



## INE5412-04208A (20162) - Sistemas Operacionais I

Dashboard ► Graduação ► Ciências da Computação (208) ► 20162 ►  
INE5412-04208A (20162) ► SC Project VPL - Theme 1: Simulation of Operating ...

Description

Submission view

### SC Project VPL - Theme 1: Simulation of Operating Systems Algorithms

**Due date:** quarta, 31 agosto 2016, 9:00

**Maximum number of files:** 50

**Type of work:** Individual work

Aqui vai uma explicação rápida sobre cada recurso disponível no simulador:

A primeira coisa que deve ser feita, é olhar a classe **Traits.h**, nela estão uma série de parâmetros que são lidos por várias das classes presentes no simulador, e é ela que faz um ajuste fino sobre como o simulador se comportará. Após feito isso, deve-se modificar o parametro *problemChosen*, dentro do bloco **Traits<Model>**, ao fim da classe. Digite neste parametro, o nome do tema que estarão fazendo, selecionado dentro do enum logo acima dele. ex:

```
static const unsigned int problemChosen = BestFit111;
```

Existe uma classe estática chamada **OperatingSystem.h**, nela há funções estáticas que retornam um ponteiro para alguns dos módulos que estão disponíveis para os alunos. Estes são:

```
static Scheduler* Process_Scheduler()
```

```
static FileSystem* File_System()
```

```
static Scheduler* Disk_Scheduler()
```

```
static MemoryManager* Memory_Manager()
```

Cada uma delas retorna um ponteiro para o respectivo módulo, para que se tenha acesso às suas funções, que podem ser úteis na confecção de alguns Temas de SC.

Dentre todas as classes aqui descritas, que estarão disponíveis à todos, os alunos devem se abster de mexer nas classes onde o nome se inicia com Module, e da classe Simulator, pois estas classes são de função vital do simulador e não tem nada a ver com os trabalhos. O simulador funcionará de tal forma que ele simulará um software rodando em um software (como um interpretador), ele utilizará instruções MIPS como programa de boot, e uma chamada syscall para executar o primeiro processo, este programa pode sim ser modificado, porém deve-se manter de mexer nestas classes acima citadas, em todas as outras aqui descritas a edição é possível onde se ver necessário.

(Nota: todas estas classes citadas incluem arquivos .h e .cpp e estão em negrito, os métodos e funções estão em itálico e em sua maioria são públicos exceto se dito o contrário, um asterisco indica que se deve implementar a função da qual ele decora, aspas simples indicam que a função não deve ser mexida (também dito no código))

(Nota 2: Algumas funções e funcionalidades das classes iniciadas por HW devem ser implementadas para alguns temas, mais especificamente sobre gerenciamento de memória (HW\_MMU))

## **Alarm**

Implementa a noção de alarme, onde pode se configurar um timer para que dispare em um período de tempo e possa disparar várias vezes repetidamente em intervalos de tempo definidos. Ainda não implementada. Deverá ser implementada para os temas 1.4, 1.5, 1.12, e onde se considere necessário.

Ela possui um Typedef double Microsecond.

*void delay(const Microsecond & time);* deve causar um tempo de pausa na Thread corrente onde ela não deve executar durante o tempo do delay, e deve voltar a executar logo que este tempo se acabe.

## **FileAllocationTable**

Implementa a noção de um sistema de Arquivos FAT. Possui tres classes dentro do arquivo

### **Abstr\_FileAllocationTable:**

**-FileAttributes:** Implementa a noção base dos atributos de um arquivo

*FileAttributes();* construtor dos atributos de arquivo.

*setFilename(char\* filename);* dá ao arquivo um novo nome; Possui get com retorno do mesmo tipo que a entrada\*

*setSize(unsigned int size);* dá ao arquivo um novo tamanho; Possui get\*

*setType(Filetype type);* dá ao arquivo um novo tipo baseado no enum logo acima destas funcoes; Possui get.\*

### **-FileAllocationEntry**

Implementa a noção de uma entrada na tabela de alocação de arquivos.

