Portuguese

Sistemas Operacionais

1.0

Gerado por Doxygen 1.8.13

Contents

Chapter 1

Escalonador de Execução Postergada

1.1 Introdução

Projeto de Sistemas Operacionais 1/2019

Professora: Dra. Alba Cristina Magalhães Alves de Melo

Aluno: Bernardo Costa Nascimento 140080279

Aluno: Cristano

1.2 Instruções Básicas

O presente trabalho é o desenvolvimento de um escalonador de execução postergada. Para compilar o projeto, basta executar o seguinte comando:

```
sh compile.sh
```

A compilação irá gerar 4 executáveis com os seguintes nomes:

- escalonador
- execucao_postergada
- shutdown
- hello

Para utilizar o sistema, basta executar o 'escalonador'. O programa deve ser iniciado com um dos três argumentos -h, -t ou -f, como exemplificador abaixo:

```
./escalonador -h&
```

O comando acima irá iniciar a execução do escalonador em modo background (&). Feito isso o usuário deve utilizar o programa execucao_postergada para passar algum programa para escalonamento.

Para executar o programa execucao_postergada, o usuário deve passar 2 argumentos: nome do programa e delay. Segue um exemplo:

```
./execucao_postergada hello 4
```

O comando acima irá selecionar o programa hello para ser executado em 4 segundos. O usuário poderá executar quantos programas desejar dessa maneira.

Quando o usuário quiser concluir a execução do sistema, basta executar o programa shutdown.

```
./shutdown
```

Uma documentação extensa a respeito de todo código pode ser encontrada neste site. Basta seguir para Ficheiros –> Lista de Ficheiros. Feito isso, basta escolher o arquivo para ler a documentação.

Chapter 2

Escalonador de Execução Postergada

Introdução

- Projeto de Sistemas Operacionais 1/2019
- Professora: Dra. Alba Cristina Magalhães Alves de Melo
- Aluno: Bernardo Costa Nascimento; 140080279
- Aluno: Cristano

Instruções Básicas

O presente trabalho é o desenvolvimento de um escalonador de execução postergada. Para compilar o projeto, basta executar o seguinte comando:

```
sh compile.sh
```

A compilação irá gerar 4 executáveis com os seguintes nomes:

- escalonador
- execucao_postergada
- shutdown
- hello

Para utilizar o sistema, basta executar o 'escalonador'. O programa deve ser iniciado com um dos três argumentos -h, -t ou -f, como exemplificador abaixo:

```
./escalonador -h&
```

O comando acima irá iniciar a execução do escalonador em modo background (&). Feito isso o usuário deve utilizar o programa execucao_postergada para passar algum programa para escalonamento.

Para executar o programa execucao_postergada, o usuário deve passar 2 argumentos: nome do programa e delay. Segue um exemplo:

```
./execucao_postergada hello 4
```

O comando acima irá selecionar o programa hello para ser executado em 4 segundos. O usuário poderá executar quantos programas desejar dessa maneira.

Quando o usuário quiser concluir a execução do sistema, basta executar o programa shutdown.

```
./shutdown
```

Após clonar este repositório, o usuário pode acessar a documentação do sistema por meio do site doxygen. Para gerar o site, baixe e execute o doxygen com o arquivo "escalonadorEP". O programa irá gerar a pasta "docs" no diretório raíz do sistema. A documentação pode ser acessada abrindo o arquivo docs/html/index.html.

Chapter 3

Índice das estruturas de dados

3.1 Estruturas de dados

Lista das estruturas de dados com uma breve descrição:

Chapter 4

Índice dos ficheiros

4.1 Lista de ficheiros

Lista de todos os ficheiros com uma breve descrição:

Chapter 5

Documentação da classe

5.1 Referência à estrutura execd

Define a estrutura de jobs que já foram executados.

`#include <dataStructures.h>`

Campos de Dados

- pid_t **pid**
  PID do processo que realizou a n-ésima execução do programa.
- char * **program**
  Programa executado.
- time_t **sent**
  Hora do envio do job.
- time_t **begin**
  Hora que o programa iniciou a execução.
- time_t **end**
  Hora que o programa concluíu a execução.
- double **makespan**
  Tempo total de execução do programa.
- struct **execd** * **next**
  Próximo processo da lista de execuções concluídas.

5.1.1 Descrição detalhada

Define a estrutura de jobs que já foram executados.

Definido na linha 32 do ficheiro dataStructures.h.

5.1.2 Documentação dos campos e atributos

5.1.2.1 begin

```
time_t begin
```

Hora que o programa iniciou a execução.

Definido na linha 40 do ficheiro dataStructures.h.

5.1.2.2 end

```
time_t end
```

Hora que o programa concluíu a execução.

Definido na linha 42 do ficheiro dataStructures.h.

5.1.2.3 makespan

```
double makespan
```

Tempo total de execução do programa.

Definido na linha 44 do ficheiro dataStructures.h.

5.1.2.4 next

```
struct execd* next
```

Próximo processo da lista de execuções concluídas.

Definido na linha 47 do ficheiro dataStructures.h.

5.1.2.5 pid

```
pid_t pid
```

PID do processo que realizou a n-ésima execução do programa.

Definido na linha 34 do ficheiro dataStructures.h.

5.1.2.6 program

```
char* program
```

Programa executado.

Definido na linha 36 do ficheiro dataStructures.h.

5.1.2.7 sent

```
time_t sent
```

Hora do envio do job.

Definido na linha 38 do ficheiro dataStructures.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **dataStructures.h**

5.2 Referência à estrutura execq

Define a estrutura da lista de jobs.

```
#include <dataStructures.h>
```

Campos de Dados

- int **job**
  Número do Job a ser executado.
- char * **name**
  Nome do programa a ser executado.
- int **rDelay**
  Delay do programa (definido pelo usuário)
- time_t **uDelay**
  Delay atualizado conforme os jobs são executados.
- time_t **sent**
  Hora de envio do job.
- struct **execq** * **prox**
  Próximo Job da lista.

5.2.1 Descrição detalhada

Define a estrutura da lista de jobs.

Definido na linha 14 do ficheiro dataStructures.h.

5.2.2 Documentação dos campos e atributos

5.2.2.1 job

```
int job
```

Número do Job a ser executado.

Definido na linha 16 do ficheiro dataStructures.h.

5.2.2.2 name

```
char* name
```

Nome do programa a ser executado.

Definido na linha 18 do ficheiro dataStructures.h.

5.2.2.3 prox

```
struct execq* prox
```

Próximo Job da lista.

Definido na linha 28 do ficheiro dataStructures.h.

5.2.2.4 rDelay

```
int rDelay
```

Delay do programa (definido pelo usuário)

Definido na linha 20 do ficheiro dataStructures.h.

5.2.2.5 sent

```
time_t sent
```

Hora de envio do job.

Definido na linha 25 do ficheiro dataStructures.h.

5.2.2.6 uDelay

```
time_t uDelay
```

Delay atualizado conforme os jobs são executados.

Definido na linha 23 do ficheiro dataStructures.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **dataStructures.h**

5.3 Referência à estrutura fTree

Define a estrutura da Fat Tree (foi implementada uma Binary Tree)

```
#include <dataStructures.h>
```

Campos de Dados

- pid_t **id**
  ID do Gerente.
- int **lenght**
  Tamanho do vetor de conexões.
- pid_t * **connects**
  Vetor de conexões.
- struct **fTree** * **left**
  Próximo nó da árvore posicionado a esquerda.
- struct **fTree** * **right**
  Próximo nó da árvore posicionado a direita.

5.3.1 Descrição detalhada

Define a estrutura da Fat Tree (foi implementada uma Binary Tree)

Definido na linha 51 do ficheiro dataStructures.h.

5.3.2 Documentação dos campos e atributos

5.3.2.1 connects

```
pid_t* connects
```

Vetor de conexões.

Definido na linha 58 do ficheiro dataStructures.h.

5.3.2.2 id

```
pid_t id
```

ID do Gerente.

Definido na linha 53 do ficheiro dataStructures.h.

5.3.2.3 left

```
struct fTree* left
```

Próximo nó da árvore posicionado a esquerda.

Definido na linha 61 do ficheiro dataStructures.h.

5.3.2.4 lenght

```
int lenght
```

Tamanho do vetor de conexões.

Definido na linha 56 do ficheiro dataStructures.h.

5.3.2.5 right

```
struct fTree* right
```

Próximo nó da árvore posicionado a direita.

Definido na linha 63 do ficheiro dataStructures.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **dataStructures.h**

5.4 Referência à estrutura hyperTorus

Define a estrutura do Hybercube e Torus.

```
#include <dataStructures.h>
```

Campos de Dados

- pid_t **id**

  ID do Gerente.
- int **length**

  Tamanho do vetor de conexões.
- pid_t * **connects**

  Vetor de conexões.
- struct **hyperTorus** * **next**

  Próximo elemento do grafo Hybercube ou Torus.

5.4.1 Descrição detalhada

Define a estrutura do Hybercube e Torus.

Definido na linha 67 do ficheiro dataStructures.h.

5.4.2 Documentação dos campos e atributos

5.4.2.1 connects

```
pid_t* connects
```

Vetor de conexões.

Definido na linha 74 do ficheiro dataStructures.h.

5.4.2.2 id

```
pid_t id
```

ID do Gerente.

Definido na linha 69 do ficheiro dataStructures.h.

5.4.2.3 length

```
int length
```

Tamanho do vetor de conexões.

Definido na linha 72 do ficheiro dataStructures.h.

5.4.2.4 next

```
struct hyperTorus* next
```

Próximo elemento do grafo Hybercube ou Torus.

Definido na linha 77 do ficheiro dataStructures.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **dataStructures.h**

5.5 Referência à estrutura manq

Define a estrutura de processos livres.

```
#include <dataStructures.h>
```

Campos de Dados

- pid_t **_id**
  ID do Gerente.
- struct **manq** * **next**
  Próximo Gerente da fila.

5.5.1 Descrição detalhada

Define a estrutura de processos livres.

Definido na linha 81 do ficheiro dataStructures.h.

5.5.2 Documentação dos campos e atributos

5.5.2.1 _id

```
pid_t _id
```

ID do Gerente.

Definido na linha 83 do ficheiro dataStructures.h.

5.5.2.2 next

```
struct manq* next
```

Próximo Gerente da fila.

Definido na linha 86 do ficheiro dataStructures.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **dataStructures.h**

5.6 Referência à estrutura msg_packet

Pacote de mensagem utilizado para comunicação entre escalonador, execucao_postergada e gerentes.

```
#include <msgQueue.h>
```

Campos de Dados

- long **type**
  Define o tipo da mensagem.
- char **name** [256]
  Nome do programa a ser escalonado.
- int **delay**
  Delay (in seconds) for execution.
- int **_mdst**
  ID do gerente destino.
- int **_id**
  ID do gerente para utilização do escalonador.
- int **finish**
  Flag de finalização.
- pid_t **pid**
  PID do filho do gerente.
- time_t **begin**
  Hora de início da execução do filho do gerente.
- time_t **end**
  Hora de término da execução do filho do gerente.

