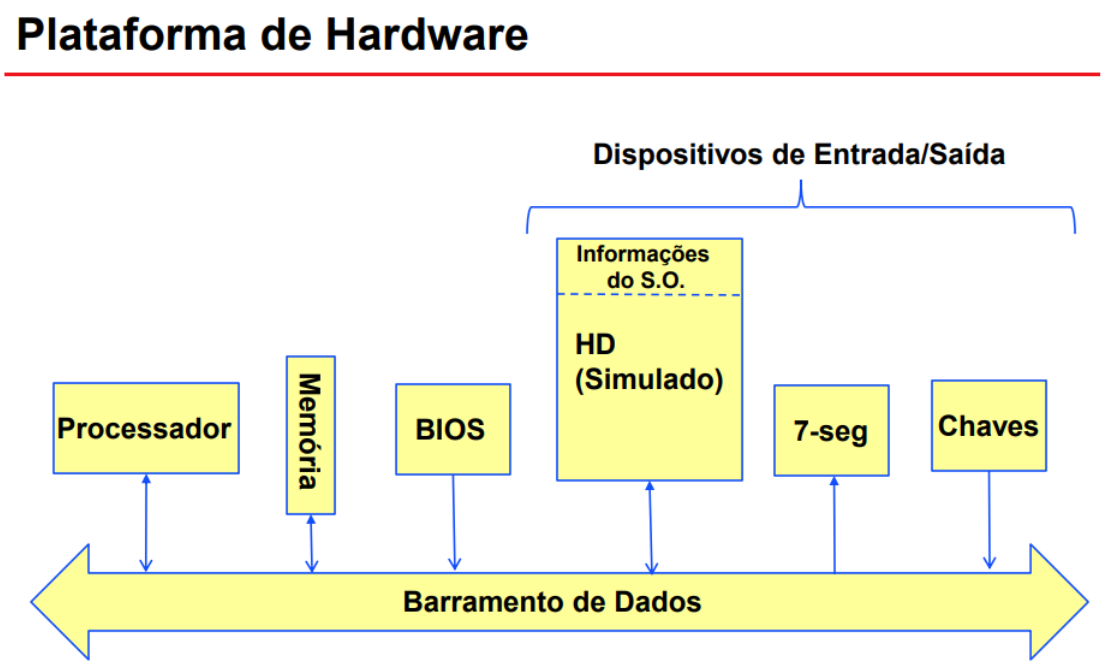
- **Entrada e Saída Utilizando Interrupção (*Interrupt Driven I/O*)**: Dessa forma, o processador é capaz de executar outras tarefas e processos enquanto espera que os dados fiquem prontos para leitura/escrita, de forma que uma interrupção é causada apenas quando já o estão e podem então de fato ser processados pela CPU.

- **Acesso Direto à Memória (*Direct Memory Access*)**: Utilizando DMA, a entrada e saída ocorre de maneira similar a quando utilizada *Programmed I/O*, contudo, os dispositivos que a estão executando possuem módulos que executam independentemente as rotinas para a conclusão da entrada e saída dos dados. Assim, após a execução de tais rotinas, o módulo de DMA (necessário para utilização de acesso direto à memória), envia um sinal para o processador que indica que a conexão com o dispositivo já pode ser encerrada e que o processo de I/O chegou ao fim.

2.7 Estruturação Física do Sistema Operacional (*Hardware*)

Para que atue como planejado realizando a correta comunicação das entradas do usuário com os módulos de processamento, o Sistema Operacional a ser implementado deve conter os elementos de *hardware* presentes em Figura 5

Figura 5 – Plataforma de *Hardware* do Sistema Operacional



Fonte: Slides de Aula

2.8 Linguagem de Descrição de *Hardware*

Para a implementação do Sistema Operacional, será utilizado uma arquitetura de processador previamente desenvolvido e um compilador, ambos os quais ainda serão detalhados mais à frente neste relatório. Com relaçao ao primeiro citado, para o seu desenvolvimento foi utilizada a linguagem Verilog (2), que nada mais é do que uma linguagem de descrição de *hardware* utilizada para o desenvolvimento e documentação de projetos de forma a permitir aos seus usuários a criação de vários níveis de abstração.

Ao ser utilizada, a linguagem se assimila à outras linguagens de programação como C/C++ e afins. Dessa forma, da mesma maneira que tais linguagens possuem variáveis tipadas como para valores inteiros, decimais, caractéres e variações, Verilog apresenta tipos relacionados à partes de circuitos digitais, tais como *wire*, *reg*, *input* e *output*.

A utilização desses tipos de variáveis permite a instanciação de *modules*, que representam partes de um projeto comum de circuitos digitais, e que quando ligados entre si, podem representar fielmente a simulação do funcionamento de sistemas computacionais complexos.

