Universidade da Beira Interior
Departamento de Informática

Nº 1 - 2020: *Simulador de Gestão de Processos, Escalonamento e Gestão de Memória*

Elaborado por:

Grupo 3: Daniel Martins Mata, Miguel Ângelo Mota Santos e Gabriel Alexandre Araújo Ribeiro

Orientador:

Professor Doutor Paul Crocker
Professor Doutor Valderi Leithardt

20 de Junho de 2020

Conteúdo

Lista de Figuras

Capítulo

1

Introdução

1.1 Enquadramento

O desenvolvimento do projeto entitulado "Gestor de Processos"surgiu no âmbito da Unidade Curricular de Sistemas Operativos em sintonia com o conteúdo programático abordado nas aulas teóricas e práticas. A partir do mesmo é possível aplicar todos os conhecimentos acerca de um sistema operativo e o seu funcionamento.
A **linguagem de programação escolhida** para o desenvolvimento da aplicação foi ***C***, uma vez que foi a linguagem escolhida pelo Docente da U.C para a realização das Fichas Práticas e acima de tudo, todos nós no grupo temos um maior "à vontade"face às outras hipóteses apresentadas.

Link para o repositório do trabalho:
`https://gitlab.com/so-groupDGM/so-project`

1.2 Motivação

O projeto de construção do Gestor de Processos tem um enorme interesse no âmbito da Unidade Curricular de Sistemas Operativos, uma vez que consegue desenvolver a capacidade de compreensão deste funcionamento. Para além da componente pedagógica para a disciplina, este projeto contribui para o nosso

enriquecimento geral do ato de programar e desenvolver aplicações para um fim particular. Acima de tudo, com a elaboração do trabalho pretendemos fomentar o nosso conhecimento.

1.3 Objetivos

O presente trabalho tem como objetivo a criação de um programa que simule o funcionamento do sistema operativo nas suas ações de gestão e execução de processos. Mais em concreto, a aplicação deve incluir o controlo do escalonamento do CPU, criação de processos, assim como a comutação de contexto. Além disso, a aplicação implementará numa segunda fase, a gestão de memória do processador.

1.4 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento.

2. O segundo capítulo – **Contextualização** – descreve os conceitos mais importantes no âmbito deste projeto, bem como as tecnologias utilizadas durante o desenvolvimento da aplicação.

3. O terceiro capítulo **Explicação do Programa** descreve em detalhe todo o trabalho de cada uma das funções efetuada e a forma como o faz.

4. O quarto capítulo – **Testes à implementação** mostra todos os testes feitos à aplicação e apresenta alguns exemplos dos resultados obtidos.

5. O quinto e último capítulo – **Conclusões e Trabalho Futuro** apresenta as conclusões principais que tirámos da realização da primeira etapa desta aplicação de Gestão de Processos. Além disso, neste capítulo é apresentado o trabalho a ser feito para a próxima parte do projeto.

Capítulo

2

Contextualização

2.1 Introdução

Neste capítulo intermédio será sintetizado o conhecimento teórico necessário para o desenvolvimento do projeto e de todas as suas diversas funcionalidades implementadas até ao momento da submissão do primeiro *Milestone*.

Toda a informação presente neste desenvolvimento foi baseada nas apresentações das aulas teóricas da Unidade Curricular [2] [3].

2.2 Desenvolvimento

Os conceitos principais à compreensão de todo o trabalho são:

- **Processo:** um processo representa um programa em execução. A noção de programa não é suficiente para representar um processo, uma vez que este possuí muito mais além da secção de texto. O *Program Counter* (PC), os registos do processador, a *Stack* do programa em exe-

cução, assim como todas as variáveis que compõem a estrutura de cada processo. Esta noção permite a partilha de cópias de um programa por vários utilizadores, obviamente com secções de dados diferentes.

- **PCB:** bloco responsável por representar um processo no sistema operativo e guardar todas as suas informações. Esta representação lógica do processo pode ser vista como uma estrutura de dados que se divide nas seguintes partes:
  - Estado do processo
  - *Program Counter*
  - Registos do CPU
  - Prioridade de escalonamento
  - Informação de gestão de memória
  - *Process ID*
  - *Progenitor Process ID*
  - Entre outros
- **Comutação de contexto:** No CPU existe sempre um processo a correr, de modo a otimizar a execução do mesmo. Como no processador só um processo pode fazer uma instrução de I/O de cada vez, isso obriga à existência da troca entre processos em execução - **Comutação de Contexto**, à existência de um processo numa fila de acordo com o seu estado e ao **escalonamento** para a escolha nas transições entre essas filas.
- **Modelo de 5 estados:** Um processo no processador pode estar num dos seguintes estados:
  - *New*: quando o processo está a ser criado;
  - *Running*: no momento em que se executam as instruções de um único processo;

  - *Waiting*: enquanto o processo espera pela sua vez paa voltar a possuir lugar no CPU;
  - *Ready*: processo espera para ser atribuído ao CPU;
  - *Terminated*: estado final da execução.

- **Escalonamento:** Um novo processo é inicialmente colocado na *ready queue*, onde permanece até ser **despachado** para o CPU.
  - Não-Preemptivo: O processo ocupa o CPU de forma interrupta até terminar ou passar ao estado *Waiting*. O grau de multiprogramação fica comprometido.
  - Preemptivo: O escalonamento preemptivo inclui a utilização de um sistema de *Interrupt* que suspende o processo para de seguida executar outro. Perigos eminentes de erros de sincronização de memória partilhada.

  O processo pode ser removido do CPU caso ocorra uma interrupção do clock passando para a *Ready Queue*. Qualquer recurso pode estar numa fila de escalonamento que varia em função da migração que ocorre nestas. As decisões de escalonamento ocorrem em diversos momentos:
  - Comutação do estado ***Running*** para o estado ***Waiting*** através da *System Call - Wait* ligado a uma ação de I/O;
  - Comutação de estado ***Waiting*** para ***Ready*** derivado a um ***interrupt***;
  - Comutação de ***Waiting*** para ***Ready*** através da terminação de I/O;
  - Terminação de um processo.