5.6.1 Descrição detalhada

Pacote de mensagem utilizado para comunicação entre escalonador, execucao_postergada e gerentes.

Definido na linha 16 do ficheiro msgQueue.h.

5.6.2 Documentação dos campos e atributos

5.6.2.1 _id

```
int _id
```

ID do gerente para utilização do escalonador.

Definido na linha 28 do ficheiro msgQueue.h.

5.6.2.2 _mdst

```
int _mdst
```

ID do gerente destino.

Definido na linha 26 do ficheiro msgQueue.h.

5.6.2.3 begin

```
time_t begin
```

Hora de início da execução do filho do gerente.

Definido na linha 35 do ficheiro msgQueue.h.

5.6.2.4 delay

```
int delay
```

Delay (in seconds) for execution.

Definido na linha 23 do ficheiro msgQueue.h.

5.6.2.5 end

```
time_t end
```

Hora de término da execução do filho do gerente.

Definido na linha 37 do ficheiro msgQueue.h.

5.6.2.6 finish

```
int finish
```

Flag de finalização.

Definido na linha 30 do ficheiro msgQueue.h.

5.6.2.7 name

```
char name[256]
```

Nome do programa a ser escalonado.

Definido na linha 21 do ficheiro msgQueue.h.

5.6.2.8 pid

```
pid_t pid
```

PID do filho do gerente.

Definido na linha 33 do ficheiro msgQueue.h.

5.6.2.9 type

```
long type
```

Define o tipo da mensagem.

Definido na linha 18 do ficheiro msgQueue.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **msgQueue.h**

5.7 Referência à estrutura pid_packet

Pacote de mensagem para comunicação entre escalonador, execucao_postergada e shutdown.

```
#include <msgQueue.h>
```

Campos de Dados

- long **type**
  Tipo da mensagem.
- pid_t **pid**
  PID do escalonador.

5.7.1 Descrição detalhada

Pacote de mensagem para comunicação entre escalonador, execucao_postergada e shutdown.

Definido na linha 41 do ficheiro msgQueue.h.

5.7.2 Documentação dos campos e atributos

5.7.2.1 pid

```
pid_t pid
```

PID do escalonador.

Definido na linha 45 do ficheiro msgQueue.h.

5.7.2.2 type

```
long type
```

Tipo da mensagem.

Definido na linha 43 do ficheiro msgQueue.h.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- **msgQueue.h**

Chapter 6

Documentação do ficheiro

6.1 Referência ao ficheiro dataStructures.c

```
#include "dataStructures.h"
```

Funções

- void **createQueue** (**execq** ∗∗queue)

  Inicia a estrutura de dados do tipo execq.
- void **insertProcess** (**execq** ∗∗queue, char ∗_name, int _delay)

  Insere um novo job na fila.
- void **removeProcess** (**execq** ∗∗queue)

  Remove um job da lista.
- void **listProcesses** (**execq** ∗queue)

  Lista todos os jobs não concluídos.
- void **updateDelays** (**execq** ∗∗queue)

  Realiza update do delay dos jobs na lista.
- void **createExecD** (**execd** ∗∗done)

  Inicializador da estrutura execd.
- void **insertExecD** (**execd** ∗∗done, pid_t pid, char ∗program, time_t sent, time_t begin, time_t end)

  Insere informação sobre todos os processos que executaram um dado job.
- void **deleteExecD** (**execd** ∗∗done)

  Deleta toda a fila de processos executados.
- void **listExecD** (**execd** ∗done)

  Lista todos os processos na fila de processos executados.
- void **createFTree** (**fTree** ∗∗_tree)

  *Inicializador da estrutura do tipo **fTree** (*p. **??***).*
- void **definesTree** (**fTree** ∗∗_tree, int _parent, int ∗_node, int _level)

  Cria a estrtura da Fat Tree (Binary Tree)
- void **readTree** (**fTree** ∗_tree)

  Função de leitura da árovre simbólica.
- void **deleteTree** (**fTree** ∗∗_tree)

  Função padrão de deleção para estrutura Fat Tree (Binary Tree)
- pid_t ∗ **get_fTreeConnection** (**fTree** ∗_tree, int _id)

Define as conexões de um dado processo gerente.

- void **createHyperTorus** (**hyperTorus** ∗∗_ht)

  *Inicializador da estrutura do tipo **hyperTorus** (*p. **??**).*
- void **definesHyper** (**hyperTorus** ∗∗_hyper, int _id)

  Define a estrutura simbólica Hypercube.
- void **definesTorus** (**hyperTorus** ∗∗_torus, int _id)

  Define a estrutura simbólica Torus.
- void **readHyperTorus** (**hyperTorus** ∗_ht)

  Função de leitura da estrutura simbólica Hypercube ou Torus.
- void **deleteHyperTorus** (**hyperTorus** ∗∗_ht)

  Função padrão de deleção das estruturas Hypercube e Torus.
- pid_t ∗ **get_htConnection** (**hyperTorus** ∗_ht, int _id)

  Define as conexões de um dado processo processo gerente.
- void **readHTConnections** (pid_t ∗connections, int _id)

  Realiza a leitura do vetor de conexões na estrutura Hypercube ou Torus.
- void **readFTConnections** (pid_t ∗connections, int _id)

  Realiza a leitura do vetor de conexões na estrutura Fat Tree.
- void **createManQ** (**manq** ∗∗_manq)

  Inicializador padrão para estrutura do tipo manq.
- void **insertManQ** (**manq** ∗∗_manq, int _id)

  Insere processos, na ordem, de execução.
- **manq** ∗ **removeManQ** (**manq** ∗∗_manq)

  Remove um processo da fila de escalonamento.
- void **readManQ** (**manq** ∗_manq)

  Realiza a leitura da fila de escalonamento.

Variáveis

- int **_jobs** = 0

  Contador de jobs executados.

6.1.1 Documentação das funções

6.1.1.1 createExecD()

```
void createExecD (
              execd ** done )
```

Inicializador da estrutura execd.

Inicializador padrão da estrutura do tipo execd

Parâmetros

execd	∗∗queue; Ponteiro para estrutura do tipo execd;

Retorna

void;

Definido na linha 85 do ficheiro dataStructures.c.

6.1.1.2 createFTree()

```
void createFTree (
              fTree ** _tree )
```

Inicializador da estrutura do tipo **fTree** (p. **??**).

Inicializador padrão da estrutura do tipo **fTree** (p. **??**).

Parâmetros

fTree *(p. **??**)*	**_tree; Ponteiro para estrutura do tipo **fTree** (p. **??**);

Retorna

void;

Definido na linha 149 do ficheiro dataStructures.c.

6.1.1.3 createHyperTorus()

```
void createHyperTorus (
              hyperTorus ** _ht )
```

Inicializador da estrutura do tipo **hyperTorus** (p. **??**).

Inicializador padrão da estrutura do tipo **hyperTorus** (p. **??**)

Parâmetros

hyperTorus *(p. **??**)*	**_ht; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);

Retorna

void;

Definido na linha 235 do ficheiro dataStructures.c.

6.1.1.4 createManQ()

```
void createManQ (
              manq ** _manq )
```

Inicializador padrão para estrutura do tipo manq.

Parâmetros

manq	**_manq; Ponteiro para a estrutura do tipo manq;

Retorna

void;

Definido na linha 381 do ficheiro dataStructures.c.

6.1.1.5 createQueue()

```
void createQueue (
              execq ** queue )
```

Inicia a estrutura de dados do tipo execq.

Inicializador padrão de uma estrutura de dados do tipo execq.

Parâmetros

execq	**queue; Ponteiro para uma estrutura do tipo execq

Retorna

void;

Definido na linha 6 do ficheiro dataStructures.c.

6.1.1.6 definesHyper()

```
void definesHyper (
              hyperTorus ** _hyper,
             int _id )
```

Define a estrutura simbólica Hypercube.

A função tem como responsabilidade definir a estrutura simbólica do tipo Hypercube. A função é chamada N vezes (no caso desse sistema, N=16), e Passa o número da chamada como argumento. O número é associado ao <_id> do gerente.

Na estrutura Hypercube, cada nó realiza 4 conexões (com exceção do nó 0 que realiza uma 5a conexão com o escalonador). Essa conexões são feitas com base no número do nó. Cada nó só se conecta com nós com <↩ _id> que tenha apenas 1 bit de diferença. Para isso, cada chamada da função roda um for para todos os 16 ID's possíveis na estrutura e realiza um XOR entre o <_id> do gerente e cada outro ID da estrutura. Caso o resultado seja 1, 2, 4 ou 8, significa que apenas 1 bit mudou e, portanto, há conexão.

As conexões são salvas em um vetor alocado em cada nó da estrutura simbólica.

Parâmetros

hyperTorus *(*p. **??***)*	**_hyper; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);
int	_id; ID do n-ésimo gerente;

Definido na linha 239 do ficheiro dataStructures.c.

6.1.1.7 definesTorus()

```
void definesTorus (
              hyperTorus ** _torus,
             int _id )
```

Define a estrutura simbólica Torus.

A função, de forma semelhante a **definesHyper()** (p. **??**), tem como objetivo define a estrutura simbólica Torus. A função será chamada N vezes (no caso desse sistema, N=16), e passa o número da chamada como argumento <_id>.

A regra da estrutura Torus é a seguinte:

- A estrutura é composta de 4 linhas com 4 gerentes em cada linha;
- A estrutura é composta de 4 colunas com 4 gerentes em cada coluna;
- Dessa forma, determina-se a coluna pela conta: col = _id%4. Se o nó estiver na coluna 0, ele se conecta com o primeiro nó a direita e com o terceiro nó a direita. Se o nó estiver na coluna 1 ou 2, se conecta com o primeiro nó a esquerda e o primeiro nó a direita. Se o nó estiver na coluna 3, se conecta com o primeiro nó a esquerda e o primeiro nó da coluna.
- Ainda, se o <_id> do nó subtraído de 4 for menor que 0 (nós na primeira linha) Eles se conectam com os nós na ultima linha ((_id+12)%16). Caso contrário, se conectam com o nó da linha de cima.
- Por fim, se o <_id> acrescido de 4 for maior que 15 (nós da última linha), eles se conectam com os nós da primeira linha ((_id+4)%16). Caso contrário, se conectam com o nó da linha de baixo.

As conexões são salvas em um vetor de cada nó da estrutura simbólica.