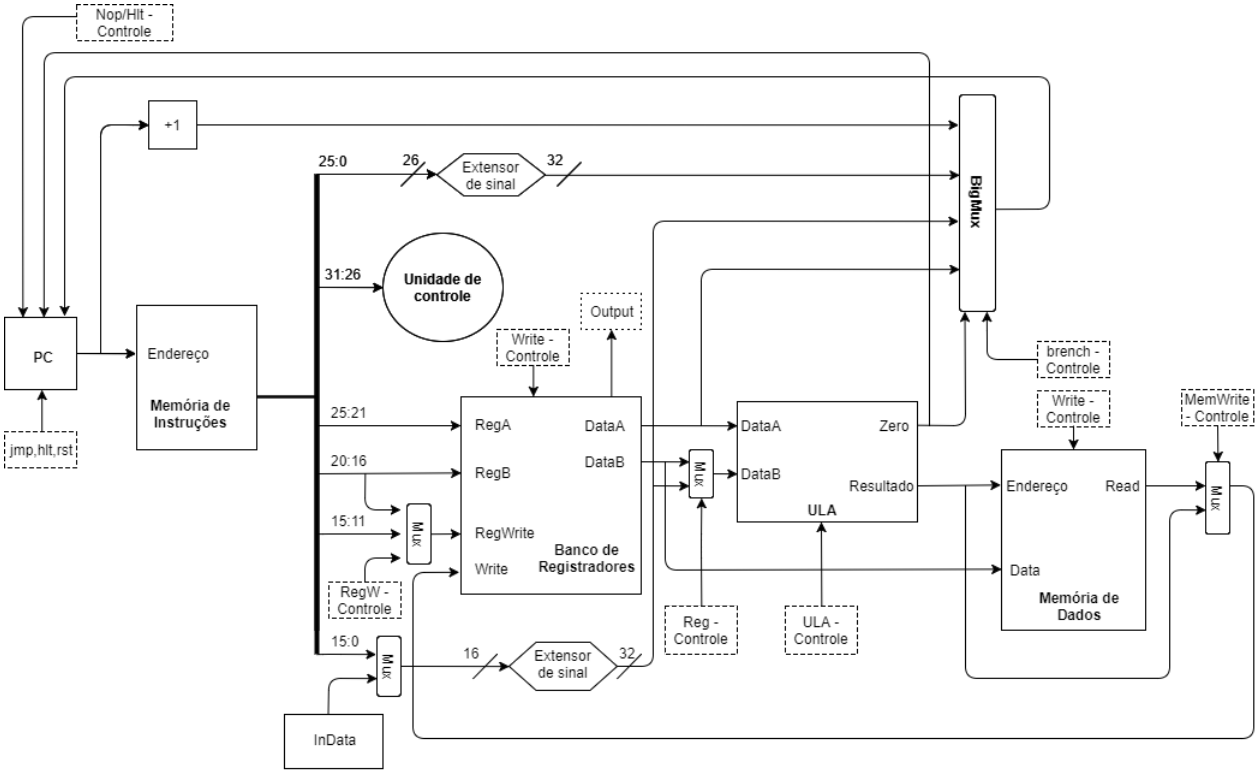
2.9 Arquitetura do Processador

O processador utilizado neste projeto (3) possui arquitetura inspirada na arquitetura MIPS (4), logo um conjunto de instruções RISC (*Reduced Instruction Set Computer*), uma vez que menos instruções tornam a organização de seus sinais de controle e módulos mais simplificada. Assim, segue uma explicação à respeito das especificidades de tal arquitetura.

2.9.1 Detalhes da Arquitetura

Como pode ser visto na Figura 6, o processador em questão possui em sua Unidade de Processamento um Contador de Programa (*Program Counter*), duas memórias, uma de instrução (*Instruction Memory*) e uma de dados (*Data Memory*), um Banco de Registradores (*Register Bank*) e uma Unidade Lógica e Aritmética (*Arithmetical Logic Unity*), fora eventuais extensores de sinal e multiplexadores.

Figura 6 – Caminho de Dados



Fonte: Autor

2.9.1.1 *Program Counter*

O contador de programa é o módulo responsável pelo armazenamento do índice da próxima instrução a ser executada, de maneira que é o módulo que garante o funcionamento de saltos normais (*jumps*) e condicionais.

2.9.1.2 *Instruction Memory*

A memória de instruções armazena todas as instruções do algoritmo que está sendo executado pelo processador. Nela há uma série de registros de 32 *bits*, cada um responsável pelo armazenamento de uma instrução, de forma que sua entrada é o índice desses registros a ser acessado (saída do *Program Counter*) e a saída é o registro armazenado em tal índice.

2.9.1.3 *Register Bank*

O banco de registradores da arquitetura em questão é utilizado em sua maior parte para o armazenamento de valores temporários pertinentes às operações em questão. Possuindo um total de 32 registradores de propósito geral, alguns deles são utilizados para fins específicos simplesmente por questão de uma maior facilidade com relação à organização dos dados quanto à utilização do compilador. Tais registradores são referentes ao endereço de retorno (*register addres*), ao ponteiro da pilha (*stack pointer*) e ao registrador de retorno de chamadas de funções.

2.9.1.4 *Arithmetic Logic Unity*

Responsável por tratar todas as possíveis operações lógicas e aritméticas do processador, a Unidade Lógica e Aritmética (ULA), atua de forma a receber dois dados (podendo eles estarem contidos no Banco de Registradores ou não) e realizar uma determinada operação com eles com base em um sinal de controle. Assim, sua saída pode ser utilizada como endereço de acesso da Memória de Dados ou simplesmente como um novo dado a ser armazenado no Banco de Registradores.