- **Escalonadores:**
  - Escalonador de curto-prazo: faz a seleção dos processos que devem ser executados e reserva o CPU. Invocação muito frequente, baseado no conceito de **Swapping** que melhora a mistura de processos.
  - Escalonador de médio-prazo: Aloca memória para os processos. Chamado com menos frequência.
  - Escalonador de longo-prazo: Escolhe os processos a serem levados para a **Ready Queue**, assim controla o grau de multiprogramação. Pouco frequente.
- **Operações sobre processos:**
  - Criação de processos: O processo progenitor cria novos processos filhos através da *System Call* - **Fork()**. Pai e filho podem executar concorrentemente, filho após pai, assim como também podem partilhar recursos ou não.
  - Terminação de processos: Um processo termina após o término da execução da sua última instrução. Além disso, o seu pai pode terminar a execução do filho através da chamada ao sistema - ***Abort***.
- **Algoritmos de escalonamento**
  - First-Come First-served: este é o algoritmo mais simples, no qual os processos vão sendo selecionados pela **ordem de chegada** à *Ready Queue*.
    Assim que o CPU termina uma tarefa, o primeiro processo da *Ready Queue*, isto é, o que ocupa a cabeça da fila, é depachado para o CPU. Assim os processos são executados um após o outro segundo a sua ordem, no entanto este algoritmo possui muitas desvantagens, como por exemplo um tempo de espera muito elevado, o que não é adequado a sistemas *Real Time*.

- **Algoritmos de gestão de memória**
  - First-Fit: O processo vai ser alocado em memória no primero espaço livre encontrado que seja suficientemente grande para o processo. Se não houver um espaço em que o tamanho seja suficiente vai ocorrer um erro. Se for encontrado um espaço livre de tamanho suficiente, o processo é alocado, e qualquer espaço que sobre torna-se num espaço livre mais pequeno.
  - Next-Fit: É uma versão modificada do **First-Fit**, começa da mesma maneira quando está à procura de um espaço livre, mas quando é chamado uma segunda vez, começa de onde tinha ficado, em vez de começar do inicio.
  - Best-Fit: A memória vai ser percorrida na sua totalidade, e vai procurar pelo espaço livre mais pequeno que seja suficiente para o processo que queremos alocar.
  - Worst-Fit: A memória vai ser percorrida na sua totalidade, e vai procurar pelo maior espaço livre e vai alocar o processo nesse espaço.

2.3 Conclusões

Após a abordagem e estudo do conteúdo programático da Unidade Curricular necessário à realização do projeto, estão reunidas todas as condições para se proceder à explicação detalhada da implementação das funcionalidades.

Capítulo

3

Explicação do Programa

3.1 Introdução

Neste capítulo procedemos à explicação com todo o detalhe partindo da justificação da separação nos diferentes ficheiros de código fonte e bibliotecas. Numa segunda parte do capítulo passaremos a explicar a sintaxe e o conteúdo de cada função, e por fim a forma como todas as funções do programa se relacionam na função *Main*.

3.2 Ficheiros fonte e bibliotecas

A aplicação está dividida em **3 ficheiros de código fonte *c*** (extensão *.c*): main.c, gestorFun.c e queue.c; 1 bibliotecas (extensão *.h*): gestor.h, um gestor de compilação (*Makefile*), 2 ficheiros com extensão *.txt*:

3.2.1 Gestor.h

Include's e Define's

No início do ficheiro gestor.h são incluídas todas as **bibliotecas usadas na aplicação**. Logo de seguida são definidas algumas **constantes globais**, como o tempo máximo que o gestor pode executar ("GESTORLIMIT") e o tempo máximo que um processo pode executar ("TIMELIMIT"). Esta última constante representa por consequência o número máximo de instruções possível num processo, visto que apenas é executada uma instrução por unidade de tempo. As constantes **TRUE e**

FALSE provêm da implementação da **QUEUE**.

Além das constantes acima mencionadas, também optámos por acrescentar as seguintes: **WAITING(2), RUNNING(1), TERMINATED(3), READY(0)**, que substituem mais facilmente os estados dos processos através de inteiros.

Estruturas de Dados

As estruturas de dados que escolhemos para representar as variáveis do nosso programa são as seguintes: **INSTRUCTION, PCB e QUEUE**.

A primeira **(INSTRUCTION)** tem como objetivo guardar as informações relativas a cada instruçãos do processo, isto é, o char representativo, o valor para efetuar a operação e o nome de um processo, que apenas é utilizado no caso da isntrução ser um *exec()*.

```
typedef struct instruction{
    char instr;     //instrução
    int n;          //valor para a variável
    char nome[15];  //nome do processo - nome em caso de ser um exec ou NULL caso contrário
}INSTRUCTION;
```

Figura 3.1: INSTRUCTION

A segunda **(PCB)** pretende representar o Process Control Block do processo, por isso guarda toda a informação característica deste, desde o nome do processo até às suas instruções. De destacar o campo *instr* do tipo de dados **INSTRUCTION** como sendo um *array* de instruções.

```c
typedef struct Node_t{
    char nome[50];    //nome do processo
    int var;          //valor da variável
    int pid;          //identificador do processo
    int ppid;         //identificador do pai do processo
    int start;        //momentos que se deve inserir o programa na ready queue
    int priori;       //prioridade de execucao 1 > 2 > 3
    int tempVida;     //tempo que tem de ser executado
    int PC;           //próxima instrução a executar
    int valStInstrs;  //valor da primeira instrução
    int numInstr;     //número de instruções
    INSTRUCTION *instr;//array as instruções do processo
    int estadoProc;   //estado de execução do processo
    int openFiles;    //º de ficheiros abertos
    struct Node_t *prev;
} PCB;//controlar cada processo
```

Figura 3.2: PCB

A estrutura de dados **(QUEUE)** é uma estrutura de dados clássica da programação, cuja implementação retirámos de [1]. A implementação do autor referenciado contém funções para contruir, destruir, empilhar, desempilhar e verificar se a pilha está vazia.

```c
//fila para guardar processos
typedef struct Queue {
    PCB *head;
    PCB *tail;
    int size;
    int limit;
} QUEUE;
```

Figura 3.3: QUEUE

Por fim, a variável **memory** do tipo **INSTRUCTION** guarda as instruções dos processos que estão a correr até um preciso momento.