Parâmetros

hyperTorus *(*p. **??***)*	**_torus; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);
int	_id; ID do n-ésimo gerente;

Retorna

void;

Definido na linha 272 do ficheiro dataStructures.c.

6.1.1.8 definesTree()

```
void definesTree (
              fTree ** _tree,
             int _parent,
             int * _node,
             int _level )
```

Cria a estrtura da Fat Tree (Binary Tree)

Função recursiva responsável pela criação da estrutura completa da árvore binária simbólica dos processos gerentes. Aqui serão calculadas todas as conexões que cada nó da árvore realiza, e cada nó irá receber um ID que, posteriormente, será associado a um processo gerente.

Parâmetros

fTree *(*p. **??***)*	∗∗_tree; Ponteiro para estrutura **fTree** (p. **??**);
int	_parent; ID do nó pai do nó sendo inserido na árvore;
int	∗_node; Ponteiro para o número do nó sendo inserido na árvore;
int	_level; Level atual da árvore sendo construído.

Definido na linha 153 do ficheiro dataStructures.c.

6.1.1.9 deleteExecD()

```
void deleteExecD (
              execd ** done )
```

Deleta toda a fila de processos executados.

Realiza a deleção e posterior "free" dos dados da fila de processos executados.

Parâmetros

execd	∗∗done; Ponteiro para estrutura do tipo execd;

Retorna

void;

Definido na linha 112 do ficheiro dataStructures.c.

6.1.1.10 deleteHyperTorus()

```
void deleteHyperTorus (
              hyperTorus ** _ht )
```

Função padrão de deleção das estruturas Hypercube e Torus.

Parâmetros

hyperTorus *(*p. **??***)*	**_ht; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);

Retorna

void;

Definido na linha 335 do ficheiro dataStructures.c.

6.1.1.11 deleteTree()

```
void deleteTree (
              fTree ** _tree )
```

Função padrão de deleção para estrutura Fat Tree (Binary Tree)

Parâmetros

fTree *(*p. **??***)*	**_tree; Ponteiro para estrutura do tipo **fTree** (p. **??**);

Retorna

void;

Definido na linha 205 do ficheiro dataStructures.c.

6.1.1.12 get_fTreeConnection()

```
pid_t * get_fTreeConnection (
              fTree * _tree,
             int _id )
```

Define as conexões de um dado processo gerente.

A função recebe a árvore simbólica gerada e o <_id> do processo gerente. Com isso, o processo gerente percorre a árvore simbólica até achar o nó com mesmo número que seu <_id> e, ao achar este nó, recupera as conexões que ele faz em forma de um vetor de tipo pid_t.

Parâmetros

fTree *(p. **??**)*	*_tree; Cópia da árvore simbólica;
int	_id; ID do processo gerente;

Retorna

pid_t*; Vetor com todas as conexões que o n-ésimo gerente faz;

Definido na linha 214 do ficheiro dataStructures.c.

6.1.1.13 get_htConnection()

```
pid_t * get_htConnection (
              hyperTorus * _hyper,
             int _id )
```

Define as conexões de um dado processo processo gerente.

A função busca na estrutura simbólica, usando o <_id> como base, o nó simbólico e as conexões que o mesmo faz no Hypercube ou Torus. Após encontrar o nó simbólico de mesmo ID que <_id> a função retorna o vetor de conexões do processo gerente.

Parâmetros

hyperTorus *(p. **??**)*	*_hyper; Cópia da estrutura simbólica **hyperTorus** (p. **??**);
int	_id; ID do processo gerente;

Retorna

pid_t*; Vetor com as conexões do n-ésimo processo gerente;

Definido na linha 346 do ficheiro dataStructures.c.

6.1.1.14 insertExecD()

```
void insertExecD (
              execd ** done,
             pid_t pid,
             char * program,
             time_t sent,
             time_t begin,
             time_t end )
```

Insere informação sobre todos os processos que executaram um dado job.

Inserção das informações resultantes da execuçaõ de um job qualquer pelo sistema. A inserção segue o padrão de fila.

Parâmetros

execd	∗∗done; Ponteiro para estrutura do tipo execd;
pid_t	pid; PID do processo que realizou a n-ésima execução do programa;
char	∗program; Nome do programa executado;
time_t	sent; Hora de envio do job;
time_t	begin; Hora do início da execução do programa pelo processo;
time_t	end; Hora de conclusão da execução do programa pelo processo;

Definido na linha 89 do ficheiro dataStructures.c.

6.1.1.15 insertManQ()

```
void insertManQ (
              manq ** _manq,
             int _id )
```

Insere processos, na ordem, de execução.

O escalonador utiliza um escalonamento bastante simples: FIFO (First In First Out). Dessa forma, a estrutura "manq" guarda, na ordem, os processos prontos para execução de um dado job.

Parâmetros

manq	∗∗_manq; Ponteiro para a fila do escalonamento;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 385 do ficheiro dataStructures.c.

6.1.1.16 insertProcess()

```
void insertProcess (
              execq ** queue,
             char * _name,
             int _delay )
```

Insere um novo job na fila.

Realiza a inserção de um novo job recebido pelo escalonador. Inserção é ordenada pelo delay. Dessa forma, não segue uma estrutura típica de fila, e sim de uma lista ordenada.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;
char	*_name; Ponteiro para o nome do programa;
int	_delay; Delay, em segundos, para execução do job;

Retorna

void;

Definido na linha 10 do ficheiro dataStructures.c.

6.1.1.17 listExecD()

```
listExecD (
              execd * done )
```

Lista todos os processos na fila de processos executados.

Realiza a impressão da informação sobre todos os processos que executaram algum programa durante toda a execução do escalonador.

Parâmetros

execd	*done; Cópia da estrutura do tipo execd;

Retorna

void;

Definido na linha 125 do ficheiro dataStructures.c.

6.1.1.18 listProcesses()

```
void listProcesses (
              execq * queue )
```

Lista todos os jobs não concluídos.

Realiza a impressão de todos os jobs que ainda estão para serem executados.

Parâmetros

execq	*queue; Cópia da estrutura do tipo execq

Retorna

void;

Definido na linha 49 do ficheiro dataStructures.c.

6.1.1.19 readFTConnections()

```
void readFTConnections (
            pid_t * connections,
            int _id )
```

Realiza a leitura do vetor de conexões na estrutura Fat Tree.

A função recebe um vetor de conexões do n-ésimo processos gerentes e imprime todas as conexões contidas no vetor. Essa função te m como objetivo apenas debug e verificação das conexões.

Parâmetros

pid_t	*connections; Vetor de conexões de um dado processo gerente;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 374 do ficheiro dataStructures.c.

6.1.1.20 readHTConnections()

```
void readHTConnections (
            pid_t * connectios,
            int _id )
```

Realiza a leitura do vetor de conexões na estrutura Hypercube ou Torus.

A função recebe um vetor de conexões do n-ésimo procesos gerentes e imprime todas as conexões contidas no vetor. Essa função tem como objetivo apenas debug e verificação das conexões.

Parâmetros

pid_t	*connections; Vetor de conexões de um dado processo gerente;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 367 do ficheiro dataStructures.c.

6.1.1.21 readHyperTorus()

```
void readHyperTorus (
            hyperTorus * ht )
```

Função de leitura da estrutura simbólica Hypercube ou Torus.

A função realiza a leitura de toda estrutura Hypercube ou Torus. Esta função tem por intuito apenas debug e verificação das estruturas simbólicas.

Parâmetros

hyperTorus *(*p. **??***)*	∗ht; Cópia da estrutura simbólica Hypercube ou Torus;

Retorna

void;

Definido na linha 321 do ficheiro dataStructures.c.

6.1.1.22 readManQ()

```
void readManQ (
            manq * _manq )
```

Realiza a leitura da fila de escalonamento.

Essa função tem por objetivo apenas a debug e verificação da estrutura.

Parâmetros

manq	∗_manq; Cópia da fila manq;

Retorna

void;

Definido na linha 414 do ficheiro dataStructures.c.

6.1.1.23 readTree()

```
void readTree (
              fTree * _tree )
```

Função de leitura da árovre simbólica.

A função realiza a leitura de toda a arvóre simbólica criada. Esta função tem o intuito de ser apenas para debug e verificação da estrutura da árvore.

Parâmetros

fTree *(p. **??**)*	*tree; Cópia da árvore simbólica;

Retorna

void;

Definido na linha 192 do ficheiro dataStructures.c.

6.1.1.24 removeManQ()

```
manq * removeManQ (
              manq ** _manq )
```

Remove um processo da fila de escalonamento.

Os processo em estado "ready" são guardados na estrutura manq. Dessa forma, quando chega a vez de um processo gerente receber a ordem de execução, o ID do processo gerente é removido da lista manq e retornado para o escalonador.

Parâmetros

manq	**_manq; Ponteiro para estrutura do tipo manq;

Retorna

manq*; Retorna o primeiro item da fila manq;

Definido na linha 405 do ficheiro dataStructures.c.

6.1.1.25 removeProcess()

```
void removeProcess (
              execq ** queue )
```

Remove um job da lista.

Realiza a remoção de um job da lista de jobs. A remoção deve ocorrer após a conclusão do job pelos processos gerentes. Sempre o primeiro job da lista é removido.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;

Retorna

void;

Definido na linha 35 do ficheiro dataStructures.c.

6.1.1.26 updateDelays()

```
void updateDelays (
              execq ** queue )
```

Realiza update do delay dos jobs na lista.

Após a execução de um job ou a inserção de um novo, é necessário atualizar o delay dos jobs na lista de execução pois algum tempo se passou. A função irá realizar a atualização dos valores de acordo com o a hora que o job foi enviado e quanto tempo se passou desde então.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;

Retorna

void;

Definido na linha 66 do ficheiro dataStructures.c.

6.1.2 Documentação das variáveis

6.1.2.1 _jobs

```
int _jobs = 0
```

Contador de jobs executados.

Definido na linha 4 do ficheiro dataStructures.c.