2.9.1.5 *Data Memory*

Por fim, o último dos componentes principais do caminho de dados da arquiteutra utilizada é a Memória de Dados. Nela são armazenados dados vindos do banco de registradores com base no cálculo de um endereço vindo da ULA. No âmbito da utilização do compilador, é nela que ficam armazenadas informações a respeito de todas as variáveis declaradas no decorrer do código. Vale ainda ressaltar que nesse módulo se encontra uma pilha implícita utilizada em contextos de parâmetros de funções, consecutivas chamadas de funções para o armazenamento dos endereços de retorno e recursão.

2.9.2 Tipos de Instruções

Por ser baseado na arquitetura MIPS, o processador utilizado possui três tipos de instrução: R, I e J.

- **Instruções do Tipo R:** tipo no qual se encontram todas as instruções que realizam operaçoes lógicas ou aritméticas com os dados dos registradores, tais como somas,

subtraçoes, deslocamentos e afins (5) (Tabela 1). Devido a isso são dividadas em *opcode*, RS, RT, RD, *shamt* e *funct*. O *opcode* é o responsável por diferenciar cada uma das instruções das outras, de forma que cada uma possua o seu *opcode* específico; RS e RT representam os endereços do banco de registradores dos quais serão retirados dados; RD possui o valor do endereço no qual serão escritos os dados resultantes da operação; *shamt* representa a quantidade de *bits* a serem deslocados caso seja necessário e por fim *funct* é utilizado para a diferenciação de instruções na ULA.

Tabela 1 – Formato R de Instrução

Tamanho (bits)	6	5	5	5	5	6
Campo	opcode	RS	RT	RD	shamt	funct
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 -0

Fonte: Autor

- **Intruções do Tipo I:** nesta categoria se encontram todas as instruções que realizam operações lógicas e aritméticas com dados de *offset*, vindos do campo imediato, executam saltos condicionais e realizam acesso à memória (Tabela 2). Neste caso a instrução é dividida, além do *opcode*, nos campos RS, RT e *offset*. O campo RS é o responsável por mostrar o valor do endereço do banco de registradores do qual será lido o dado a ser utilizado; RT representa o endereço do mesmo banco no qual será escrito o possível resultado da instrução e por fim o campo *offset*, ou imediato, carrega consigo um valor codificado que será usado diretamente na instrução, seja para possíveis saltos condicionais ou para operações aritméticas.

Tabela 2 – Formato I de Instrução

Tamanho (bits)	6	5	5	11
Campo	opcode	RS	RT	offset
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: Autor

- **Instrução do Tipo J:** tipo de instrução responsável pelas instruções de saltos (Tabela 3). Devido a isso sua divisão é a mais simples das instruções: *opcode* representa a identificação da instrução em si e todos os outros 26 *bits* são destinados ao endereço para o qual se quer saltar, o que garante uma distância muito maior para o salto do que se para o seu cálculo fosse utilizada a mesma quantidade de *bits* que em outros tipos de instrução.

Tabela 3 – Formato J de Instrução

Tamanho (bits)	6	26
Campo	opcode	address
Bits	31 - 26	25 - 0

Fonte: Autor

2.9.3 Conjunto de Instruções

Dadas as devidas explicações, o conjunto de instruções contido na arquitetura do processador utilizado pode ser visto em Tabela 4. Na tabela, os campos RS, RT e RD são referentes aos campos das instruções propriamente ditos, M representa a memória e STACK a pilha implícita citada.

Tabela 4 – Conjunto de Instruções

Instrução	**Tipo**	**Mnemônico**	***Opcode***	**Operação**
adição	R	add	000000	R[RD] ← R[RS] + R[RT]
subtração	R	sub	000001	R[RD] ← R[RS] - R[RT]
set on less than	R	slt	000110	if(R[RS] < R[RT]) R[RD] = 1
set on less or equal than	R	slt	011101	if(R[RS] <= R[RT]) R[RD] = 1
set on equal	R	slt	011110	if(R[RS] == R[RT]) R[RD] = 1
set on not equal	R	slt	011111	if(R[RS] != R[RT]) R[RD] = 1
deslocamento à esquerda	R	shfl	000111	R[RD] ← sl(*shamt*)R[RS]
deslocamento à direita	R	shfr	001000	R[RD] ← sr(*shamt*)R[RS]
NOT	R	not	001001	R[RT] ← !R[RS]
AND	R	and	001010	R[RD] ← R[RS] & R[RT]
OR	R	or	001011	R[RD] ← R[RS] | R[RT]
XOR	R	or	001100	R[RD] ← R[RS] ∧ R[RT]
multiplicação	R	mult	001101	R[RD] ← R[RS] x R[RT]
divisão	R	div	001110	R[RD] ← R[RS] / R[RT]
NOP	R	nop	010110	não realiza operação
HLT	R	hlt	010111	para a contagem do PC
adição com imediato	I	addi	000010	R[RT] ← R[RS] + IM
subtração com imediato	I	subi	000011	R[RT] ← R[RS] - IM
incremento	I	inc	000100	R[RT] ← R[RS] + 1
decremento	I	dec	000101	R[RT] ← R[RS] - 1
load	I	ld	001111	R[RT] ← M[IM + R[RS]]
load imediato	I	ldi	010000	R[RD] ← IM
store	I	str	010001	M[R[RS] + IM] ← R[RT]
branch on equal	I	beq	010010	if(R[RS] == R[RT]) PC ← IM
branch on not equal	I	bneq	010011	if(R[RS] != R[RT]) PC ← IM
jump to register	I	jmpr	010101	PC ← R[RS]
input	I	in	011000	R[RS] ← IN
output	I	out	011001	OUT ← R[RS]
push	I	push	011011	STACK[SP] ← R[RT]
pop	I	pop	011100	R[RT] ← STACK[SP]
jump	J	jmp	010100	PC ← IM
jump and link	J	jal	011010	R[31] ← PC + 1, PC ← IM