A estrutura de dados**(NODO)** é uma estrutura de dados simples que vai ser usada para a alocação de memória.

```
typedef struct Nodo {
    int pid;
    struct Nodo* next;
}NODO;
```

Figura 3.4: NODO

A estrutura de dados**(BEST)** é uma estrutura de dados simples que vai ser usada para definir o **NODO** mínimo.

```
typedef struct best {
    int size;
    struct NODO* head;
}BEST;
```

Figura 3.5: BEST

Protótipos de funções

Finalmente, no ficheiro gestor.h surgem as declarações dos protótipos das funções utlizadas em todo o programa Abaixo listamos os **protótipos** respetivos às funções sobre **QUEUE** da referência [1]:

- QUEUE *ConstructQueue(int limit);
- void DestructQueue(QUEUE *queue);
- int Enqueue(QUEUE *pQueue, PCB *item);
- PCB *Dequeue(QUEUE *pQueue);

- int isEmpty(QUEUE* pQueue);

- **As funções que desenvolvemos na aplicação são as seguintes:**
- int **countInstr**(char* nome): devolve o número de insruções de um processo passado como parâmetro;
- QUEUE ***readPrograms**(): lê os nomes dos programas que constam no ficheiro plan.txt e o respetivo tempo de entrada para a fila de *Ready*;
- int **countControl**(): conta o número de comandos de controlo que estão no ficheiro control.txt;
- INSTRUCTION* **lerIntr**(char *nome): Lê todas as instruções presentes nos ficheiros relativos a um processo em particular;
- PCB* **addRunning**(PCB* proc, int *i): adiciona o processo ao array Memory, ou seja, coloca as instruções em execução;
- PCB* **FCFS**(QUEUE* Prontos, int *time, int *index, QUEUE* Terminated): controla o escalonamento do tipo *First Come First Served*;
- PCB* **execInstr**(PCB* proc, int *i): executa as instruções de um processo a partir do array memory.
- void **report**(int *tempo, QUEUE* Bloqueados, QUEUE* Prontos, QUEUE* Terminados): imprime o estado atual do sistema no stdout.
- int **deallocate_mem**(int process_id, NODO* auxMem): retira processos da memória.
- int **fragment_count**(): serve para contar a quantidade de furos de 1 ou 2 unidades de tamanho.
- NODO* **NewNode**(int pid): cria um novo nodos.
- NODO* **LastNode**(NODO *head):
- int **countNodes**(): conta os nodos existentes.
- void **display**(NODO *memMam);
- int **alocate_units_mem_first**(int process_id, int num_units, NODO* auxMem): Aloca processos na memória usando o algoritmo First-Fit.
- int **alocate_units_mem_next**(int process_id, int num_units, NODO* auxMem): Aloca processos na memória usando o algoritmo Next-Fit.

- int **alocate_units_mem_best**(int process_id, int num_units, NODO* auxMem): Aloca processos na memória usando o algoritmo Best-Fit.
- int **alocate_units_mem_worst**(int process_id, int num_units, NODO* auxMem): Aloca processos na memória usando o algoritmo Worst-Fit.
- void **displayQueue**(QUEUE* Prontos): Mostra a fila Prontos.
- int **random_int**(int min, int max): devolve um inteiro entre o min e max dados.
- PCB* **SJFS**(QUEUE* Prontos, int *index): Algoritmo do Shortest Job First.
- PCB* **Prioridades**(QUEUE* Prontos, int *index): Algoritmo do escalonamento por Prioridades.
- NODO* **findFirstPid**(NODO* head, int pid): Devolve o first pid.
- BEST* **NewNodeBest**(int size):Cria/inicializa um novo node best.
- int**mediaFragmentosExternos**(NODO* auxMemSol):Devolve o número médio de fragmentos externos.
- int **tempoAlocacaoMedio**(int nodos, int num_alocs):Devolve o número médio de nós atravessados nas listas durante a alocação.
- void **gerarSolicitacoes**(int numSolic, NODO* auxMemSol): Gera solicitações usando a componente geração de solicitações, de forma aleatória e chama as respectivas funções de alocação/dealocação de memória. Além disso atualiza os parâmetros de desempenho. **A componente de relatório estatístico está incorporada nesta função.**
- int **somaVetor**(int *v, int tam):Retorna a soma de todos os valores de um vetor de inteiros.

3.2.2 gestorFun.c

Neste ficheiro constam os corpos das funções desenvolvidas no trabalho.

– int **countIntr**(char* nome) :
O objetivo principal é servir de auxiliar para as outras funções no que diz respeito a alocação de espaço em estruturas de dados, em particular em **readPrograms**. Nesta função é

aberto o ficheiro do qual se pretende ler as instruções e de seguida, verifica-se a abertura e no final, num ciclo, incrementa-se uma variável *count* cada vez que ocorre um *new line*.

- INSTRUCTION* **lerIntr**(char *nome) :
  o ficheiro do processo é aberto e o sucesso da sua abertura é verificado. Chama-se a **função acima** implementada para **contar o nº de instruções** de forma a alocar um **array do tipo INSTRUCTION (Instr)** para guardar as informações lidas.
  Num ciclo e enquanto se ler um *char* seguido de um *int* até ao final do ficheiro, o inteiro será atribuído ao campo **n** e o *char* atribuído ao campo *instr*, na **posição k** do array, que incrementa a cada iteração.
  Por fim será devolvido o array **Instr** com todas as instruções do ficheiro. Esta função será embutida na seguinte de forma a que sejam lidas as instruções de cada processo.

- QUEUE* **readPrograms**() :
  esta função vai abrir o ficheiro plan.txt em modo leitura, confirma se a abertura foi bem sucedida. São declaradas variáveis: (**Programas**) para guardar uma fila dos programas dados inicialmente ao gestor de processos, **nome**), uma string para guardar o nome do programa e **start** para o tempo em que um processo entra para a **fila de prontos**. A variável do tipo FILE guarda o apontador para o ficheiro em causa. Enquanto a leitura de uma string seguida de um inteiro for bem sucedida - **ciclo while com a condição da função fscanf devolver 2**. Dentro do ciclo alocamos uma variável **proc** do tipo PCB, para que seja possível atribuir-lhe as informações lidas do ficheiro. Isto é, ao campo proc->start será atribuído o valor de start e o proc->instr irá receber o retorno da função **lerInstr**. Aproveitamos para atribuir ao campo **proc->numInstr** o retorno da função **countInstr**. Agora que o proc possui todas as informações necessárias será empilhado numa fila inicial - Programas. O ponteiro para o ficheiro é fechado e a **função devolve a fila de Programas fornecidas ao gestor**.