*FilleAllocationEntry(fileIdentifier inode, HW\_HardDisk::blockNumber block, FileAttributes attributes);* Construtor da entrada da tabela de alocação\*

*fileIdentifier getNode();* retorna um nodo da tabela de entradas, o próprio arquivo.\*

*HW\_HardDisk::blockNumber getBlock();* retorna o primeiro bloco onde este arquivo se encontra dentro do HD.\*

*FileAttributes getAttributes()*: retorna os atributos do arquivo que se encontra nesta entrada da tabela.\*

--Estas tres funcoes possuem set com entrada do mesmo tipo do retorno delas.

## **-FileAllocationTable**

Implementa o sistema de arquivo FAT em si.

*FileAllocationTable(HardDisk HD)*: construtor da tabela de alocação de arquivos\*

*void addFileEntry(FileAllocationEntry fatEntry)*: deve adicionar uma nova entrada na tabela de arquivos\*

*void removeFileEntryByNode(FileAllocationEntry::fileIdentifier inode)*: deve remover uma entrada da tabela de arquivos passando como entrada o arquivo(Nodo)\*

*void removeFileEntryByRank(unsigned int rank)*: deve remover uma entrada da tabela dada a posicao na tabela \*

*FileAllocationEntry getFileEntryByNode(FileAllocationEntry::fileIdentifier inode)* deve retornar a entrada da tabela dado o arquivo em si (Nodo)\*

*FileAllocationEntry getFileEntryByRank(unsigned int rank)*: deve retornar a entrada da tabela dada a posicao em que se encontra na tabela \*

*void setFileEntry(unsigned int rank, FileAllocationEntry fatEntry)*: deve modificar a entrada em rank para a nova entrada\*

## **FileAllocator**

Classe abstrata que deve implementar o algoritmo de alocação de arquivos, por exemplo indexado, contínuo, etc.

*FileAllocator(HardDisk\* disk, FileAllocationTable\* fat)*: Construtor da classe\*

*void CreateFile (const unsigned char\* path)*: deve criar um novo arquivo e aloca-lo no caminho dentro da tabela de alocação e no HD \*;

*void removeFile(const unsigned char\* path)*: deve deletar um arquivo da fat e do HD\*

*FileAllocationEntry::fileIdentifier openFile(const unsigned char\* path)*: deve cuidar de abrir o arquivo, preparando-o para ser acessado. Deve checar permissões antes de abrí-lo. Deve retornar um identificador para o arquivo\*

*void closeFile( const FileAllocationEntry::fileIdentifier file)*: Deve cuidar de fechar o arquivo, terminando todas as operacoes feitas nele e gravando-o no HD caso tenha sido alterado.

*unsigned int readFile(const FileAllocationEntry::fileIdentifier file)*: deve ler do arquivo\*

*unsigned int writeFile(const FileAllocationEntry::fileIdentifier file)*: deve escrever no arquivo\*

*void seek(const unsigned long numByte)*:

## **FileSystem**

Deve implementar a noção de sistema de arquivos, como criar arquivo, procurar arquivo, deletar arquivo, hierarquia de arquivos etc. Ainda não implementada. Deve ser implementada para os temas 1.10, e possivelmente 1.11.

## **MemoryChunk**

Implementa a noção de um pedaço de memória, onde este pode ser uma partição ou uma página, que deve ser lido, escrito e endereçado. Semi-implementada. Deve ser completada para os temas 1.1, 1.2, e 1.3.

*MemoryChunk(unsigned int size):* Construtor da classe, deve setar todas as variáveis como o tamanho do espaço de memória, suas permissões, conteúdo, estado de alocação etc. \*

*virtual ~MemoryChunk():* Destrutor da classe \*

*unsigned int getSize() const:* Retorna o tamanho deste espaço de memória.

*bool isExecutable / isWritable / isReadable():* retorna cada uma das permissões do respectivo espaço de memória.