Fonte: Autor

2.10 Compilador para *Cminus*

Para este projeto foi também utilizado um compilador de linguagem *Cminus* (6), criado utilizando *Antlr4*. Essa linguagem se assemelha muito com a linguagem C já conhecida, com exceção apenas de alguns elementos nela não presentes, como por exemplo a inexistência de vetores bidimensionais e de alocação de memória, a falta de laços de

repetição do tipo *for*, inexistência de ponteiros e do tipo de variável de ponto flutuante (*float*), dentre outros. Vale ressaltar que este último é devido à limitações da arquitetura do processador.

Segue então alguns dos erros semânticos reconhecidos pelo compilador, todos, como já descrito, semelhantes à erros semânticos da linguagem de programação C, e ainda a gramática da linguagem *Cminus* presente na Figura 7.

- Variável não declarada;

- Atribuição de tipos de dados inválidos (*int* para *void*);

- Declaração com tipo inválido de variável;

- Variável já declarada;

- Função não declarada;

- Chamada de função com número errado de parâmetros;

- Função "*main*"não declarada;

- Declarações inválidas (variáveis com nomes de funções ou funções com nomes de variáveis);

Figura 7 – Gramática BNF da linguagem *Cminus*

programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [NUM] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID (params) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista } | { local-declarações } | { statement-lista } | { }
local-declarações → local-declarações var-declaração | var-declaração
statement-lista → statement-lista statement | statement
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if (expressão) statement | if (expressão) statement else statement
iteração-decl → while (expressão) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [expressão]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → (expressão) | var | ativação | NUM
ativação → ID (arg-lista) | ID ( )
arg-lista → arg-lista , expressão | expressão

Fonte: Autor

3 Desenvolvimento

Nesta seção será explicado na medida do possível como foram realizadas as alterações, tanto no compilador, quando na arquitetura de processador, ambos já descritos anteriormente, que permitiram que o sistema operacional desenvolvido pudesse funcionar da maneira como o faz.

3.1 Planejamento do Sistema Operacional

Primeiramente, no desenvolvimento desse projeto, foi tomada a decisão de se realizar a implementação do Sistema Operacional por meio de elementos de *software*, ou seja, foi realizado o desenvolvimento de um código na linguagem *cminus* referente ao SO, de forma que algumas eventuais alterações no compilador previamente desenvolvido se tornaram necessárias. Ainda assim, vale ressaltar que foram necessárias alterações em nível também de *hardware*, para adequar determinadas rotinas no processador à instruções que por sua vez poderiam ser chamadas por meio da utilização do compilador.

Sendo assim, o Sistema Operacional em questão foi desenvolvido de maneira dual; ou seja, quando se trata de regiões, tanto em memória quanto em registradores, respectivas à processos e ao próprio SO, estas foram explicitamente separadas, de forma que a arquitetura do processador foi alterada para dar suporte à isso. Tal fato será mais à frente explicado em maiores detalhes.

Uma breve prévia do seu funcionamento: O sistema se inicia pela BIOS, a qual é responsável por realizar alguns testes de verificação com o usuário. Após fazê-los, ela carrega o Sistema Operacional armazenado no HD na Memória de Instruções e troca o fluxo de execução para o da memória em questão. Assim, dá-se início, de fato ao SO. Este por sua vez funciona fazendo infinitas verificações com a entrada do usuário com relação ao que se deseja fazer; para isso são oferecidas duas opções, a de operação com arquivos (criar, renomear ou deletar) e a de operação com processos, que diz respeito à execução dos arquivos armazenados e possivelmente previamente manipulados. Sempre que uma dessas operações é realizada, o sistema retorna ao menu inicial, a partir do qual o usuário pode optar por realizar eventuais outras operações.

Com relação às rotinas que serão também mais à frentes explicadas e que deram suporte à atuação do programa Sistema Operacional, para que estas pudessem ser desencadeadas corretamente, foi necessária a criação de um novo tipo de instrução na arquitetura do processador, chamado neste caso de tipo X. Seguem, portanto, as instruções pertencentes à esse novo tipo.

Tabela 5 – Adição ao Conjunto de Instruções

Instrução	**Tipo**	**Mnemônico**	***Opcode***	**Operação**
BIOS to Memory	X	btm	100000	Muda o fluxo de instruções para a memória
load os	X	ldos	100010	MEM[OS][RD] ← HD[PROC=RS][RT]
move HD to Memory	X	mhdm	100011	MEM[PROC=RS][RD] ← HD[PROC=RS][RT]
move HD to reg	X	mdhr	100100	RD ← HD[PROC=RS][RT]
store HD	X	strhd	100101	HD[PROC=RS][RT] ← RD
swap process	X	sprc	100110	processo atual ← RS
concatenação	R	conc	100111	R[RD] ← {RS[15:0], RT[15:0]}
change message	X	cmsg	101000	display lcd ← IM
change write shift	X	cwsfh	101001	troca *shift* de escrita
change read shift	X	crsfh	101010	troca *shift* de leitura
get pc	X	getpc	101011	R[28] ← PROC PC
set pc	X	setpc	101100	PROC PC ← R[RS]
syscall	X	syscall	111110	R[28] ← INTERRUPTION
end program	X	endp	111111	sinaliza o fim do programa em execução

Fonte: Autor

Conforme se mostrar necessário no decorrer das próximas explicações, cada uma das instruções acima será devidamente detelhada.