- int **countControl**() :
  esta função conta o número de **comandos do ficheiro control.txt** baseado no **número de linhas**, ao considerar um co-

mando por linha. A variável count é inicializada a 0 e incrementada cada vez que surge um *newline*. A função tem o propósito de ajudar na criação de um array de comandos e para controlar os comandos que já foram executados.

- PCB* **execInstr**(PCB* proc, int*i, int*t, QUEUE* Blocked, PCB* Terminated, QUEUE* Prontos) :
  a função recebe como parâmetro um processo a ser executado, o indíce da primeira instrução a ser executada, assim como um apontador para uma variável de tempo e as filas respetivas aos diferentes estados dos processos para que os processos posssam ser transferidos entre elas. Para que seja possível a execução, criámos **2 variáveis** para **guardar a instrução - instr** e o respetivo **valor** – **num** a surtir alterações em **proc->var**. Numa série de **if's** encadeados verificamos a igualdade da variável letra com os vários tipos de instruções presentes no ficheiro de cada processo.
  Nas operações aritméticas básicas apenas será efetuada a alteração direta ao valor da variável global do proc – **proc->var**. Cada vez que uma instrução é executada, o index será incrementado, assim como a variável tempo, visto que é executada uma instrução por unidade de tempo. Por fim, a função devolve o processo com os seus campos atualizados.

- PCB* **addRunning**(PCB* proc, int *i) :
  Esta função recebe como parâmetros, o processo **proc** que passará do estado "Ready"para o estado "Running", e o respectivo indíce **i** no array **memory**. Já dentro da função, criámos uma variável **k** e inicializámo-la como **0**,sendo esse o indíce da primeira instrução do processo. Após a inicialização das variáveis ,verifica-se se o processo que foi recebido como parâmetro está no estado "READY"(variável **estadoProc** na estrutura de dados). Se estiver, mostra-se uma mensagem a informar o utilizador que o processo está no estado de "Running". A partir desse momento, inicializa-se uma variável auxiliar com o valor para o qual a variável **i** aponta. Essa variável, a qual chamámos de **aux** tem o seu uso no ciclo **while** como condição de paragem, onde **i** será menor que **aux** mais o número de instruções no processo **proc**. Também como condição de paragem, **k** tem de ser menor que o numero de instru-

ções. Dentro do ciclo adiciona-se a cada instância **i** no array **memory** a instrução **k** do processo **proc**, e no fim incrementa-se ambas as instâncias de forma a percorrer ambos os arrays. Após o ciclo, altera-se o estado do processo(**estadoProc** para "RUNNING") e retornamos o processo.

- PCB* FCFS(QUEUE* **Prontos**, int *time, int *pidPr, int *index, QUEUE* Terminated) :
  Esta função recebe como parâmetros as filas **Prontos** e **Terminated** para que o processo possa ser transferido entre elas, um apontador para o inteiro **index** que será o indíce passado para a chamada da função **addRunning**, e um apontador para o inteiro **time** que será o tempo que o processo leva a executar. Começamos a função ao inicializar uma variável chamada **procFinal** que será o processo que vai executar. Depois de todas as variáveis estarem inicializadas, verificamos se a fila **Prontos** está ou não vazia com o uso de um **if** e da função auxiliar **isEmpty**. Se a fila estiver vazia a funçao acaba e retorna o processo como estava. Se a fila não estiver vazia inicializamos uma nova variável chamada **startingTime**(sendo esse o momento em que o processo começa a correr), a qual igualamos ao apontador **time**. Após isso, tiramos o primeiro processo da fila **Prontos**, atribuimo-lo à variável **procFinal**, alteramos o seu estado para "Running"e adicionamo-lo ao array **memory** através da chamada da função auxiliar que criámos anteriormente chamada **addRunning**, a qual recebe como parâmetros o **procFinal** e o dito **index**. Para finalizar, verificamos no momento em que o processo terminou a sua execução, com outro **if**, se o tempo que o processo demorou a executar é igual ao seu tempo de vida mais o tempo do instante em que o mesmo começou. Se se verificar, alteramos o **estadoProc** para **TERMINATED** e com a ajuda da função auxiliar **Enqueue**, pomos o processo na fila de "Terminated"oficialmente retirando o processo do estado "Running". A função retorna então o **procFinal**.
  No final da função é atribuído ao campo **proc->pid** o valor do apontador **pidPr**.

- int **deallocate_mem**(int process_id, NODO* auxMem):
  Esta função recebe como parâmetros o **id** do processo que

queremos dealocar e a estrutura de dados da memória. Esta função irá percorrer a estrutura de dados até encontrar o **id** de que estamos á procura, ao encontrar o **id** certo, iguala o **pid** da memória a **-1**

- int **alocate_units_mem_first**(int process_id, int num_units, NODO* auxMem):
  Esta função recebe como parâmetros o **id** do processo que queremos alocar em memória, o **num_units**, ou seja, o tamanho que o processo vai ocupar e a estrutura de dados da memória. Vamos criar um NODO auxiliar**(NODO *aux = auxMem)** que nos permitirá percorrer a lista. Com o **aux** vamos percorrer a lista, se o nodo estiver vazio vamos continuar a percorrer enquanto houver nodos vazios, sempre que encontramos um nodo vazio a variável **count** aumenta em um. Depois com o **aux** vamos ver se existem nodos vazios suficientes, se forem nodos suficientes vai preencher os nodos necessários.

- int **alocate_units_mem_best**(int process_id, int num_units, NODO* auxMem):
  Esta função recebe como parâmetros o **id** do processo que queremos alocar em memória, o **num_units**, ou seja, o tamanho que o processo vai ocupar e a estrutura de dados da memória. A lógica deste algoritmo é que, caso existam mais do que um espaço vazio suficiente para alocar o bloco de memória que o processo necessita, o algoritmo aloca o mesmo, para o espaço que "melhor lhe serve", ou seja, o que deixa menos espaços vazios(-1's) possíveis. Para o fazer, precisámos de 3 vectores alocados, e progressivamente realocados dinamicamente. Os vetores são: **int *sub**, **int *x** e **NODO** y**. O primeiro while, percorre a lista através do ponteiro auxiliar **aux**, inicializado anteriormente a **auxMem**. Se durante esse while, é encontrado um nodo vazio(aux->pid=-1), o contador **c2** vai ser incrementado. Esse contador vai ser portanto o número de espaços vazios contínuos existentes na memória, e portanto o número de blocos vazios. Após realocarmos os vectores para o tamanho do número de blocos vazios. Os vectores vão estar em sintonia. Haverá o mesmo número de instâncias no vector de números de blocos vazios(cada um com o número