6.2 Referência ao ficheiro dataStructures.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
```

Estruturas de Dados

- struct **execq**

  Define a estrutura da lista de jobs.
- struct **execd**

  Define a estrutura de jobs que já foram executados.
- struct **fTree**

  Define a estrutura da Fat Tree (foi implementada uma Binary Tree)
- struct **hyperTorus**

  Define a estrutura do Hybercube e Torus.
- struct **manq**

  Define a estrutura de processos livres.

Macros

- #define **CHILDS** 16

  Número de Gerentes para Hypercube e Torus.
- #define **FATCHILDS** 15

  Número de Gerentes para Fat Tree.

Definições de tipos

- typedef struct **execq** **execq**

  Define a estrutura da lista de jobs.
- typedef struct **execd** **execd**

  Define a estrutura de jobs que já foram executados.
- typedef struct **fTree** **fTree**

  Define a estrutura da Fat Tree (foi implementada uma Binary Tree)
- typedef struct **hyperTorus** **hyperTorus**

  Define a estrutura do Hybercube e Torus.
- typedef struct **manq** **manq**

  Define a estrutura de processos livres.

Funções

- void **createQueue** (**execq** ∗∗queue)

  Inicia a estrutura de dados do tipo execq.
- void **insertProcess** (**execq** ∗∗queue, char ∗_name, int _delay)

  Insere um novo job na fila.
- void **removeProcess** (**execq** ∗∗queue)

  Remove um job da lista.
- void **listProcesses** (**execq** ∗queue)

  Lista todos os jobs não concluídos.
- void **updateDelays** (**execq** ∗∗queue)

  Realiza update do delay dos jobs na lista.
- void **createExecD** (**execd** ∗∗done)

  Inicializador da estrutura execd.
- void **insertExecD** (**execd** ∗∗done, pid_t pid, char ∗program, time_t sent, time_t begin, time_t end)

  Insere informação sobre todos os processos que executaram um dado job.
- void **deleteExecD** (**execd** ∗∗done)

  Deleta toda a fila de processos executados.
- void **listExecD** (**execd** ∗done)

  Lista todos os processos na fila de processos executados.
- void **createFTree** (**fTree** ∗∗_tree)

  *Inicializador da estrutura do tipo **fTree** (*p. **??**).*
- void **definesTree** (**fTree** ∗∗_tree, int _parent, int ∗_node, int _level)

  Cria a estrtura da Fat Tree (Binary Tree)
- void **readTree** (**fTree** ∗_tree)

  Função de leitura da árovre simbólica.
- void **deleteTree** (**fTree** ∗∗_tree)

  Função padrão de deleção para estrutura Fat Tree (Binary Tree)
- pid_t ∗ **get_fTreeConnection** (**fTree** ∗_tree, int _id)

  Define as conexões de um dado processo gerente.
- void **createHyperTorus** (**hyperTorus** ∗∗_ht)

  *Inicializador da estrutura do tipo **hyperTorus** (*p. **??**).*
- void **definesHyper** (**hyperTorus** ∗∗_hyper, int _id)

  Define a estrutura simbólica Hypercube.
- void **definesTorus** (**hyperTorus** ∗∗_torus, int _id)

  Define a estrutura simbólica Torus.
- void **readHyperTorus** (**hyperTorus** ∗ht)

  Função de leitura da estrutura simbólica Hypercube ou Torus.
- void **deleteHyperTorus** (**hyperTorus** ∗∗ht)

  Função padrão de deleção das estruturas Hypercube e Torus.
- pid_t ∗ **get_htConnection** (**hyperTorus** ∗_hyper, int _id)

  Define as conexões de um dado processo processo gerente.
- void **readHTConnections** (pid_t ∗connections, int _id)

  Realiza a leitura do vetor de conexões na estrutura Hypercube ou Torus.
- void **readFTConnections** (pid_t ∗connections, int _id)

  Realiza a leitura do vetor de conexões na estrutura Fat Tree.
- void **createManQ** (**manq** ∗∗_manq)

  Inicializador padrão para estrutura do tipo manq.
- void **insertManQ** (**manq** ∗∗_manq, int _id)

  Insere processos, na ordem, de execução.
- **manq** ∗ **removeManQ** (**manq** ∗∗_manq)

  Remove um processo da fila de escalonamento.
- void **readManQ** (**manq** ∗_manq)

  Realiza a leitura da fila de escalonamento.

6.2.1 Documentação das macros

6.2.1.1 CHILDS

```
#define CHILDS 16
```

Número de Gerentes para Hypercube e Torus.

Definido na linha 9 do ficheiro dataStructures.h.

6.2.1.2 FATCHILDS

```
#define FATCHILDS 15
```

Número de Gerentes para Fat Tree.

Definido na linha 11 do ficheiro dataStructures.h.

6.2.2 Documentação dos tipos

6.2.2.1 execd

```
typedef struct execd execd
```

Define a estrutura de jobs que já foram executados.

6.2.2.2 execq

```
typedef struct execq execq
```

Define a estrutura da lista de jobs.

6.2.2.3 fTree

```
typedef struct fTree fTree
```

Define a estrutura da Fat Tree (foi implementada uma Binary Tree)

6.2.2.4 hyperTorus

```
typedef struct  hyperTorus  hyperTorus
```

Define a estrutura do Hybercube e Torus.

6.2.2.5 manq

```
typedef struct  manq  manq
```

Define a estrutura de processos livres.

6.2.3 Documentação das funções

6.2.3.1 createExecD()

```
void createExecD (
              execd ** done )
```

Inicializador da estrutura execd.

Inicializador padrão da estrutura do tipo execd

Parâmetros

execd	**queue; Ponteiro para estrutura do tipo execd;

Retorna

void;

Definido na linha 85 do ficheiro dataStructures.c.

6.2.3.2 createFTree()

```
void createFTree (
              fTree ** _tree )
```

Inicializador da estrutura do tipo **fTree** (p. **??**).

Inicializador padrão da estrutura do tipo **fTree** (p. **??**).

Parâmetros

fTree *(p. **??**)*	∗∗_tree; Ponteiro para estrutura do tipo **fTree** (p. **??**);

Retorna

void;

Definido na linha 149 do ficheiro dataStructures.c.

6.2.3.3 createHyperTorus()

```
void createHyperTorus (
              hyperTorus ** _ht )
```

Inicializador da estrutura do tipo **hyperTorus** (p. **??**).

Inicializador padrão da estrutura do tipo **hyperTorus** (p. **??**)

Parâmetros

hyperTorus *(p. **??**)*	∗∗_ht; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);

Retorna

void;

Definido na linha 235 do ficheiro dataStructures.c.

6.2.3.4 createManQ()

```
void createManQ (
              manq ** _manq )
```

Inicializador padrão para estrutura do tipo manq.

Parâmetros

manq	∗∗_manq; Ponteiro para a estrutura do tipo manq;

Retorna

void;

Definido na linha 381 do ficheiro dataStructures.c.

6.2.3.5 createQueue()

```
void createQueue (
              execq ** queue )
```

Inicia a estrutura de dados do tipo execq.

Inicializador padrão de uma estrutura de dados do tipo execq.

Parâmetros

execq	**queue; Ponteiro para uma estrutura do tipo execq

Retorna

void;

Definido na linha 6 do ficheiro dataStructures.c.

6.2.3.6 definesHyper()

```
void definesHyper (
              hyperTorus ** _hyper,
             int _id )
```

Define a estrutura simbólica Hypercube.

A função tem como responsabilidade definir a estrutura simbólica do tipo Hypercube. A função é chamada N vezes (no caso desse sistema, N=16), e Passa o número da chamada como argumento. O número é associado ao <_id> do gerente.

Na estrutura Hypercube, cada nó realiza 4 conexões (com exceção do nó 0 que realiza uma 5a conexão com o escalonador). Essa conexões são feitas com base no número do nó. Cada nó só se conecta com nós com <↩ _id> que tenha apenas 1 bit de diferença. Para isso, cada chamada da função roda um for para todos os 16 ID's possíveis na estrutura e realiza um XOR entre o <_id> do gerente e cada outro ID da estrutura. Caso o resultado seja 1, 2, 4 ou 8, significa que apenas 1 bit mudou e, portanto, há conexão.

As conexões são salvas em um vetor alocado em cada nó da estrutura simbólica.

Parâmetros

hyperTorus *(*p. **??***)*	**_hyper; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);
int	_id; ID do n-ésimo gerente;

Definido na linha 239 do ficheiro dataStructures.c.

6.2.3.7 definesTorus()

```
void definesTorus (
             hyperTorus ** _torus,
            int _id )
```

Define a estrutura simbólica Torus.

A função, de forma semelhante a **definesHyper()** (p. **??**), tem como objetivo define a estrutura simbólica Torus. A função será chamada N vezes (no caso desse sistema, N=16), e passa o número da chamada como argumento <_id>.

A regra da estrutura Torus é a seguinte:

- A estrutura é composta de 4 linhas com 4 gerentes em cada linha;
- A estrutura é composta de 4 colunas com 4 gerentes em cada coluna;
- Dessa forma, determina-se a coluna pela conta: col = _id%4. Se o nó estiver na coluna 0, ele se conecta com o primeiro nó a direita e com o terceiro nó a direita. Se o nó estiver na coluna 1 ou 2, se conecta com o primeiro nó a esquerda e o primeiro nó a direita. Se o nó estiver na coluna 3, se conecta com o primeiro nó a esquerda e o primeiro nó da coluna.
- Ainda, se o <_id> do nó subtraído de 4 for menor que 0 (nós na primeira linha) Eles se conectam com os nós na ultima linha ((_id+12)%16). Caso contrário, se conectam com o nó da linha de cima.
- Por fim, se o <_id> acrescido de 4 for maior que 15 (nós da última linha), eles se conectam com os nós da primeira linha ((_id+4)%16). Caso contrário, se conectam com o nó da linha de baixo.

As conexões são salvas em um vetor de cada nó da estrutura simbólica.

Parâmetros

hyperTorus *(p. **??**)*	**_torus; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);
int	_id; ID do n-ésimo gerente;

Retorna

void;

Definido na linha 272 do ficheiro dataStructures.c.

6.2.3.8 definesTree()

```
void definesTree (
             fTree ** _tree,
            int _parent,
```

```
            int * _node,
            int _level )
```

Cria a estrtura da Fat Tree (Binary Tree)

Função recursiva responsável pela criação da estrutura completa da árvore binária simbólica dos processos gerentes. Aqui serão calculadas todas as conexões que cada nó da árvore realiza, e cada nó irá receber um ID que, posteriormente, será associado a um processo gerente.

Parâmetros

fTree *(*p. **??***)*	∗∗_tree; Ponteiro para estrutura **fTree** (p. **??**);
int	_parent; ID do nó pai do nó sendo inserido na árvore;
int	∗_node; Ponteiro para o número do nó sendo inserido na árvore;
int	_level; Level atual da árvore sendo construído.

Definido na linha 153 do ficheiro dataStructures.c.

6.2.3.9 deleteExecD()

```
void deleteExecD (
            execd ** done )
```

Deleta toda a fila de processos executados.

Realiza a deleção e posterior "free" dos dados da fila de processos executados.

Parâmetros

execd	∗∗done; Ponteiro para estrutura do tipo execd;

Retorna

void;

Definido na linha 112 do ficheiro dataStructures.c.

6.2.3.10 deleteHyperTorus()

```
void deleteHyperTorus (
            hyperTorus ** _ht )
```

Função padrão de deleção das estruturas Hypercube e Torus.

Parâmetros

hyperTorus *(p. **??**)*	∗∗_ht; Ponteiro para estrutura do tipo **hyperTorus** (p. **??**);

Retorna

void;

Definido na linha 335 do ficheiro dataStructures.c.

6.2.3.11 deleteTree()