3.2 Atuação do Sistema

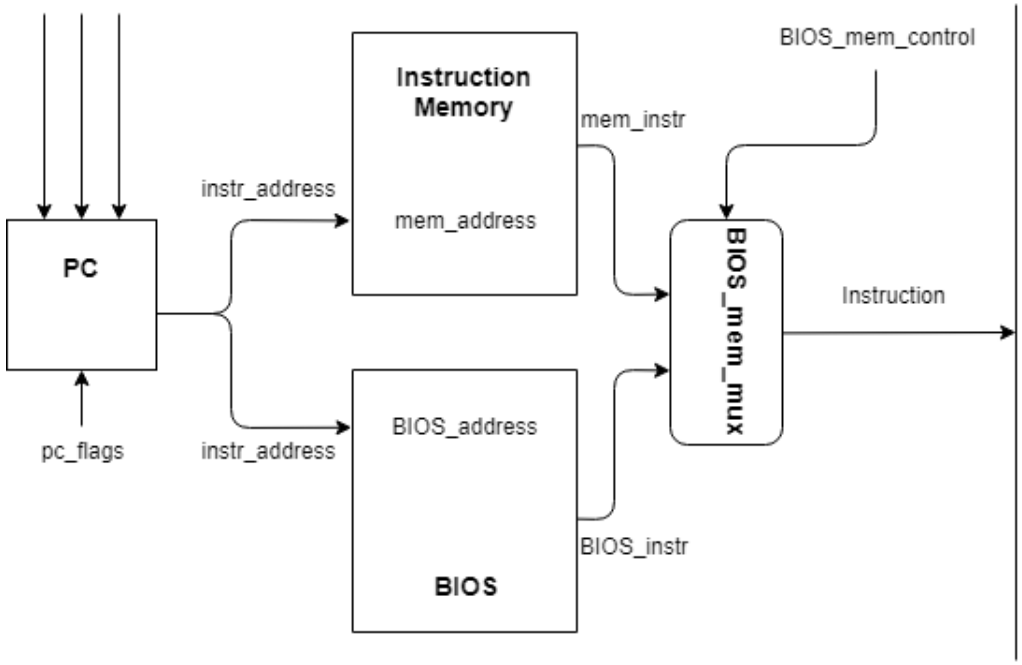
A partir de agora serão explicados os detalhes à respeito das alterações realizadas no projeto para o funcionamento do Sistema Operacional. Tais explicações serão feitas tentando ao máximo se assimilar ao breve funcionamento do SO descrito na seção anterior.

3.2.1 BIOS

A BIOS (*Basic Input Output System*) neste projeto foi implementada exatamente como uma memória de instruções, na qual são previamente armazenadas as instruções referentes ao código que desempenha o seu funcionamento. Nela as instruções são executadas assim como na Memória de Instruções fatídica; existe um endereço de entrada referente ao contador de programa de suas instruções e sua saída é uma instrução de 32 *bits* do sistema.

A diferenciação entre as instruções da BIOS e da própria Memória de Instruções foi feita por meio de um multiplexador localizado em sua saída, cujas entradas são as instruções saídas tanto da BIOS quanto da Memória de Instruções e a saída é a instrução referente à qual das duas estava sendo executada, como pode ser visto na figura abaixo.

Figura 8 – Funcionamento da BIOS



Fonte: Autor

Dessa forma o estado desse multiplexador era iniciado no valor referente à utilização da saída da BIOS, assim, quando a BIOS terminasse de rodar, a sua última instrução, (btm), seria responsável por alterar o estado desse multiplexador para que passasse a tomar como saída as instruções da Memória de Instruções e zerar o contador de programa, dando assim início ao sistema.

3.2.2 Mantedor de processos (*Process Keeper*)

Para um entendimento mais detalhado à respeito dos tópicos ainda por vir, é necessário o esclarecimento à respeito da organização de processos desse projeto. Por limitações de tamanho do HD e das regiões de memória, a arquitetura em questão suporta um total de 16 processos diferentes, de forma que seu respectivo número é armazenado em *hardware*, e tal armazenamento é feito por meio do módulo *Process Keeper*.

Este módulo apenas possui um registrador utilizado para armazenar o número do processo sendo executado, que, no caso, é um número de 0 a 15, totalizando assim os 16 processos. Contudo, para uma melhor localização e organização das regiões de endereçamento de cada um dos outros módulos, o processo de número 0 foi sempre rotulado como sendo o próprio Sistema Operacional. Com base nessa rotulação de processo foi possível de se organizar corretamente suas posições no HD tanto como seus registradores em momentos de troca de contexto.

Tendo esse fato em mente, a Memória de Instruções desse projeto, como dito

anteriormente, possui também duas regiões, uma referente ao próprio Sistema Operacional e a outra referente aos processos a serem executados. Assim, a movimentaçao dos dados do HD para a Memória de Instruções realizda pela BIOS armazenava os dados na região de memória das instruções do SO, as quais eram feitas por meio da instrução (ldos).