de espaços vazios -1") como haverá de instâncias no vector y, que é um vector de apontadores, onde cada apontador aponta para o começo do bloco vazio. Após atravessar a lista através do aux, verifica-se se algum dos int's em **x**(blocos vazios) ultrapassa o valor de **num_units**, para verificar se existe sequer algum bloco suficientemente grande para alocar memória. Se tal acontecer, a variável de verificação **int q**, é igualada a 1, para em baixo verificar-se se **q** é ou não diferente de 1(se não, quer dizer que não há nada a fazer, e a alocação foi um falhanço, retornando-se -1). Depois, com a ajuda de um **for**, mete-se no vector **sub** a diferença entre o tamanho do bloco vazio com o tamanho da memória do processo, para depois, mais abaixo, verificar-se qual é a instância com o menor valor, mas sempre igual ou superior a 0. Enquanto isto é feito(com a ajuda de outro **for**, se em algum dos casos, uma das instâncias em **sub** for igual a 0, acaba o for e a função termina logo, dado que não há melhor alocação do que a que não deixa fragmentos.

- int **alocate_units_mem_worst**(int process_id, int num_units, NODO* auxMem):
  A lógica é igual ao Best-Fit, só que em vez de se procurar no vetor **sub** pelo menor, procura-se pelo maior.

- void **gerarSolicitacoes**(int numSolic, NODO* auxMemSol):
  Nesta componente que gera as solicitações são dados como parâmetros o **número de solicitações**, para controlar as iterações no ciclo **while** e o apontador para a **lista** criada para o efeito da gestão de memória.
  Usamos a função **srand()** para gerar a **semente** que cria a aleatoriedade de todo o processo. Logo de seguida são declaradas as diversas variáveis para cada número aleatório, criado a partir da função **random_int**.
  Apresentamos o estado atual da memória e, de seguida, iniciamos o ciclo **while** que irá fazer o respetivo número de solicitações.

  variáveis aleatórias:
  ger: serve para selecionar se vai ser alocado ou dealocado as unidades de memória;
  num_units: número de unidadades de memória que vão ser tida em conta;

random_algoritmo: número aletáorio que correspode ao algoritmo de alocação de memória a ser usado;

Para seleção de algoritmo vamos proceder ao seguinte para alocar:
o algoritmo é executado a partir da chamada à função respetiva, o **random_int** é incrementado de forma a serem todos eles processos diferentes e ocorre a tarefa de report de memória.
Caso seja para dealocar, é chamada a função de dealocação e procede-se ao relatório também.
Da maneira que a **chamada de dealocação** está feita com 10 mil solicitações a **probabilidade** de alguma dar certo é **muito reduzida**, portanto este processo funciona melhor com um número mais baixo de solici

- int **alocate_units_mem_next**(int process_id, int num_units, NODO* auxMem):
  Esta função recebe como parâmetros o **id** do processo que queremos alocar em memória, o **num_units**, ou seja, o tamanho que o processo vai ocupar e a estrutura de dados da memória. Começamos por criar uma variável **count=0**,e **nodos=0**, uma variável **x=num_units**, e um nodo auxiliar para percorrer a lista **aux**. Vamos começar a correr a lista **aux** com um while, se um nodo estiver vazio, a partir desse, enquanto houver nodos vazios vamos aumentar o count e o nodos em 1. Depois, vamos verificar se a quantidade de nodos vazios são suficientes. Se o inicio não é o primeiro nodo, vamos percorrer a lista **auxMem** até chegar ao espaço vazio. Quando chegar vamos preencher "x"nodos, quando isto acontece sai do while principal. No final do while, o **count** volta a zero e **memMan = auxMem**

- PCB* **SJFS**(QUEUE* Prontos, int *index):
  Esta função vai receber como parâmetros a fila **Prontos** e um apontador para o inteiro **index**. Começamos por criar uma variavél de controlo do min **(PCB* minTemVida)**, e uma variavél **procFinal** que será o processo a executar. Criamos tambe uma variável inteira count inicializada a 0. Se a fila **Prontos** não estiver vazia, enquato o **count**for menor que o tamanho da fila **Prontos**(usando um while) vamos começar por retirar

o **procFinal** da fila **Prontos**, depois, se o estado do **procFinal** for igual a **READY** vamos comparar os **tempVida** do **procFinal** e do **minTemVida** para descobrir o mais pequeno. Depois colocamos o **procFinal** na fila **Prontos** e aumentamos o **count** em um. Depois saímos do while e passamos o **procFinal** para o estado **Running** com a chamada á função **addRunning**(minTempVida,index).

– PCB* **Prioridades**(QUEUE* Prontos, int *index):
Esta função vai receber como parâmetros a fila **Prontos** e um apontador para o inteiro **index**. Começamos por criar uma variavél de controlo do min **(PCB* minPriori)**, e uma variavél **procFinal** que será o processo a executar. Criamos tambe uma variável inteira count inicializada a 0. Se a fila **Prontos** não estiver vazia, enquato o **count**for menor que o tamanho da fila **Prontos**(usando um while) vamos começar por retirar o **procFinal** da fila **Prontos**, depois, se o estado do **procFinal** for igual a **READY** vamos comparar as prioridades **(priori)** do **procFinal** e do **minPriori** para descobrir o mais pequeno. Depois colocamos o **procFinal** na fila **Prontos** e aumentamos o **count** em um. Depois saímos do while e passamos o **procFinal** para o estado **Running** com a chamada á função **addRunning**(minPriori, index).

3.2.3 – Main.c

No programa principal começamos por apresentar um menu que permite ao utilizador selecionar o algoritmo de escalonamento ou um dos reports disponiveis (através de um scanf) segundo o qual toda a gestão se vai desenrolar. Ao ser selecionado o escalonador depois vai aparecer outro menu para ser selecionado o algoritmo de alocação de memória que vai ser usado para cada processo.
No início do programa, são declaradas algumas variáveis, tais como o tempo global do gestor (**tempo**), entre outras variáveis auxiliares para efeitos de output e de debugging. Temos também um inteiro sob a forma de apontador que guarda o índice do array memory correspondente à última posição (**indexMemory**) e o index da última instrução do array que foi executada. A variável (**procFinal**) representa cada processo após ter sofrido a ação dos comandos.