```
void deleteTree (
              fTree ** _tree )
```

Função padrão de deleção para estrutura Fat Tree (Binary Tree)

Parâmetros

fTree *(p. **??**)*	∗∗_tree; Ponteiro para estrutura do tipo **fTree** (p. **??**);

Retorna

void;

Definido na linha 205 do ficheiro dataStructures.c.

6.2.3.12 get_fTreeConnection()

```
pid_t* get_fTreeConnection (
              fTree * _tree,
             int _id )
```

Define as conexões de um dado processo gerente.

A função recebe a árvore simbólica gerada e o <_id> do processo gerente. Com isso, o processo gerente percorre a árvore simbólica até achar o nó com mesmo número que seu <_id> e, ao achar este nó, recupera as conexões que ele faz em forma de um vetor de tipo pid_t.

Parâmetros

fTree *(p. **??**)*	∗_tree; Cópia da árvore simbólica;
int	_id; ID do processo gerente;

Retorna

pid_t*; Vetor com todas as conexões que o n-ésimo gerente faz;

Definido na linha 214 do ficheiro dataStructures.c.

6.2.3.13 get_htConnection()

```
pid_t* get_htConnection (
              hyperTorus * _ht,
             int _id )
```

Define as conexões de um dado processo processo gerente.

A função busca na estrutura simbólica, usando o <_id> como base, o nó simbólico e as conexões que o mesmo faz no Hypercube ou Torus. Após encontrar o nó simbólico de mesmo ID que <_id> a função retorna o vetor de conexões do processo gerente.

Parâmetros

hyperTorus *(*p. **??***)*	*_hyper; Cópia da estrutura simbólica **hyperTorus** (p. **??**);
int	_id; ID do processo gerente;

Retorna

pid_t*; Vetor com as conexões do n-ésimo processo gerente;

Definido na linha 346 do ficheiro dataStructures.c.

6.2.3.14 insertExecD()

```
void insertExecD (
              execd ** done,
             pid_t pid,
             char * program,
             time_t sent,
             time_t begin,
             time_t end )
```

Insere informação sobre todos os processos que executaram um dado job.

Inserção das informações resultantes da execuçaõ de um job qualquer pelo sistema. A inserção segue o padrão de fila.

Parâmetros

execd	**done; Ponteiro para estrutura do tipo execd;

Parâmetros

pid_t	pid; PID do processo que realizou a n-ésima execução do programa;
char	∗program; Nome do programa executado;
time_t	sent; Hora de envio do job;
time_t	begin; Hora do início da execução do programa pelo processo;
time_t	end; Hora de conclusão da execução do programa pelo processo;

Definido na linha 89 do ficheiro dataStructures.c.

6.2.3.15 insertManQ()

```
void insertManQ (
              manq ** _manq,
             int _id )
```

Insere processos, na ordem, de execução.

O escalonador utiliza um escalonamento bastante simples: FIFO (First In First Out). Dessa forma, a estrutura "manq" guarda, na ordem, os processos prontos para execução de um dado job.

Parâmetros

manq	∗∗_manq; Ponteiro para a fila do escalonamento;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 385 do ficheiro dataStructures.c.

6.2.3.16 insertProcess()

```
void insertProcess (
              execq ** queue,
             char * _name,
             int _delay )
```

Insere um novo job na fila.

Realiza a inserção de um novo job recebido pelo escalonador. Inserção é ordenada pelo delay. Dessa forma, não segue uma estrutura típica de fila, e sim de uma lista ordenada.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;
char	*_name; Ponteiro para o nome do programa;
int	_delay; Delay, em segundos, para execução do job;

Retorna

void;

Definido na linha 10 do ficheiro dataStructures.c.

6.2.3.17 listExecD()

```
void listExecD (
              execd * done )
```

Lista todos os processos na fila de processos executados.

Realiza a impressão da informação sobre todos os processos que executaram algum programa durante toda a execução do escalonador.

Parâmetros

execd	*done; Cópia da estrutura do tipo execd;

Retorna

void;

Definido na linha 125 do ficheiro dataStructures.c.

6.2.3.18 listProcesses()

```
void listProcesses (
              execq * queue )
```

Lista todos os jobs não concluídos.

Realiza a impressão de todos os jobs que ainda estão para serem executados.

Parâmetros

execq	*queue; Cópia da estrutura do tipo execq

Retorna

void;

Definido na linha 49 do ficheiro dataStructures.c.

6.2.3.19 readFTConnections()

```
void readFTConnections (
            pid_t * connections,
            int _id )
```

Realiza a leitura do vetor de conexões na estrutura Fat Tree.

A função recebe um vetor de conexões do n-ésimo processos gerentes e imprime todas as conexões contidas no vetor. Essa função te m como objetivo apenas debug e verificação das conexões.

Parâmetros

pid_t	*connections; Vetor de conexões de um dado processo gerente;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 374 do ficheiro dataStructures.c.

6.2.3.20 readHTConnections()

```
void readHTConnections (
            pid_t * connections,
            int _id )
```

Realiza a leitura do vetor de conexões na estrutura Hypercube ou Torus.

A função recebe um vetor de conexões do n-ésimo procesos gerentes e imprime todas as conexões contidas no vetor. Essa função tem como objetivo apenas debug e verificação das conexões.

Parâmetros

pid_t	*connections; Vetor de conexões de um dado processo gerente;
int	_id; ID do n-ésimo processo gerente;

Retorna

void;

Definido na linha 367 do ficheiro dataStructures.c.

6.2.3.21 readHyperTorus()

```
void readHyperTorus (
            hyperTorus * _ht )
```

Função de leitura da estrutura simbólica Hypercube ou Torus.

A função realiza a leitura de toda estrutura Hypercube ou Torus. Esta função tem por intuito apenas debug e verificação das estruturas simbólicas.

Parâmetros

hyperTorus *(*p. **??***)*	∗ht; Cópia da estrutura simbólica Hypercube ou Torus;

Retorna

void;

Definido na linha 321 do ficheiro dataStructures.c.

6.2.3.22 readManQ()

```
void readManQ (
            manq * _manq )
```

Realiza a leitura da fila de escalonamento.

Essa função tem por objetivo apenas a debug e verificação da estrutura.

Parâmetros

manq	∗_manq; Cópia da fila manq;

Retorna

void;

Definido na linha 414 do ficheiro dataStructures.c.

6.2.3.23 readTree()

```
void readTree (
              fTree * _tree )
```

Função de leitura da árovre simbólica.

A função realiza a leitura de toda a arvóre simbólica criada. Esta função tem o intuito de ser apenas para debug e verificação da estrutura da árvore.

Parâmetros

fTree *(*p. **??***)*	*tree; Cópia da árvore simbólica;

Retorna

void;

Definido na linha 192 do ficheiro dataStructures.c.

6.2.3.24 removeManQ()

```
manq* removeManQ (
              manq ** _manq )
```

Remove um processo da fila de escalonamento.

Os processo em estado "ready" são guardados na estrutura manq. Dessa forma, quando chega a vez de um processo gerente receber a ordem de execução, o ID do processo gerente é removido da lista manq e retornado para o escalonador.

Parâmetros

manq	**_manq; Ponteiro para estrutura do tipo manq;

Retorna

manq*; Retorna o primeiro item da fila manq;

Definido na linha 405 do ficheiro dataStructures.c.

6.2.3.25 removeProcess()

```
void removeProcess (
              execq ** queue )
```

Remove um job da lista.

Realiza a remoção de um job da lista de jobs. A remoção deve ocorrer após a conclusão do job pelos processos gerentes. Sempre o primeiro job da lista é removido.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;

Retorna

void;

Definido na linha 35 do ficheiro dataStructures.c.

6.2.3.26 updateDelays()

```
void updateDelays (
              execq ** queue )
```

Realiza update do delay dos jobs na lista.

Após a execução de um job ou a inserção de um novo, é necessário atualizar o delay dos jobs na lista de execução pois algum tempo se passou. A função irá realizar a atualização dos valores de acordo com o a hora que o job foi enviado e quanto tempo se passou desde então.

Parâmetros

execq	**queue; Ponteiro para estrutura do tipo execq;

Retorna

void;

Definido na linha 66 do ficheiro dataStructures.c.

6.3 Referência ao ficheiro escalonador.c

```
#include "escalonador.h"
```

Macros

- #define **HYPER** "-h"

  Define o argumento para estrutura Hypercube.
- #define **TORUS** "-t"

  Define o argumento para estrutura Torus.
- #define **FAT** "-f"

  Define o argumento para estrutura Fat Tree.

Funções

- void **shutdown** ()

  Função para finalização do escalonador.
- void **new_schedule** ()

  Função responsável por agendar uma nova execução.
- void **send_pid** ()

  Função responsável por enviar o PID do escalonador.
- void **execute_job** ()

  Função responsável por executar jobs.
- void **delayed_scheduler** (int managers)

  Função base do escalonador.
- int **main** (int argc, char ∗argv[ ])

  Escalonador de processos via estruturas Hypercube, Torus ou Fat Tree.

Variáveis

- char ∗ **option**

  Variável que guarda a opção de estrutura do escalonador.
- **execq** ∗ **eq**

  Fila de jobs pra execução.
- **execd** ∗ **ed**

  Fila de jobs que já foram executados.
- int **_alarm**

  Variável que guarda o tempo - em segundos - para disparar o alarme.
- int **msgsmid**

  Variável que guarda o ID da fila de mensagens entre escalonador e gerentes.
- int **_managers**

  Variável que guarda a quantidade de gerentes para o escalonamento.
- int **finish** = 0

  Flag de finalização do sistema.
- pid_t ∗ **pids**

  Array com os PIDs de todos os gerentes.
- **manq** ∗ **_ready**

  Fila de gerentes prontos para executar.

6.3.1 Documentação das macros

6.3.1.1 FAT

```
#define FAT "-f"
```

Define o argumento para estrutura Fat Tree.

Definido na linha 8 do ficheiro escalonador.c.

6.3.1.2 HYPER

```
#define HYPER "-h"
```

Define o argumento para estrutura Hypercube.

Definido na linha 4 do ficheiro escalonador.c.

6.3.1.3 TORUS

```
#define TORUS "-t"
```

Define o argumento para estrutura Torus.

Definido na linha 6 do ficheiro escalonador.c.

6.3.2 Documentação das funções

6.3.2.1 delayed_scheduler()