## MemoryManager

Implementa o gerenciador de memória, é ele quem coordena alocação e desalocação de memória, endereçamento, tradução de endereços lógicos para físicos etc. Semi-implementada. Deve ser completada para os temas 1.1, 1.2, 1.3 e 1.8.

```
enum MemoryAllocationAlgorithm {FirstFit, NextFit, BestFit, WorstFit};
```

*MemoryManager(MemoryAllocationAlgorithm algorithm):* Construtor da classe, a entrada se refere ao método de alocação a ser utilizado, escolhido aleatoriamente pelo simulador ou dado como entrada.

*MemoryChunk\* allocateMemory(unsigned int size):* aqui deve ser implementado um jeito de alocar um espaço de memória, de tamanho *size* utilizando o método de alocação escolhido para encontrar um espaço livre, depois criando uma nova partição para àquele espaço, e retornado um ponteiro para o chunk criado. \*

*void deallocateMemory(MemoryChunk\* chunk):* aqui deve ser implementado um jeito de desalocar um espaço de memória, dado um ponteiro para um chunk especificado. \*

*void showMemory():* aqui deve ser implementado a impressão de como está organizada a memória, no console, no formato especificado dentro da função no código. \*

*unsigned int getNumMemoryChunks():* deve ser implementada para retornar o número de chunks. \*

*MemoryChunk\* getMemoryChunk(unsigned int index):* deve ser implementada para retornar o chunk na posição especificada por index. \*

## Process

Implementa a noção de um processo, que pode rodar aplicações e possuir threads, ser escalonado e etc. Semi-implementado. Deve ser implementada para os temas 1.4, e 1.5.

*Process(unsigned int parentId):* construtor que deve setar as variáveis, dando ao processo criado um ID do parente que a criou e todas as outras que são importantes. \*

*unsigned int getParentID():* retorna o ID do processo pai

*unsigned int getID():* retorna o ID do processo

*Entity\* getEntity():* retorna a entidade do simulador'

*static unsigned int getNewId():* dá ao processo atual um novo ID e o retorna.

*static std::list<Process\*> getProcessesList():* retorna a lista de processos do sistema.

*static Process\* exec():* esta função é muito importante pois processos chegarão ao sistema quando for invocada esta função. A implementação dessa chamada de sistema deve criar um **Process** (assuma que o pai de todos os processos tem id=0), alocar memória para ele invocando a chamada de sistema **Partition\* MemoryManager::allocateMemory(unsigned int size)**, inicializar os novos atributos que porventura você tenha criado, colocá-lo na lista de processos no sistema e criar uma thread (correspondente a "main") para esse processo invocando a chamada **static Thread\* Thread::thread\_create(Process\* parent)**. O método retorna o processo criado. \*

*static void exit(int status = 0):* Processos serão finalizados quando for invocada esta chamada de sistema. A implementação dessa chamada deve desalocar a memória do processo que extá executando invocando **void MemoryManager::deallocateMemory(Partition\* partition)**, excluir todas as threads desse processo, excluir o processo (destruir o objeto **Process**), invocar o escalonador para escolher outra thread, invocando **Thread\* Scheduler::choose()** e então o despachador para iniciar a execução da thread escolhida pelo escalonador, invocando **void Thread::dispatch(Thread\* previous, Thread\* next)**. \*

## Scheduler

Implementa a noção do escalonador. Pode servir para a criação dos escalonadores de Processos, e de Disco.

*Thread\* choose():* deve ser implementado para escolher uma nova thread para rodar. Deve retornar um ponteiro para uma nova Thread. \*

*Thread\* choosen():* deve retornar a última thread escolhida pela função *choose()*. \*

*void reschedule():* Esse método precisa varrer a fila de prontos *\_readyQueue*, ajustando os atributos necessários das threads na fila, possivelmente tendo que removê-las da fila e depois reinserí-las para que haja reordenamento. \*

*void insert(Thread\* thread):* deve inserir uma thread na fila de prontos *\_readyQueue*.

*Thread\* remove(Thread\* thread):* deve remover a thread recebida como entrada da fila de prontos *\_readyQueue*.