Declarámos também uma variãveis inteira **pidProc** que serve para atribuição dos respetivos **identificadores de processos**.

Filas fulcrais do programa

O programa é sustentado por 3 filas:

* Programas: fila inicial que alberga todos os programas lidos do ficheiro plan.txt com toda a informação que é inserida a partir da função readProgram().
* Blocked
* Terminated

Toda a gestão está assente nas trocas de processos entre estas filas.
De seguida lemos os comandos de control.txt substituindo o código da função lerControl(), uma vez que não conseguimos implentar a partir da chamada à função.

Tarefa principal do programa

No inicio começamos por criar uma variável **escal**, que vai ser utilizada para escolher a opção do menú que queremos correr.

Quando o **escal = 1** vamos começar por retirar o **proc** da fila **Programas**, e vamos colocar na fila **Prontos**, esta troca é feita sequencialmente. Depois mandamos o procFinal para a função **FCFS** para ser tratado.
Quando o **escal = 2** passamos um número random de processos entre 0 e o size da fila **Blocked** para a fila **Prontos**
Quando o **escal = 3** vamos mover os processos da fila **Programas** para a fila **Prontos** de uma só vez,depois retiramos um processo sequencialmente por ordem de chegada da fila **ProgramasAux** e vamos mandar o procFinal para a função **SJFS** para ser tratada.
Quando o **escal = 4** vamos mover os processos da fila Programas para a fila Prontos de uma só vez, depois vamos tratar das prioridades, usando a função **countInstr** vamos ver quantos processos existem no ficheiro "plan.txt", depois vamos criar um vetor para guardar a prioridade de cada processo, atribuir as prioridades aos respectivos processos que estão no array **priori**[] que foi definido no inicio do programa e que contêm

as prioridades dos processos no ficheiro "plan.txt".Depois retiramos um processo sequencialmente por ordem de chegada da lista **ProgramasAux** e vamos mandar o **procFinal** para a função **Prioridades** para ser tratada.

A lista **ProgramasAux** serve como uma auxiliar que contém exatamente o mesmo que a fila **Programas**. Esta nova lista auxiliar irá ser usada para obtermos o **proc** que será necessário para entrar na execução das instruções para funções de controlo, entre outras. Quando o **escal = 5** o programa vai fazer uma chamada á função Report.
Quando **escal = 7** o programa vai ser terminado.

Vai depois haver um ciclo while que só vai acabar quando fizer todos os comandos que estão no ficheiro "control.txt", este vai ser o ciclo principal. Dentro do ciclo e após a verificação inicial, o programa vai percorrer o array de controlos (**controlArr**) e conforme isso verifica a igualdade com os vários char (**instr**) representativos para executar o respetivo excerto de código que corresponde ao controlo.
–**Caso instr seja igual a E:**
Ao *startingTime* é atribuído o tempo atual do gestor para que seja possível controlar o tempo de execução de cada processo, através da comparação com a soma do *startingTime* com o *tempoVida*. É nesta situação que vão ser escolhidos os escalonadores.
–**Caso instr seja igual a R:**
Faz uma chamada á função **report**.

Vai haver outra ciclo while, que vai tratar das instruções em memória de cada processo.Dentro do ciclo e após a verificação inicial, o programa vai percorrer o array de controlos (**controlArr**) e conforme isso verifica a igualdade com os vários char (**instr1**).

–**Caso instr1 seja igual a I:**
Nesta situação vai ser escolhido o instante em que o processo vai ser interrompido, primeiro o seu estado vai ser alterado **procFinal->estadoProc = WAITING** e depois é colocado na fila **Blocked**.
–**Caso instr1 seja igual a T:**
Ao *startingTime* é atribuído o tempo atual do gestor para que

seja possível controlar o tempo de execução de cada processo. O seu estado vai ser alterado **procFinal->estadoProc = TERMINATED** e depois é colocado na fila **Terminated**.

Gestão de Memória

Após a execução dos algoritmos de Escalonamento, é inicializada uma nova lista ligada chamada *auxMemSol*, da mesma forma que inicializámos a *auxMem* anteriormente,e que vai representar a nossa memória disponível(a qual iremos usar no gerador de Solicitações).
Inicializamos a variável *int N_units* a 10000, que será a **quantidade de solicitações** feitas na experiência da gestão de memória.
Por fim, chamamos a função **gerarSolicitacoes**. Apresentamos o estado final da memória para verificar que tudo correu como esperado e terminamos o programa.

3.3 Conclusões

A gestão de processos, em particular, o escalonamento assenta no conceito do **Modelo de 5 estados** e nas transferências entre as diferentes filas de processos. O gestor de memória deve ser capaz de lidar com a simulação da passagem do tempo e as funções complementam-se para conseguir um resultado final conjunto.

Capítulo

4

Testes à implementação

4.1 Introdução

Neste capítulo dedicado aos testes da implementação atual do programa faremos várias tentativas com **diferentes *inputs* e os respetivos *outputs*** de forma a clarificar o funcionamento do gestor com base nos resultados obtidos.

4.2 Apresentação e explicação

Nas seguintes imagens segue-se a demonstração dos diversos testes feitos ao programa.
Primeiramente, neste capítulo será apresentada a **imagem com o input seguida do respetivo output**.

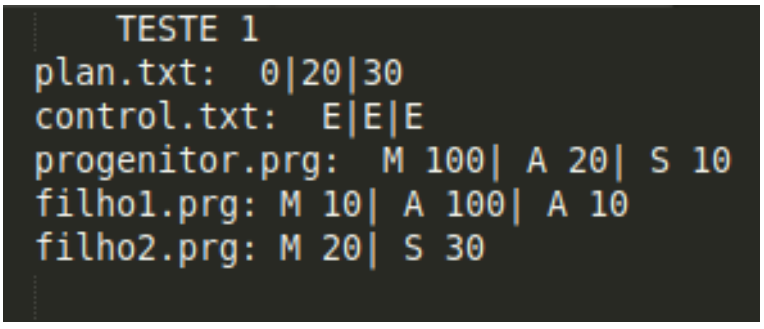

Figura 4.1: Input: Teste 1 Neste primeiro teste verificamos que o comando **E** e as instruções aritméticas **M, A, S** estão funcionais.

```
---Gestor de processos---
Escalonadores:
1-FCFS
-------------------------
Selecione uma opção do menu:
1
ControlMain[0]: E
ControlMain[1]: E
ControlMain[2]: E

Starting time:0
Memory index: 0
Running
0
IndexExec: 0
---------------
Entrei na execução das instrs
num: 100
Proc->var: 100
num: 20
Proc->var: 120
num: 10
Proc->var: 110
Processo Executado!
---------------
var: 110
---------------
Tempo: 3
Starting time:3
Memory index: 3
Running
3
IndexExec: 3
---------------
Entrei na execução das instrs
num: 10
Proc->var: 10
num: 100
Proc->var: 110
num: 10
Proc->var: 120
Processo Executado!
---------------
var: 120
---------------
Tempo: 6
Starting time:6
Memory index: 6
Running
6
IndexExec: 6
---------------
```