```
delayed_scheduler (
             int managers )
```

Função base do escalonador.

A função delayed scheduler é a responsável pelo controle do escalonador. Aqui o escalonador inicialmente seta todos os processos gerentes como prontos, colocando-os na lista de "ready" e, feito isso, o escalonador fica bloqueado esperando novos jobs.

A função prepara o tratamento de quatro sinais distintos (SIGINT, SIGUSR1, SIGUSR2 e SIGALRM) cada qual responsável por uma tarefa distinta: shutdown, send_pid, new_scheduler e execute_job respectivamente.

Ao executar a função, caso não seja possível obter a fila, o escalonador é finalizado.

Parâmetros

int	managers;

Retorna

void

Definido na linha 173 do ficheiro escalonador.c.

6.3.2.2 execute_job()

```
void execute_job ( )
```

Função responsável por executar jobs.

A função é a que, de fato, executa os jobs escalonados. Inicialmente um contador é setado para zero e um laço é executado enquanto houver processos prontos para executar. Para cada processo "ready", o escalonador envia uma mensagem para o gerente "zero" redirecioná-la até o nó destino. A mensagem contém a ordem de execução do job.

Após o envio de todas as ordens de execução, o escalondor executa um laço infinito para receber as respostas dos gerentes, avisando que terminaram a execução do job. Cada execução é inserida em uma fila - que guarda pid, programa, hora de envio, hora de início da execução, hora de conclusão de execução -, e quando o contador atinge a marca de 15/16 (a depender da estrutura escolhida para execução do escalonador), é imprimida a informação de resumo do job e o mesmo é removido da lista de jobs. O laço executa um break para concluir a execução e aguardar a próxima execução.

O escalonador ficará bloqueado esperando a próxima execução, ou inserção de jobs, caso a fila de execução esteja vazia.

Retorna

void

Definido na linha 117 do ficheiro escalonador.c.

6.3.2.3 main()

```
int main (
              int argc,
              char * argv[] )
```

Escalonador de processos via estruturas Hypercube, Torus ou Fat Tree.

A função principal do escalonador. Aqui ocorre toda a preparação para a execução do escalonador e de seus processos gerentes. A execução começa recebendo o parâmetro de estrutura (-h, -t ou -f). O arquivo "jobs.txt" é criado para guardar o contador de jobs enviados (o propósito é para utilização do programa "execucao_postergada"). Feito isso, os canais de comunicação são criados - um para comunicação com o programa "execucao_postergada" e "shutdown", outro para os processos gerentes e um terceiro exclusivo para envio dos jobs. O escalonador envia mensagem inicial, contendo seu PID, para comunicação, via signals, com "execucao_postergada" ou com o "shutdown". O escalonador verifica o tipo de estrutura que foi escolhido para execução - hypercube, torus ou fat tree -, e cria os grafos ou a árvore simbólica dos nós gerentes.

Após a criação das estruturas simbólicas, o escalonador realiza os N forks para criação dos processos gerentes. Para cada chamada fork o PID do filho é armazenado em um vetor, o "_id" é atualizado para o próximo filho e, caso alguma chamada fork ocasione erro, os processos que já tenham sido criados são terminados enviando um SIGKILL a eles.

Os filhos criados executam uma função para receber, da estrutura simbólica, as conexões que eles fazem e, a partir dai, seguem para execução da função de gerenciamento. O pai segue para execução da função de escalonador postergado.

Ao retornar da função de escalonamento (devido ao shutdown), o processo irá remover as estruturas simbólicas da memória, enviar uma mensagem a todos os gerentes para encerrarem a execução e esperar pelo exit dos mesmos. Por fim, as filas de mensagem serão removidas e o processo finalizado.

Parâmetros

argc;	Quantidade de argumentos passados na CLI
∗argv[];	Argumentos passados na CLI

Definido na linha 227 do ficheiro escalonador.c.

6.3.2.4 new_schedule()

```
void new_schedule ( )
```

Função responsável por agendar uma nova execução.

A função recebe uma mensagem do programa "execucao_postergada". O recebimento é bloqueado de forma que, se a mensagem não estiver disponível, o escalonador fica bloqueado. Após a recepção da mensagem, caso a fila de jobs esteja vazia, um alarme inicial é setado. Caso contrário, o novo job é inserido na fila, os delays são atualizados de acordo, e a lista de jobs é imprimida. Caso o job sendo inserido tenha um delay menor do que o primeiro job que estava em primeiro da fila anteriormente, o alarme é atualizado. Caso o delay seja 0, a função executa um kill, enviando um SIGALRM para dar inicio imediato a execução do job.

Retorna

void

Definido na linha 75 do ficheiro escalonador.c.

6.3.2.5 send_pid()

```
void send_pid ( )
```

Função responsável por enviar o PID do escalonador.

Ao iniciar a execução do escalonador, a função **main()** (p. **??**) envia o PID do escalonador para a fila que fará a comunicação com o programa "execucao_postergada". Toda vez que uma nova ordem de job é recebida, o escalonador recebe um sinal e deverá reenviar o seu PID para tal fila, de forma que quando a próxima execução do "execucao_postergada" ocorrer, terá o PID para a comunicação disponível.

Retorna

void

Definido na linha 107 do ficheiro escalonador.c.

6.3.2.6 shutdown()

```
void shutdown ( )
```

Função para finalização do escalonador.

A função shutdown é a responsável por finalizar a execução do escalonador. Ao receber um sinal do tipo **SIGINT**, o escalonador deve printar as informações de resumo da execução, bem como quaisquer jobs não executados.

Além disso, a flag de finalização é setada para "1" e a execução do escalonador é concluída, retornando para função main.

No sumário constam as seguintes informações:

- Processos que não foram executados (número do job, nome do programa e delay);
- Todos os processos executados por todos os gerentes (PID, nome do programa, hora de recepção, hora de início da execução, hora de término da execução e makespan);

Retorna

void

Definido na linha 55 do ficheiro escalonador.c.

6.3.3 Documentação das variáveis

6.3.3.1 _alarm

```
int _alarm
```

Variável que guarda o tempo - em segundos - para disparar o alarme.

Definido na linha 43 do ficheiro escalonador.c.

6.3.3.2 _managers

```
int _managers
```

Variável que guarda a quantidade de gerentes para o escalonamento.

Definido na linha 47 do ficheiro escalonador.c.

6.3.3.3 _ready

```
manq* _ready
```

Fila de gerentes prontos para executar.

Definido na linha 53 do ficheiro escalonador.c.

6.3.3.4 ed

```
execd* ed
```

Fila de jobs que já foram executados.

Definido na linha 41 do ficheiro escalonador.c.

6.3.3.5 eq

```
execq* eq
```

Fila de jobs pra execução.

Definido na linha 39 do ficheiro escalonador.c.

6.3.3.6 finish

```
int finish = 0
```

Flag de finalização do sistema.

Definido na linha 49 do ficheiro escalonador.c.

6.3.3.7 msgsmid

```
int msgsmid
```

Variável que guarda o ID da fila de mensagens entre escalonador e gerentes.

Definido na linha 45 do ficheiro escalonador.c.

6.3.3.8 option

```
char* option
```

Variável que guarda a opção de estrutura do escalonador.

Definido na linha 37 do ficheiro escalonador.c.

6.3.3.9 pids

```
pid_t* pids
```

Array com os PIDs de todos os gerentes.

Definido na linha 51 do ficheiro escalonador.c.

6.4 Referência ao ficheiro escalonador.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <signal.h>
#include <string.h>
#include "dataStructures.h"
#include "managerProcess.h"
```

Funções

- void **delayed_scheduler** (int managers)

  Função base do escalonador.
- void **shutdown** ()

  Função para finalização do escalonador.
- void **new_schedule** ()

  Função responsável por agendar uma nova execução.
- void **send_pid** ()

  Função responsável por enviar o PID do escalonador.
- void **execute_job** ()

  Função responsável por executar jobs.

6.4.1 Documentação das funções

6.4.1.1 delayed_scheduler()

```
void delayed_scheduler (
             int managers )
```

Função base do escalonador.

A função delayed scheduler é a responsável pelo controle do escalonador. Aqui o escalonador inicialmente seta todos os processos gerentes como prontos, colocando-os na lista de "ready" e, feito isso, o escalonador fica bloqueado esperando novos jobs.

A função prepara o tratamento de quatro sinais distintos (SIGINT, SIGUSR1, SIGUSR2 e SIGALRM) cada qual responsável por uma tarefa distinta: shutdown, send_pid, new_scheduler e execute_job respectivamente.

Ao executar a função, caso não seja possível obter a fila, o escalonador é finalizado.

Parâmetros

int	managers;

Retorna

void

Definido na linha 173 do ficheiro escalonador.c.

6.4.1.2 execute_job()

```
void execute_job ( )
```

Função responsável por executar jobs.

A função é a que, de fato, executa os jobs escalonados. Inicialmente um contador é setado para zero e um laço é executado enquanto houver processos prontos para executar. Para cada processo "ready", o escalonador envia uma mensagem para o gerente "zero" redirecioná-la até o nó destino. A mensagem contém a ordem de execução do job.

Após o envio de todas as ordens de execução, o escalondor executa um laço infinito para receber as respostas dos gerentes, avisando que terminaram a execução do job. Cada execução é inserida em uma fila - que guarda pid, programa, hora de envio, hora de início da execução, hora de conclusão de execução -, e quando o contador atinge a marca de 15/16 (a depender da estrutura escolhida para execução do escalonador), é imprimida a informação de resumo do job e o mesmo é removido da lista de jobs. O laço executa um break para concluir a execução e aguardar a próxima execução.

O escalonador ficará bloqueado esperando a próxima execução, ou inserção de jobs, caso a fila de execução esteja vazia.

Retorna

void

Definido na linha 117 do ficheiro escalonador.c.

6.4.1.3 new_schedule()

```
void new_schedule ( )
```

Função responsável por agendar uma nova execução.

A função recebe uma mensagem do programa "execucao_postergada". O recebimento é bloqueado de forma que, se a mensagem não estiver disponível, o escalonador fica bloqueado. Após a recepção da mensagem, caso a fila de jobs esteja vazia, um alarme inicial é setado. Caso contrário, o novo job é inserido na fila, os delays são atualizados de acordo, e a lista de jobs é imprimida. Caso o job sendo inserido tenha um delay menor do que o primeiro job que estava em primeiro da fila anteriormente, o alarme é atualizado. Caso o delay seja 0, a função executa um kill, enviando um SIGALRM para dar inicio imediato a execução do job.

Retorna

void

Definido na linha 75 do ficheiro escalonador.c.

6.4.1.4 send_pid()

```
void send_pid ( )
```

Função responsável por enviar o PID do escalonador.

Ao iniciar a execução do escalonador, a função **main()** (p. **??**) envia o PID do escalonador para a fila que fará a comunicação com o programa "execucao_postergada". Toda vez que uma nova ordem de job é recebida, o escalonador recebe um sinal e deverá reenviar o seu PID para tal fila, de forma que quando a próxima execução do "execucao_postergada" ocorrer, terá o PID para a comunicação disponível.