Dessa forma, a saída de tal memória, que armazena paralelamente os dados do Sistema Operacional e dos processos é simplesmente uma operação de seleção com base no número de processo do Mantedor de Processos: caso o processo seja igual a 0, toma-se como saída do módulo as instruções referentes ao SO, caso contrário, toma-se como saída as instruções dos processos.

Por fim, o número armazenado no Mantedor de Processos é alterado por meio de uma das novas instruções adicionadas ao ISA do projeto, no caso, (sprc), além de poder também ser alterado por meio de um sinal de interrupção, que será descrito mais à frente.

3.2.3 Memória de Instruções

Mesmo após a breve explicação do funcionamento da Memória de Instruções da seção acima, nem tudo sobre este módulo foi esclarecido.

Como dito anteriormente, a movimentação dos dados do HD para a região do Sistema Operacional dessa memória é realizada unica e exclusivamente durante a execução da BIOS por meio da instrução (ldos), contudo, para a movimentação de processos a serem executados do HD para a região dessa memória de processos, foi utilizada uma instrução muito similar, contudo, diferente, responsável por mover dados da memória para o HD (mhdm).

A unica real diferença entre ambas as instruções desencadeadas por funções diferentes é o fato de a primeira, (ldos) possuir uma *flag* em *hardware* responsável por indicar que a escrita dos dados deve ser feita na região do Sistema Operacional, enquando que a outra, (mhdm), não possui essa *flag*, de forma que quando é utilizada os dados não são movidos para a região do Sistema Operacional, mas sim para a região de processos da Memória de Instruções.

3.2.4 Contador de Programa (*Program Counter*)

Este módulo, anteriormente responsável apenas pelo armazenamento de um valor de contador de programa recebeu uma grande alteração com relação aos outros módulos da arquitetura do projeto.

Uma vez que ocorre simultaneamente pela Memória de Instruções o armazenamento de instruções referentes tanto ao Sistema Operacional quando aos processos, foi necessário também se adaptar o Contador de Programa para que este passasse a armazenar tanto a próxima instrução a ser realizada pelo próprio SO quando a próxima instrução a ser

realizada pelo próximo processo. Assim, mais uma vez com base no número do processo em *hardware* mantido pelo *Process Keeper*, a saída do Contador de programa era diferenciada da mesma maneira que a saída da Memória de Instruções.

Contudo, o funcionamento dos dois contadores de programa possui uma diferença; caso o número do processo em execução seja 0, ou seja, caso o Sistema Operacional esteja sendo executado, é possível por meio de duas instruções especiais se obter o valor armazenado no contador de programa de processos (getpc) e se alterar tal mesmo valor (setpc), caso este que ocorre em momentos de escalonamento de processos uma vez que processos diferentes possuem contadores de programas diferentes.

3.2.5 Endereçamentos e outros módulos

Uma vez esclarecidos os módulos que sofreram as maiores alterações no projeto, é importante se ressaltar a maneira com a qual foi realizado o endereçamento das regiões de memória e do disco rígido dos processos.

3.2.5.1 Disco Rígido e Memória de Instruções

Para este projeto foi utilizada a solução mais simples possível; cada programa, inclusive o próprio Sistema Operacional, possui um número de instruções limitado a 2048, que é, no caso, o tamanho máximo da região de cada processo dentro do HD e de cada uma das regiões da Memória de Instruções. Tal número foi escolhido pois é a menor potência de 2 grande o suficiente para armazenar o Sistema Operacional, de forma que tal número é utilizado para realizar um deslocamento no índice de acesso do endereço do HD, garantindo assim que a região certa será acessada.

Vale ressaltar que tal abordagem, obviamente, possui desvantagens, sendo uma delas um grande desperdício de memória, uma vez que programas em execução que não o Sistema Operacional mal chegam a ter um décimo de seu tamanho, mas ainda assim possuem 2048 posições de memória alocadas para si. Contudo, dentro do escopo do projeto dessa disciplina, tal desvantagem não proporcionou nenhum peso ou dificuldade no decorrer de seu desenvolvimento.

3.2.5.2 Memória de Dados

Para este módulo, de maneira similar ao HD, no qual cada processo possui uma região de 2048 posições, cada processo possui para si uma região de 252 posições, as quais, ao contrário do HD em que o deslocamento para tais posições é feito diretamente, devem ser especificadas "manualmente"durante a escrita e/ou posicionamento desses programas.

Em todos os casos com exceção do Sistema Operacional, as primeiras 32 posições dessas 252 são reservadas especificamente para os dados dos registradores de cada processo,

que são acessadas exclusivamente durante a troca de contexto.

Por fim, por possuirem regiões de memória próprias, cada programa também possui sua própria região de pilha, que neste caso foi limitada a 100 posições para cada processo. Assim, o processo 1 acessa a memória a partir do endereço 252 e a pilha a partir do endereço 100, o processo 2 acessa a memória a partir do endereço 512 e a pilha a partir do endereço 200, e assim por diante até o processo 15.

3.2.5.3 Banco de Registradores e Endereçador (*Address Setter*)

No caso do endereçamento no banco de registradores foi utilizada uma abordagem diferente e não muito intuitiva se comparada à utilizada nos outros módulos; um dobro no seu tamanho.