Figura 4.2: Output: Teste 1.1

```
---------------
Entrei na execução das instrs
num: 20
Proc->var: 20
num: 30
Proc->var: -10
Processo Executado!
---------------
var: -10
---------------
Tempo: 8

Memory[0]: M 100
Memory[1]: A 20
Memory[2]: S 10
Memory[3]: M 10
Memory[4]: A 100
Memory[5]: A 10
Memory[6]: M 20
Memory[7]: S 30

Tempo atual: 8
```

[H]

Figura 4.3: Output: Teste 1.2

```
   TESTE 2
plan.txt:  0|20|30
control.txt:  E|E|R
progenitor.prg:  M 100| A 20| T
filho1.prg: M 10| B| A 10
filho2.prg: M 20| S 30
```

Figura 4.4: Input: Teste 2 Neste Input mostramos que a instrução **T** e a instrução **R** estão a funcionar.

```
---Gestor de processos---
Escalonadores:
1-FCFS
------------------------
Selecione uma opção do menu:
1
ControlMain[0]: E
ControlMain[1]: E
ControlMain[2]: R

Starting time:0
Memory index: 0
Running
0
IndexExec: 0
---------------
Entrei na execução das instrs
num: 100
Proc->var: 100
num: 20
Proc->var: 120
num: 0
Proc->var: 120
Processo Executado!
---------------
var: 120
---------------
Tempo: 3
Starting time:3
Memory index: 3
Running
3
IndexExec: 3
---------------
Entrei na execução das instrs
num: 10
Proc->var: 10
num: 0
Proc->var: 10
num: 0
Proc->var: 10
Processo Executado!
---------------
var: 10
---------------
Tempo: 6
```

Figura 4.5: Output: Teste 2.1

```
Report do simulador:
TEMPO ATUAL: 6

PROCESSO EM EXECUÇÂO:
---------------
PROCESSOS BLOQUEADOS:
---------------
PROCESSOS PRONTOS A EXECUTAR:

PID: 0
PPID: 0
Prioridade: 0
Valor: 0
Tempo de iniciação: 30
Tempo do CPU Usado: 1
---------------
PROCESSOS TERMINADOS:

PID: 0
PPID: 0
Prioridade: 0
Valor: 120
Tempo de iniciação: 0
Tempo do CPU Usado: 3

PID: 1
PPID: 0
Prioridade: 0
Valor: 10
Tempo de iniciação: 20
Tempo do CPU Usado: 3

Memory[0]: M 100
Memory[1]: A 20
Memory[2]:  0
Memory[3]: M 10
Memory[4]:  0
Memory[5]:  0

Tempo atual: 6
```

Figura 4.6: Output: Teste 2.2

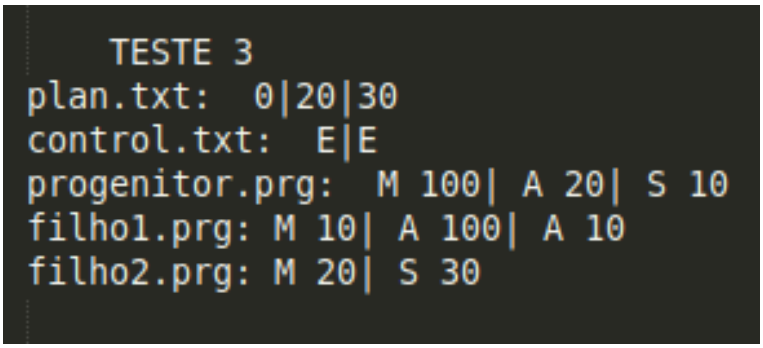

Figura 4.7: Input: Teste 3 Quando o número de processos em **plan.txt** não for igual ao número de controlos em **control.txt** serão executados os processos que tiverem um comando associado.

```
---Gestor de processos---
Escalonadores:
1-FCFS
------------------------
Selecione uma opção do menu:
1
ControlMain[0]: E
ControlMain[1]: E

Starting time:0
Memory index: 0
Running
0
IndexExec: 0
---------------
Entrei na execução das instrs
num: 100
Proc->var: 100
num: 20
Proc->var: 120
num: 10
Proc->var: 110
Processo Executado!
---------------
var: 110
---------------
Tempo: 3
Starting time:3
Memory index: 3
Running
3
IndexExec: 3
---------------
Entrei na execução das instrs
num: 10
Proc->var: 10
num: 100
Proc->var: 110
num: 10
Proc->var: 120
Processo Executado!
---------------
var: 120
---------------
Tempo: 6

Memory[0]: M 100
Memory[1]: A 20
Memory[2]: S 10
Memory[3]: M 10
Memory[4]: A 100
Memory[5]: A 10

Tempo atual: 6
```

Figura 4.8: Output: Teste 3.1

```
---Gestor de processos---
Escalonadores:
1-FCFS
------------------------
Selecione uma opção do menu:
1
ControlMain[0]:



Tempo atual: 0
```

Figura 4.10: Output: Teste 4.1

```
    TESTE 4
plan.txt:  0|20|30
control.txt:  vazio
progenitor.prg:  M 100| A 20| S 10
filho1.prg: M 10| A 100| A 10
filho2.prg: M 20| S 30
```

Figura 4.9: Input: Teste 4 Quando não forem fornecidos quaisquer comandos ao programas a partir do ficheiro **control.txt**, o programa não executa ações nenhum processo.