Retorna

void

Definido na linha 107 do ficheiro escalonador.c.

6.4.1.5 shutdown()

```
void shutdown ( )
```

Função para finalização do escalonador.

A função shutdown é a responsável por finalizar a execução do escalonador. Ao receber um sinal do tipo **SIGINT**, o escalonador deve printar as informações de resumo da execução, bem como quaisquer jobs não executados.

Além disso, a flag de finalização é setada para "1" e a execução do escalonador é concluída, retornando para função main.

No sumário constam as seguintes informações:

- Processos que não foram executados (número do job, nome do programa e delay);
- Todos os processos executados por todos os gerentes (PID, nome do programa, hora de recepção, hora de início da execução, hora de término da execução e makespan);

Retorna

void

Definido na linha 55 do ficheiro escalonador.c.

6.5 Referência ao ficheiro execucao_postergada.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/signal.h>
#include <sys/shm.h>
#include "msgQueue.h"
```

Funções

- int **main** (int argc, char ∗argv[ ])

  Programa para envio de job de execução postergada.

6.5.1 Documentação das funções

6.5.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Programa para envio de job de execução postergada.

O programa "execucao_postergada" deve receber exatamente dois argumentos: nome do programa e delay (em segundos). O programa inicia verificando quantos argumentos foram inseridos na CLI e, caso o número seja diferene de dois, o programa imprime que o número de argumentos está errado e encerra. Caso contrário, a execução segue.

Após a checagem de número de argumentos, o programa verifica se o segundo argumento (delay) contém apenas números. Caso o argumento contenha outros caracteres, o programa imprime que o argumento deve conter apenas números e encerra. Caso contrário, a execução continua.

Feito isso, o programa verifica se obteve um identificador de fila de mensagens e, em caso negativo, encerra o programa. Em caso positivo, o programa busca a mensagem na fila que irá conter o PID do escalonador para o envio da mensagem contendo o programa e o delay para execução. Após a recepção dessa mensagem, um sinal é enviado para escalonador para que uma nova mensagem, contendo seu PID, seja enviado para futura utilização do "execucao_postergada".

O programa monta a mensagem contendo o nome do programa e o delay de execução e envia para a fila de mensagens para que o escalonador inicie o job, acessa o arquivo "jobs.txt" para ler o contador de jobs e, por fim, imprime a informação do job enviado para execução.

Caso o identificador não seja obtido, o programa imprime um erro e encerra.Caso ocorra um erro de envio, o programa informa com um print.

Parâmetros

int	argc; Quantidade de argumentos passados na CLI
char	∗argv[]; Argumentos passados na CLI

Retorna

int;

Definido na linha 41 do ficheiro execucao_postergada.c.

6.6 Referência ao ficheiro hello.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

Funções

- int **main** ()

6.6.1 Documentação das funções

6.6.1.1 main()

```
int main ( )
```

Definido na linha 5 do ficheiro hello.c.

6.7 Referência ao ficheiro managerProcess.c

```
#include "managerProcess.h"
```

Funções

- void **manager_exit** ()

  Executa a finalização do processo gerente.
- void **manager_process** (int _id, pid_t ∗connections, char ∗ **option**)

  Função de gerenciamento dos processos.

6.7.1 Documentação das funções

6.7.1.1 manager_exit()

```
void manager_exit ( )
```

Executa a finalização do processo gerente.

A função "manager_process" executa um laço infinito, esperando sempre pela recepção de mensagens para redirecionamento ou execução. Dessa forma, ao receber um sinal **SIGQUIT** do escalonador, a execução é desviada para esta função, que executa um exit(0) para encerrar o gerente.

Retorna

void

Definido na linha 3 do ficheiro managerProcess.c.

6.7.1.2 manager_process()

```
void manager_process (
            int _id,
            pid_t * connections,
            char * option )
```

Função de gerenciamento dos processos.

Visão Geral

A função "managerProcess" executa um laço infinito para toda a execução. Dentro, todos os tratamentos de mensagem e execução dos gerentes são realizados.

Ao receber uma mensagem, é verificado o destino da mesma. Se o destino for "-1", O gerente sabe que a mensagem está fluindo no sentido do escalonador. Caso o destino seja >= 0, o gerente sabe que a mensagem está fluindo no sentido dos gerentes.

Caso a mensagem recebida tenha como destino um outro gerente - que não o gerente que executou a recepção -, a mensagem é redirecionada para o gerente da lista de conexão mais próximo do destino, ou o próprio destino. Caso a mensagem tenha como destino o gerente que a recebeu, ele faz o tratamento.

Como dito anteriormente, as mensagens fluem em dois sentidos. No caso das mensagens que fluem do escalonador para os gerentes, ao encontrarem os destinos certos, os gerentes dão início a execução do programa solicitado pelo usuário e, após a execução ou erro da mesma, gerente envia as informações resultantes para o escalonador.

A execução é realizada da seguinte forma: o gerente executa um fork. O filho irá executar o programa de fato, enquanto que o gerente salva a hora de inicio da execução, aguarda a saída do filho, salva a hora de término da execução e envia a mensagem de conclusão da execução para o escalonador.

No caso das mensagens fluindo na direção do escalonador, os gerentes apenas fazem o redirecionamento sem maiores tratamentos.

Cálculo de rotas: As rotas são traçadas em dois sentidos: escalonador -> gerentes; gerentes -> escalonador. Para ambos, exite uma subdivisão entre estruturas. Fat Tree e Hypercube executam um cálculo, Torus executa um cálculo diferente. Inicialmente havia sido desenvolvido apenas um tipo de cálculo para cada sentido porém, isso ocasionou congestionamento na estrutura Torus.

- **Fat Tree e Hybercube**
  - **Escalonador => Gerentes:** A estratégia é verificar se a i-ésima conexão do gerente é menor que o destino e diferente de -1 (identificador do escalonador). Caso as duas condições sejam verdadeiras "aux" recebe o valor da i-ésima conexão. Caso a i-ésima conexão seja o destino, "aux" recebe o valor e sai do loop FOR. Feito isso, a mensagem é redirecionada para o próximo nó da estrutura ou para o nó destino.
  - **Gerentes => Escalonador:** Inicialmente a variável "aux" é setada para um valor alto arbitrário. Um for é executado para verificar qual das conexões do nó é a menor (mais próxima do escalonador). "aux" recebe a menor conexão. Caso alguma das conexões do nó seja -1 (escalonador), "aux" recebe o o valor da conexão com o escalonador e uma saída do for é executada. Ao final da execução do for a mensagem é enviada ao escalonador ou é redirecionada para um nó mais próxima do escalonador. obs: A estratégia pode ser resumida como uma tentativa de se alcançar o nó "0" para, então, enviar ao escalonador. A menor conexão é escolhida pois será numéricamente mais próxima de "0".
- **Torus**
  - **Escalonador => Gerentes:** Inicialmente a variável "aux" recebe o valor do destino módulo "4" e "aux2" recebe "0". A ideia é verificar em qual coluna o destino se encontra. Um for é executado para verificar se alguma das n conexões do nó está na mesma coluna. Caso esteja ou seja o nó destino, "aux2" recebe o valor da conexão. Após o for, é verificado se "aux2" ainda é "0". Em caso positivo, "aux2" recebe o ID do nó acrescido de "1". É verificado se o valor de "aux2" módulo "4" é igual a "0". Em caso positivo, "aux2" recebe o resto da divisão de "aux2" por "4". A ideia é manter o redirecionamento na mesma coluna mas, caso não seja possível, a mensagem é redirecionada para o nó a direita do nó atual.
  - **Gerentes => Escalonador:** Inicialmente a variável "aux" começa com um valor alto arbitrário. Um laço for é executado para verificar todas as conexões do nó. O nó verifica se a i-ésima conexão que faz está na coluna "0", por meio de um módulo "4". Caso esteja e caso a conexão seja um ID menor que o seu próprio, "aux" recebe o ID dessa conexão. Caso a conexão tenha o destino, "aux" é setada para o destino final. Ao final do laço for, caso a variável "aux" ainda contenha o valor arbitrário inicial, o nó irá definí-la como a conexão diretamente a esquerda do nó. Caso o nó seja "0", aux é setado para o valor "-1" que simboliza o escalonador.

Parâmetros

int	_id; ID do gerente (entre 0 e 15);
pid_t	*connections; Array com os id's dos gerentes ao qual se conecta;
char	*option; Define a opção de estrutura utilizada pelo escalonador;

Retorna

void;

Definido na linha 7 do ficheiro managerProcess.c.

6.8 Referência ao ficheiro managerProcess.h

```
#include <sys/types.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>
#include <sys/signal.h>
#include <sys/wait.h>
#include "msgQueue.h"
```

Funções

- void **manager_exit** ()

  Executa a finalização do processo gerente.
- void **manager_process** (int _id, pid_t ∗connections, char ∗ **option**)

  Função de gerenciamento dos processos.

6.8.1 Documentação das funções

6.8.1.1 manager_exit()

```
void manager_exit ( )
```

Executa a finalização do processo gerente.

A função "manager_process" executa um laço infinito, esperando sempre pela recepção de mensagens para redirecionamento ou execução. Dessa forma, ao receber um sinal **SIGQUIT** do escalonador, a execução é desviada para esta função, que executa um exit(0) para encerrar o gerente.

Retorna

void

Definido na linha 3 do ficheiro managerProcess.c.

6.8.1.2 manager_process()