Normalmente, um banco de registradores de um processador similar à arquitetura MIPS possui 32 registradores de propósito geral, contudo, uma vez que há a necessidade de se diferenciar cada conjunto de 32 registradores de cada processo juntamente com os 32 registradores de acesso do próprio Sistema Operacional, a capacidade do banco foi aumentada para 64 registradores e foi criado um módulo auxiliar para o cálculo do endereço a ser acessado, sendo este o *Address Setter*.

Este módulo nada mais é do que uma máquina de estados que armazena os estados tanto do deslocamento de escrita quanto do deslocamento de leitura do banco. Tais estados podem ser individualmente alterados pelas instruções *change write shift* (cwsfh) e *change read shift* (crsfh), respectivamente. Assim, o que esses estados fazem é simplesmente *setar* um sinal que possui valores zero ou um, o qual habilita o deslocamento de 32 posições no banco de registradores.

Dessa forma, ao se reservar os 32 primeiros registradores do banco para o Sistema Operacional e os outros 32 para os processos, é possível se manter um encapsulamento com relação aos valores do SO e dos processos, sem que eles se sobrescrevam. Ainda, com a alteração nos deslocamentos de leitura e escrita, foi possível se fazer com que o Sistema Operacional possuísse acesso aos registradores dos processos, o que facilitou a implementação das chamadas de sistema (*syscalls*), que também serão melhor descritas à frente. Assim, ao se alterar o deslocamento de escrita, uma instrução de adição, que em seu corpo possuiria os valores especificados como:

$$add\ \$r4,\ \$r4,\ \$r5 \tag{3.1}$$

na verdade pôde realizar a operação:

$$add\ \$r(4+32),\ \$r4,\ \$r5 \tag{3.2}$$

Da mesma maneira, se alterando o deslocamento de leitura, uma instrução dada como:

$$add\ \$r5,\ \$r6,\ \$r7 \tag{3.3}$$

desencadearia na realizada a operação:

$$add\ \$r5,\ \$r(6+32),\ \$r(7+32) \tag{3.4}$$

Com a abordagem descrita acima bastou apenas se armazenar e carregar nas posições corretas da memória os registradores de cada processo durante momentos de troca de contexto utilizando tais deslocamentos para se garantir uma integridade de seus respectivos dados.

3.3 Gerenciamento de Arquivos

Assim como o próprio Sistema Operacional, os arquivos por ele tratado foram todos organizados via *software* por meio de vários vetores.

Tal organização foi feita de modo que o tamanho, em termos de instruções, de cada arquivo foi armazenado em um vetor de tamanhos de arquivos, assim como seus respectivos identificadores. Dessa forma foi possível se ter acesso ao identificador (nome) de cada arquivo ao se acessar o conteúdo de determinada posição no vetor ao mesmo tempo que ao seu respectivo número em *hardware* ao se acessar o próprio índice numérico no vetor.

Ao se utilizar tal abordagem, renomeação e deleção de arquivos se tornaram processos triviais; para renomear um arquivo, apenas é requisitado ao usuário um identificador de um arquivo, o qual é buscado no vetor de identificadores e alterado com base em outra entrada numérica do usuário, literalmente uma sobreposição de valores em um vetor, uma vez que o seu "real identificador", que é dado por sua posição no vetor e que é utilizado para os deslocamentos a nível de *hardware*, não é alterado.

Um processo similar é realizado durante a deleção de um arquivo, uma vez que é apenas buscado no vetor de identificadores o identificador de entrada do usuário; caso ele seja encontrado, a posição dele no vetor é armazenada e com base nela são modificados nessa mesma posição os vetores de tamanho de arquivo e de identificadores. O conteúdo da posição do vetor de tamanho de arquivos na posição armazenada é zerada, de forma que a referência para o arquivo no HD é apagada e o seu identificador é colocado em 99, o qual, nesse projeto, representa o identificador de posição vazia. Assim, simplesmente sobrepondo os dados referente ao arquivo requisitado em seus respectivos vetores com

valores específicos, o arquivo é "apagado", de forma que em sua posição em *hardware*, agora liberada, pode ser escrito outro arquivo, caso requisitado.

Por fim, o processo de criação de arquivos é um pouco mais refinado do que os outros dois citados. Para se criar um arquivo, primeiramente é necessario se encontrar uma posição no vetor de identificadores de arquivos que possua o conteúdo 99, que, como dito anteriormente, é o identificador de posição vazia. Uma vez encontrada essa posição, o valor dessa posição será usado, em *hardware*, para armazenar o novo arquivo a ser escrito.

Em seguida, é requisitado pelo SO uma entrada do usuário referente ao tamanho, em instruções, do arquivo. Para cada instrução são necessários outras duas entradas do usuário; uma referente aos primeiros *bits* da instrução e a outra referente aos últimos *bits*, ambas realizadas por meio dos *switches* da FPGA, os quais são concatenados por meio da instrução de concatenação (conc) e armazenados no HD em sua referente posição em *hardware* por meio da instrução *store hd* (strhd). Por fim, é requisitado uma entrada do usuário que será tida como identificador do novo arquivo criado.

Dessa maneira, por exemplo, caso hajam 9 arquivos já no HD, cada um, por padrão, acessando uma posição em *hardware* referente ao seu próprio identificador, ao se criar um novo arquivo este utilizará necessariamente a próxima posição em *hardware* livre, no caso, 10, mas ainda assim poderá ter um identificador numérico qualquer, como, por exemplo, 23.