## Thread

Implementa a noção de threads, que são executadas pelos processos.

*static void sleep (Queue<Thread\*> \* q):* deve bloquear a thread em execução, tirando-a da fila de prontos, lembre-se que bloqueio não é o mesmo que espera ocupada. \*

*static void wakeup(Queue<Thread\*>\* q):* deve desbloquear a primeira thread bloqueada, recolocando-a na fila de prontos.\*

*static Thread\* thread\_create(Process parent):* Threads criarão novas threads quando for invocada esta chamada, que precisa ser implementada. A implementação dessa chamada deve criar um objeto **Thread**, inicializar os novos atributos que porventura você tenha criado, colocá-la na lista de threads no sistema, e se o escalocador for preemptivo então deve chamar o escalonador para escolher outra thread, invocando **Thread\* Scheduler::choose()** e então o despachador para iniciar a execução da thread escolhida pelo escalonador, invocando **static void Thread::dispatch(Thread\* previous, Thread\* next)**. O método retorna a thread criada.\*

*static std::list<Thread\*>\* getThreadsList():* retorna a lista de threads existentes.

*int join():* esta função deve verificar se a thread que a chama está no estado FINISHING, se não, ela deve trocar o estado da Thread *\_running* para *waiting* e retirá-la da fila de prontos, e colocá-la na fila de espera pela thread que chamou join, depois escalonar para que se rode uma nova thread. Para

melhor ilustrar essa chamada, se o código da thread T1 tiver a chamada T2->join(), então T1 é quem está executando (running) e T2 é quem foi invocada (this), e é T1 que deve ser bloqueada esperando por T2 (se T2 não estiver FINISHING).\*

*Process\* getProcess()*: retorna o processo pai da thread

*int getPriority() const*: retorna a prioridade da thread

*Protected*:

*static Thread\* thread\_create(Process\* parent)*: Threads criarão novas threads quando for invocada esta chamada que precisa ser implementada. A implementação dessa chamada deve criar um objeto **Thread**, inicializar os novos atributos que porventura você tenha criado, colocá-la na lista de threads no sistema, e se o escalador for preemptivo então deve chamar o escalonador para escolher outra thread, invocando **Thread\* Scheduler::choose()** e então o despachador para iniciar a execução da thread escolhida pelo escalonador, invocando **static void Thread::dispatch(Thread\* previous, Thread\* next)**. O método retorna a thread criada. \*

*static void exit(int status = 0)*: Threads serão finalizadas quando for invocada esta chamada, que precisa ser implementada. A implementação dessa chamada deve colocar a thread que está executando no estado FINISHING, verificar se há alguma thread na lista de threads bloqueadas esperando por essa thread. Se houver, todas as threads na lista devem ser colocadas no estado READY e colocadas na fila de threads prontas para executar. Em qualquer caso, deve-se ainda chamar o escalonador para escolher outra thread, invocando **Thread\* Scheduler::choose()** e então o despachador para iniciar a execução da thread escolhida pelo escalonador, invocando **static void Thread::dispatch(Thread\* previous, Thread\* next)**.\*

*static void yield()*: Threads podem decidir deixar a CPU invocando esta chamada, que precisa ser implementada. A implementação dessa chamada deve colocar a thread que está executando no estado READY, incluí-la na fila de threads prontas, chamar o escalonador para escolher outra thread, invocando **Thread\* Scheduler::choose()** e então o despachador para iniciar a execução da thread escolhida pelo escalonador, invocando **static void Thread::dispatch(Thread\* previous, Thread\* next)**. \*

*static Thread\* running()*: retorna um ponteiro para a thread que está rodando no momento.