```
void manager_process (
            int _id,
            pid_t * connections,
            char * option )
```

Função de gerenciamento dos processos.

Visão Geral

A função "managerProcess" executa um laço infinito para toda a execução. Dentro, todos os tratamentos de mensagem e execução dos gerentes são realizados.

Ao receber uma mensagem, é verificado o destino da mesma. Se o destino for "-1", O gerente sabe que a mensagem está fluindo no sentido do escalonador. Caso o destino seja >= 0, o gerente sabe que a mensagem está fluindo no sentido dos gerentes.

Caso a mensagem recebida tenha como destino um outro gerente - que não o gerente que executou a recepção -, a mensagem é redirecionada para o gerente da lista de conexão mais próximo do destino, ou o próprio destino. Caso a mensagem tenha como destino o gerente que a recebeu, ele faz o tratamento.

Como dito anteriormente, as mensagens fluem em dois sentidos. No caso das mensagens que fluem do escalonador para os gerentes, ao encontrarem os destinos certos, os gerentes dão início a execução do programa solicitado pelo usuário e, após a execução ou erro da mesma, gerente envia as informações resultantes para o escalonador.

A execução é realizada da seguinte forma: o gerente executa um fork. O filho irá executar o programa de fato, enquanto que o gerente salva a hora de inicio da execução, aguarda a saída do filho, salva a hora de término da execução e envia a mensagem de conclusão da execução para o escalonador.

No caso das mensagens fluindo na direção do escalonador, os gerentes apenas fazem o redirecionamento sem maiores tratamentos.

Cálculo de rotas: As rotas são traçadas em dois sentidos: escalonador -> gerentes; gerentes -> escalonador. Para ambos, exite uma subdivisão entre estruturas. Fat Tree e Hypercube executam um cálculo, Torus executa um cálculo diferente. Inicialmente havia sido desenvolvido apenas um tipo de cálculo para cada sentido porém, isso ocasionou congestionamento na estrutura Torus.

- **Fat Tree e Hybercube**
  - **Escalonador => Gerentes:** A estratégia é verificar se a i-ésima conexão do gerente é menor que o destino e diferente de -1 (identificador do escalonador). Caso as duas condições sejam verdadeiras "aux" recebe o valor da i-ésima conexão. Caso a i-ésima conexão seja o destino, "aux" recebe o valor e sai do loop FOR. Feito isso, a mensagem é redirecionada para o próximo nó da estrutura ou para o nó destino.
  - **Gerentes => Escalonador:** Inicialmente a variável "aux" é setada para um valor alto arbitrário. Um for é executado para verificar qual das conexões do nó é a menor (mais próxima do escalonador). "aux" recebe a menor conexão. Caso alguma das conexões do nó seja -1 (escalonador), "aux" recebe o o valor da conexão com o escalonador e uma saída do for é executada. Ao final da execução do for a mensagem é enviada ao escalonador ou é redirecionada para um nó mais próxima do escalonador. obs: A estratégia pode ser resumida como uma tentativa de se alcançar o nó "0" para, então, enviar ao escalonador. A menor conexão é escolhida pois será numéricamente mais próxima de "0".
- **Torus**
  - **Escalonador => Gerentes:** Inicialmente a variável "aux" recebe o valor do destino módulo "4" e "aux2" recebe "0". A ideia é verificar em qual coluna o destino se encontra. Um for é executado para verificar se alguma das n conexões do nó está na mesma coluna. Caso esteja ou seja o nó destino, "aux2" recebe o valor da conexão. Após o for, é verificado se "aux2" ainda é "0". Em caso positivo, "aux2" recebe o ID do nó acrescido de "1". É verificado se o valor de "aux2" módulo "4" é igual a "0". Em caso positivo, "aux2" recebe o resto da divisão de "aux2" por "4". A ideia é manter o redirecionamento na mesma coluna mas, caso não seja possível, a mensagem é redirecionada para o nó a direita do nó atual.
  - **Gerentes => Escalonador:** Inicialmente a variável "aux" começa com um valor alto arbitrário. Um laço for é executado para verificar todas as conexões do nó. O nó verifica se a i-ésima conexão que faz está na coluna "0", por meio de um módulo "4". Caso esteja e caso a conexão seja um ID menor que o seu próprio, "aux" recebe o ID dessa conexão. Caso a conexão tenha o destino, "aux" é setada para o destino final. Ao final do laço for, caso a variável "aux" ainda contenha o valor arbitrário inicial, o nó irá definí-la como a conexão diretamente a esquerda do nó. Caso o nó seja "0", aux é setado para o valor "-1" que simboliza o escalonador.

Parâmetros

int	_id; ID do gerente (entre 0 e 15);
pid_t	*connections; Array com os id's dos gerentes ao qual se conecta;
char	*option; Define a opção de estrutura utilizada pelo escalonador;

Retorna

void;

Definido na linha 7 do ficheiro managerProcess.c.

6.9 Referência ao ficheiro msgQueue.c

```
#include "msgQueue.h"
```

Funções

- int **create_channel** (int key)
  Função para criar filas de mensagens.
- int **get_channel** (int key)
  Retorna o identificador de uma fila de mensagens.
- int **delete_channel** (int msg_id)
  Delata uma fila de mensagens.

6.9.1 Documentação das funções

6.9.1.1 create_channel()

```
int create_channel (
            int key )
```

Função para criar filas de mensagens.

A função é responsável por criar canais de comunicação (fila de mensages) com base em uma chave <key>. O msgget é executado utilizando a <key> fornecida e utilizando a flag de criação **IPC_CREAT** e **IPC_EXCL** que torna o canal único para a chave fornecida.

Parâmetros

int	key; Chave para criação de uma fila de mensagens.

Retorna

int; Retorna o identificador da fila ou -1 em caso de erro.

Definido na linha 3 do ficheiro msgQueue.c.

6.9.1.2 delete_channel()

```
int delete_channel (
            int msg_id )
```

Delata uma fila de mensagens.

A função recebe um identificador <msg_id> para deletar uma fila. Caso o identificador seja negativo, a função retorna -1 pois a fila não existe. Caso contrário, é executado msgctl para remover a fila identificada por <msg_id>. Retorna 0 em caso de sucesso ou -1 em caso de erro.

Parâmetros

int	msg_id; Identificador da fila de mensagens

Retorna

int; Retorna 0 em caso de sucesso ou -1 em caso de erro.

Definido na linha 13 do ficheiro msgQueue.c.

6.9.1.3 get_channel()

```
int get_channel (
            int key )
```

Retorna o identificador de uma fila de mensagens.

A função é responsável por obter o identificador de uma fila de mensagens com base na chave <key>. Caso a fila já tenha sido criada, é retornado o identificador da mesma. Caso contrário, é retornado -1.

Parâmetros

int	key; Chave identificadora da fila de mensagens

Retorna

int; Retorna o identificador da fila ou -1 em caso de erro.

Definido na linha 8 do ficheiro msgQueue.c.

6.10 Referência ao ficheiro msgQueue.h

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
#include <sys/stat.h>
```

Estruturas de Dados

- struct **msg_packet**

  Pacote de mensagem utilizado para comunicação entre escalonador, execucao_postergada e gerentes.
- struct **pid_packet**

  Pacote de mensagem para comunicação entre escalonador, execucao_postergada e shutdown.

Macros

- #define **MQ_SD** 140080279

  Chave para o canal de comunicação entre escalonador, execucao_postergada e shutdown.
- #define **MQ_SM** 0x8349

  Chave para o canal de comunicação entre escalonador e gerentes.
- #define **MQ_SJ** 1400802792

  Chave para o canal de envio de Jobs.

Definições de tipos

- typedef struct **msg_packet** **msg_packet**

  Pacote de mensagem utilizado para comunicação entre escalonador, execucao_postergada e gerentes.
- typedef struct **pid_packet** **pid_packet**

  Pacote de mensagem para comunicação entre escalonador, execucao_postergada e shutdown.

Funções

- int **create_channel** (int key)

  Função para criar filas de mensagens.
- int **get_channel** (int key)

  Retorna o identificador de uma fila de mensagens.
- int **delete_channel** (int msg_id)

  Delata uma fila de mensagens.

6.10.1 Documentação das macros

6.10.1.1 MQ_SD

```
#define MQ_SD 140080279
```

Chave para o canal de comunicação entre escalonador, execucao_postergada e shutdown.

Definido na linha 9 do ficheiro msgQueue.h.

6.10.1.2 MQ_SJ

```
#define MQ_SJ 1400802792
```

Chave para o canal de envio de Jobs.

Definido na linha 13 do ficheiro msgQueue.h.

6.10.1.3 MQ_SM

```
#define MQ_SM 0x8349
```

Chave para o canal de comunicação entre escalonador e gerentes.

Definido na linha 11 do ficheiro msgQueue.h.

6.10.2 Documentação dos tipos

6.10.2.1 msg_packet

```
typedef struct msg_packet msg_packet
```

Pacote de mensagem utilizado para comunicação entre escalonador, execucao_postergada e gerentes.

6.10.2.2 pid_packet

```
typedef struct pid_packet pid_packet
```

Pacote de mensagem para comunicação entre escalonador, execucao_postergada e shutdown.

6.10.3 Documentação das funções

6.10.3.1 create_channel()

```
int create_channel (
            int key )
```

Função para criar filas de mensagens.

A função é responsável por criar canais de comunicação (fila de mensages) com base em uma chave <key>. O msgget é executado utilizando a <key> fornecida e utilizando a flag de criação **IPC_CREAT** e **IPC_EXCL** que torna o canal único para a chave fornecida.

Parâmetros

int	key; Chave para criação de uma fila de mensagens.

Retorna

int; Retorna o identificador da fila ou -1 em caso de erro.

Definido na linha 3 do ficheiro msgQueue.c.

6.10.3.2 delete_channel()

```
int delete_channel (
            int msg_id )
```

Delata uma fila de mensagens.

A função recebe um identificador <msg_id> para deletar uma fila. Caso o identificador seja negativo, a função retorna -1 pois a fila não existe. Caso contrário, é executado msgctl para remover a fila identificada por <msg_id>. Retorna 0 em caso de sucesso ou -1 em caso de erro.

Parâmetros

int	msg_id; Identificador da fila de mensagens

Retorna

int; Retorna 0 em caso de sucesso ou -1 em caso de erro.

Definido na linha 13 do ficheiro msgQueue.c.

6.10.3.3 get_channel()

```
int get_channel (
            int key )
```

Retorna o identificador de uma fila de mensagens.

A função é responsável por obter o identificador de uma fila de mensagens com base na chave <key>. Caso a fila já tenha sido criada, é retornado o identificador da mesma. Caso contrário, é retornado -1.

Parâmetros

int	key; Chave identificadora da fila de mensagens

Retorna

int; Retorna o identificador da fila ou -1 em caso de erro.

Definido na linha 8 do ficheiro msgQueue.c.

6.11 Referência ao ficheiro README.md

6.12 Referência ao ficheiro shutdown.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/signal.h>
#include "msgQueue.h"
```

Funções

- int **main** (int argc, char ∗argv[ ])

  Programa responsável por finalizar a execução do escalonador postergado.

6.12.1 Documentação das funções

6.12.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Programa responsável por finalizar a execução do escalonador postergado.

O programa "shutdown" trabalha recebendo o PID do escalonador postergado, por meio de uma fila de mensagens, e, com o PID, envia um sinal **SIGINT** para o escalonador. Após o envio do sinal para encerramento da execução do escalonador, o programa irá acessar o arquivo "jobs.txt" e irá resetar o contador.

O escalonador, por sua vez, trata o sinal, desviando a execução para rotina de tratamento que irá encerrar a execução do programa.

Caso algum erro ocorra na recepção ou no envio, o usuário será avisado.

IMPORTANTE: O programa shutdown só deve ser executado se o escalonador não estiver executando um job ou se estiver esperando para começar uma execução. Não utilize o comando caso exista uma execução em curso!

Parâmetros

int	argc; Número de argumentos passados na CLI
char	∗argv[]; Argumentos passados na CLI

Retorna

0;

Definido na linha 26 do ficheiro shutdown.c.