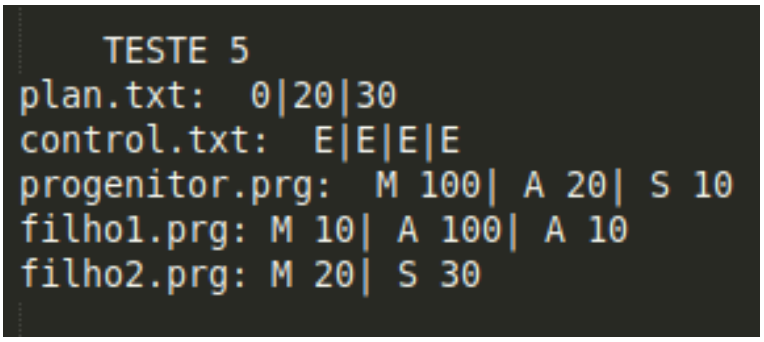

Figura 4.11: Input: Teste 5 Quando houver excesso de comandos o programa executa na perfeição, pelo que se conclui que o que determina a falha da aplicação é o défice de controlos face ao número de processos.

```
---Gestor de processos---
Escalonadores:
1-FCFS
-------------------------
Selecione uma opção do menu:
1
ControlMain[0]: E
ControlMain[1]: E
ControlMain[2]: E
ControlMain[3]: E

Starting time:0
Memory index: 0
Running
0
IndexExec: 0
---------------
Entrei na execução das instrs
num: 100
Proc->var: 100
num: 20
Proc->var: 120
num: 10
Proc->var: 110
Processo Executado!
---------------
var: 110
---------------
Tempo: 3
Starting time:3
Memory index: 3
Running
3
IndexExec: 3
---------------
Entrei na execução das instrs
num: 10
Proc->var: 10
num: 100
Proc->var: 110
num: 10
Proc->var: 120
Processo Executado!
---------------
var: 120
---------------
Tempo: 6
Starting time:6
Memory index: 6
Running
6
IndexExec: 6
```

Figura 4.12: Output: Teste 5.1

```
Entrei na execução das instrs
num: 20
Proc->var: 20
num: 30
Proc->var: -10
Processo Executado!
---------------
var: -10
---------------
Tempo: 8

Memory[0]: M 100
Memory[1]: A 20
Memory[2]: S 10
Memory[3]: M 10
Memory[4]: A 100
Memory[5]: A 10
Memory[6]: M 20
Memory[7]: S 30

Tempo atual: 8
```

Figura 4.13: Output: Teste 5.2

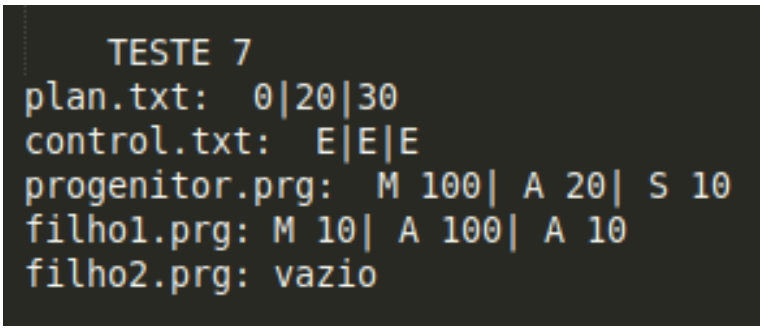

Figura 4.14: Input: Teste 6 Quando não houver instruções num determinado processo, esse processo não corre.

```
---Gestor de processos---
Escalonadores:
1-FCFS
------------------------
Selecione uma opção do menu:
1
ControlMain[0]: E
ControlMain[1]: E
ControlMain[2]: E

Starting time:0
Memory index: 0
Running
0
IndexExec: 0
---------------
Entrei na execução das instrs
num: 100
Proc->var: 100
num: 20
Proc->var: 120
num: 10
Proc->var: 110
Processo Executado!
---------------
var: 110
---------------
Tempo: 3
Starting time:3
Memory index: 3
Running
3
IndexExec: 3
---------------
Entrei na execução das instrs
num: 10
Proc->var: 10
num: 100
Proc->var: 110
num: 10
Proc->var: 120
Processo Executado!
---------------
var: 120
---------------
Tempo: 6
Starting time:6
Memory index: 6
Running
6
IndexExec: 6
```

Figura 4.15: Output: Teste 6.1

```
Entrei na execução das instrs
num: 0
Proc->var: 0
Processo Executado!
--------------
var: 0
--------------
Tempo: 7

Memory[0]: M 100
Memory[1]: A 20
Memory[2]: S 10
Memory[3]: M 10
Memory[4]: A 100
Memory[5]: A 10
Memory[6]:  0

Tempo atual: 7
```

Figura 4.16: Output: Teste 6.2

4.3 Conclusões

Embora feitos testes exaustivamente á gestão de memória, estes não foram documentados.

Capítulo

5

Conclusões e Trabalho Futuro

5.1 Conclusões Principais

5.2 Trabalho Futuro

Além das pequenas explicações deixamos algum código em comentário para servir de debugging.

O que é que ficou por fazer, e porquê?

Neste projeto, ficou por terminar o exec() e o fork() (**Instruções L e C** devido à complexidade das funcionalidades e da sua implementação face à nossa interpretação do tema. Decidimos que iremos terminá-las na 2ª parte do projeto. Ficou por implementar também o controlo **I** e o controlo **T**, dado que, da maneira que estávamos a percorrer o código, se existisse uma ordem para **I**(Interromper) ou **T**(Terminar) o processo a decorrer, o mesmo já tinha sido executado, portanto não haveria nada a interromper ou terminar.

O que é que seria interessante fazer, mas não foi feito por não ser exatamente o objetivo deste trabalho?

No nosso ponto de vista seria interessante uma implementação do gestor com base no **modelo de 7 estados**. Esta abordagem traria um maior detalhe ao trabalho e seria concerteza ainda mais desafiante.

Bibliografia

[1] ArnonEilat. queue.c, 2019. [Online] https://gist.github.com/ArnonEilat/4471278. Último acesso a 13 de Maio de 2020.

[2] Paul Crocker. Capítulo 4: Processos, 2020. [Online] http://www.di.ubi.pt/~operativos/teoricos/capitulo4.pdf. Último acesso a 13 de Maio de 2020.

[3] Paul Crocker. Capítulo 5: Escalonamento da CPU, 2020. [Online] http://www.di.ubi.pt/~operativos/teoricos/capitulo5.pdf. Último acesso a 13 de Maio de 2020.