3.4 Gerenciamento de Processos

Com relação à execução dos arquivos armazenados pelos mecanismos descritos anteriormente, o que, por definição, os torna processos, foi utilizado uma abordagem similiar ao algoritmo *Round Robin*, que utiliza de uma fila circular para a execução dos processos requisitados.

Para a execução bem sucedida desse algoritmo foi simulada por meio de um vetor na linguagem *cminus* uma fila circular responsável por armazenar o identificador de cada um dos processos a serem executados. Uma vez com essa estrutura, basicamente o algoritmo foi executado como a seguir:

É acessado o primeiro identificador da fila de processos; com base nele é acessado o seu índice em *hardware*; caso esse processo ainda não tenha terminado de executar, é atualizado o valor do seu contador de programa e seu contexto de registradores é movido para o banco de registradores na região de processos. Em seguida, o fluxo de execução é jogado para o do processo, o qual executará até ocorrer uma preempção. Uma vez ocorrida a preempção, um registrador especial armazena o seu valor, bom base no qual é possível se decidir o que fazer: terminar o processo, caso este tenha alcançado o seu fim, realizar um

tipo de chamada de sistema, caso requisitado, ou simplesmente ignorá-la, no caso de uma interrupção de *quantum*. Assim, após tais verificações a partir da interrupção ocorrida, o contexto de registradores do processo é armazenado, o próximo índice da fila a ser acessado é calculado e por fim é escolhido o identificador de processo em tal novo índice para se dar início ao processo novamente. O algoritmo termina de rodar quando todos os processos alcançam seu fim, que, no caso, é delimitado por uma instrução específica.

Assim, mesmo realizando comparações eventualmente desnecessárias, o algoritmo foi capaz de escalonar corretamente entre processsos de forma a executar todos, suas chamadas de sistema e finalizá-los, retornando o fluxo de execução ao sistema operacional.

3.4.1 Troca de contexto

As chamadas descritas na seção acima referentes à troca de contexto se mostraram suficientemente operantes para garantir a integridade dos dados de cada um dos processos a serem executados.

Dessa forma, seu funcionamento se deu com base nas já descritas instruções de alteração de deslocamente de leitura e de escrita no banco de registradores; o Sistema Operacional, ao determinar qual o número em *hardware* do próximo processo a ser executado, utilizava da alteração de tais deslocamentos juntamente com um deslocamento na região de memória do processo para carregar o valor de seus 32 registradores, localizados nas primeiras 32 posições da memória de tal região, para o banco. Uma vez realizada essa ação, ele então alterava o número do processo em *hardware* para o número do processos especifidado, dando assim um tempo de processamento ao processo em questão que logo retornaria ao SO quando ocorresse uma chamada de sistema e ou interrupção. Uma vez detectada a interrupção, o Sistema Operacional voltaria a ser executado, salvando assim o contador de programa do processo executado e usando os mesmos deslocamentos para o carregamento dos registradores para agora realizar o armazenamento destes, para que quando o próximo processo fosse executado, este fosse propriamente preparado de forma que não houvesse quebra na integridade dos dados dos processos.

4 Considerações Finais

Este projeto como um todo se demonstrou o mais desafiador dos projetos dos laboratórios do curso até o momento, uma vez que nele são necessárias por parte do aluno habilidades com relação ao conhecimento de compiladores, para eventuais ajustes; arquitetura de processadores, pelo mesmo motivo; sistemas operacionais, para um bom planejamento do SO a ser desenvolvido e suas eventuais intrincâncias tanto em *hardware* quanto em *software* e também de padrões de projeto, uma vez que a organização dos mecanismos desenvolvidos foi de suma importância para um bom funcionamento do projeto final.

Assim, vale ressaltar que um melhor aconselhamento e acompanhamento realizado com os alunos provavelmente traria resultados mais eficientes e com menos disparidades com relação à capacidade operacional dos diferentes projetos desenvolvidos e da carga necessária para realizá-los.

Por fim esse projeto deve agradecimentos à algumas pessoas, sendo elas os alunos Victor Renó Poglioni e Igor Luppi pelas longas conversas nos meses de desenvolvimento do projeto à respeito de eventuais soluções dos problemas que poderiam surgir durante sua implementação e, por último mas não menos importante, o aluno Gabriel Borin Takahashi, sem o qual este projeto não teria sido possível uma vez que a máquina na qual foram realizados os experimentos que comprovaram seu funcionamento foi por ele emprestada.

Referências

1 TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Citado 6 vezes nas páginas 8, 9, 10, 11, 12 e 13.

2 VERILOG.COM. *Verilog*. 2012. <http://www.verilog.com/>. [Online, acessado 5/05/2017]. Citado na página 14.

3 MOURA, B. *BMCORE1*. 2017. <https://github.com/BrunoBMoura/BMCORE1>. [Online, acessado 5/05/2018]. Citado na página 15.

4 BRITTON, R. *Introduction to the MIPS Architecture*. 2013. <http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/02-MIPSArchitecture.pdf>. [Online, acessado 5/05/2017]. Citado na página 15.

5 STALLINGS, W. *Computer Organization and Architecture*. 9th edition. ed. Upper Saddle River/New Jersey, EUA: Pearson, 2013. Citado na página 17.

6 MOURA, B. *bmcorepiler*. 2018. <https://github.com/BrunoBMoura/bm_core_piler>. [Online, acessado 5/09/2018]. Citado na página 18.