*static void dispatch(Thread\* previous, Thread\* next)*: Threads são despachadas, ou seja, têm seus contextos trocados, quando se invoca esta chamada que já está implementada. A implementação desse método deve inicialmente verificar se a próxima thread (**next**) é nula ou não. Se for, nada precisa ser feito (isso só ocorre quando a fila de prontos é vazia e não há thread para ser escalonada). Se não for, então o atributo *\_running* deve ser atualizado e a thread a ser executada precisa ser colocada no estado RUNNING e retirada da fila de prontos. Então deve ser verificado se a thread anterior (**previous**) é diferente de nula e também se é diferente da próxima thread. Se não for, então basta restaurar o contexto da próxima thread, invocando **static void CPU::restore\_context(Thread\* next)**. Se for, então é preciso verificar se a thread anterior estava no estado RUNNING e caso sim, então a thread anterior deve passar para o estado READY e ser colocada na fila de threads prontas. Após esse teste deve-se fazer a troca de contexto entre as threads, invocando o método **static void CPU::switch\_context(Thread\* previous, Thread\* next)**

-As classes Mediadoras (Mediator) funcionam como as controladoras para os dispositivos de hardware, é através delas que os alunos poderão acessar os módulos de hardware.

CPU

Implementa a controladora da CPU, e permite a troca de contextos, e carregamento de contextos existentes na pilha.

*static void switch\_context(Thread\* previous, Thread\* next):* faz a troca de contexto entre as threads *previous* e *next*.

*static void restore\_context(Thread\* next):* restaura o contexto da thread *next*.

*static void getInstance():* retorna uma instância do mediador da CPU.

*static void setInstance(unsigned int \_instance):* atribui uma instancia de simulador à variavel estatica *\_instance*.

## HD

Permite o controle do hardware de Disco Rígido, nela devem ser implementadas funções para gravar e ler do Disco, além de mover blocos de memória, e etc. Deve ser implementada para os Temas 1.9.

A interface pública do hardware do HD permite que se leia e escreva no disco, de do arquivo HW\_HardDisk.c, existe uma classe chamada DiskAccesRequest, que é o tipo para a entrada para as funções de leitura e escrita no HD, ela possui atributos como o setor, a track e a prioridade da requisição.

## MMU

Permite o controle do hardware da MMU, que deve cuidar de reduzir o overhead para traduzir endereços lógicos para físicos do processador, e gerenciar as tabelas de alocação. Deve ser implementado para os temas 1.1, 1.2, e 1.3 e 1.8;

Existe mais de uma MMU em hardware. Uma é a "default" para os trabalhos que não se utilizam dela, e mais duas, uma para alocação contínua e outra para paginada. O hardware em si já está em implementado porém o software, ou seja o mediador deve ser implementado para que funcione adequadamente com o hardware.

## Timer

Permite o controle do hardware do Timer, configurando as interrupções lançadas por ele e etc. Possui uma função denominada interrupt handler, que deve ser implementada para os Temas 1.4, 1.5, 1.6, ela deve tratar das interrupções de timer no sistema.

Uma última classe importante é **Simulator**, que possui duas funções interessantes, porém não deve ser mexida:

*Simulator\* getInstance()*, uma função estática que retorna um singleton de simulador, para que se tenha acesso a suas funções,'

*double getTnow()*, que retorna o tempo atual da simulação;'

---

**OBS: O código foi disponibilizado via GitHub no repositório <https://github.com/rlcancian/20162-INE5412-SC-Theme1-StartPoint>, permitindo que os alunos implementem em seu próprio ambiente de execução, e atualizem (importem) o código periodicamente para o VPL para verificação do andamento do Sistema Computacional.**

## NAVIGATION



### Dashboard

- Site home

Moodle UFSC

Current course

INE5412-04208A (20162)

Participants

Badges

8 agosto - 14 agosto

15 agosto - 21 agosto

22 agosto - 28 agosto

29 agosto - 4 setembro

5 setembro - 11 setembro

12 setembro - 18 setembro


19 setembro - 25 setembro

26 setembro - 2 outubro

3 outubro - 9 outubro

10 outubro - 16 outubro

17 outubro - 23 outubro

 **SC Project VPL - Theme 1: Simulation of Operating ...**

- Description

- Submission view

My courses

## ADMINISTRATION



Course administration

---

You are logged in as Cesar Smaniotto Junior (12100740) (Log out)  
INE5412-04208A (20162)