Pontifícia Universidade do Rio Grande do Sul
Escola Politécnica - 2020/1

GII de Sistemas Operacionais

Aluno: Gabriel Fanto Stundner
Data de Entrega: 16/07/2020
Curso: Engenharia de Software

Porto Alegre - Rio Grande do Sul
Brasil

Resolução da Questão 1

a) Memória de Partição Fixa

Explicando de uma forma simples, uma memória de partição fixa é uma memória separada em Partições, que são blocos de posições para salvar processos e suas informações, onde cada Partição nesse formato possui o mesmo tamanho.

Na minha explicação colocarei às Partições como Posições dentro de um ArrayList, que é uma estrutura que se pode guardar valores de Diferente tipos, onde podemos acessar ele com Métodos em Java.

Para se saber o tamanho de cada Partição é necessário fazer um cálculo simples, onde pegamos o tamanho máximo de Memória disponível e o dividimos pelo número de Partições desejadas.

No programa que construí separei a memória de 32 posições em 4 Partições, que ficaram com 8 posições em cada Partição:

32/4 = 8 onde,

A Posição 0 fica a Lista de Processos do Sistema

Partição 1 = Posições 1 a 8

Partição 2 = Posições 9 a 16

Partição 3 = Posições 17 a 24

Partição 4 = Posições 25 a 32

Dessa forma, eu elaborei um algoritmo simples para mostrar como funciona a Memória de Forma Programada em Java que apresenta essa memória de Partições Fixas:

```java
class Memory {
    private int size = 32;
    private List<Object> memory;
    public Memory(){
        memory = new ArrayList<>();
        for(int i = 0 ; i < size ; i++){memory.add(i,null);}
    }

    public void teste(){
        for(int i = 0 ; i < memory.size() ; i++){
            System.out.println("Position " + i + " : " + memory.get(i));
        }
    }
    public List<Object> getMemory(){return memory;}
    public int getMemSize(){return size;}

    // Só para definir uma memoria particionada
    public int getPartition(int numberProcess){
        switch(numberProcess){
            case 1: return 8;
            case 2: return 16;
            case 3: return 24;
            case 4: return 32;
        }
        return -1;
    }
}
```

Como mostrado no Código acima, temos às Seguintes Variáveis:

- **size:** é o tamanho máximo da memória de teste
- **memory:** é um ArrayList que serve para ser a memória dividida em Posições

Possui os Seguintes Métodos:

- **Memory():** Método Construtor que inicializa uma Memória de 32 Posições onde todas elas possuem *Null,* que significa que todas às posições estão vazias esperando receber algum dado nelas
- **teste():** Método só para poder ver a memória pelo terminal
- **getMemory():** Este é o Método que vai ser chamado pelas outras classes, onde podemos acessar a memória e alterá-la como quisermos
- **getMemSize():** Este Método retorna o tamanho da memória
- **getPartition(int numberProcess):** Este método é um exemplo básico de partição, onde pegamos o id do processo e o colocamos em uma das 4 Partições possíveis já criadas.

Está Classe é uma demonstração bem xula de elaboração de uma Memória Particionada de tamanho fixo, onde ela não leva em consideração no caso de outros Processos além dos quatro criados para exemplo, em uma Memória Particionada Fixa real é verificado se uma Partição está livre para poder ser escolhida a um Processo específico.

Agora que temos a Memória de Partições Fixas para podermos usar, temos que elaborar às classes necessárias para o Tipo de Process Control Block desse formato de Memória.

No Process Control Block, toda vez que é trocado um Processo ele deve salvar todo o Contexto que estava sendo usado para o Processo específico, por isso, elaborei uma Classe *ContextData* que salva às posições dos Registradores na memória dentro de Variáveis, como mostrado abaixo:

```
class ContextData{
    int r1,r2,r3,r4,r5,r6,r7,r8;
    int idProcess;
    public ContextData(int idProcess){
        this.idProcess = idProcess;
    }
    public int getRegister(String register){
        switch (register) {
            case "r1": return r1;case "r2": return r2;
            case "r3": return r3;case "r4": return r4;
            case "r5": return r5;case "r6": return r6;
            case "r7": return r7;case "r8": return r8;
        }
        return -1;
    }
    public void save(int r1,int r2,int r3, int r4, int r5, int r6, int r7, int r8){
        this.r1 = r1;this.r2 = r2;this.r3 = r3;this.r4 = r4;
        this.r5 = r5;this.r6 = r6;this.r7 = r7;this.r8 = r8;
    }
}
```

Está classe é um Simples Objeto que vai salvar às posições de memória dos Registradores dentro dos seus Processos, onde possui às seguintes variáveis:

- **r1,r2,r3,r4,r5,r6,r7,r8:** são inteiros que no caso vão ser às posições do nosso ArrayList da Memória
- **idProcess:** vai servir para sabermos qual Processo possui os Dados

E possui os seguintes Métodos:

- **ContexData(int idProcess):** inicializador do ContextData para podermos usar essa Classe no Processo
- **getRegister(String register):** retorna o valor do registrador desejado entrado como String no Método
- **save(int r1,...,int r8):** esse é o Método que vai salvar às posições dos Registradores no contexto e que irão ficar guardados no Processo onde essa Classe foi inicializada.

Em uma Classe de ContextData original, todos os dados e informações que estão rodando no momento é salvo, até mesmo o Programa se for preciso, para que quando for trocado o Processo é retirado da Memória ele tenha tudo que precisa para ser rodado de novo.

Agora finalmente temos nossa Classe que é o Processo em Si, onde temos o seguinte código:

```
class Process{
    private int id; // Identificador Unico do Processo
    private String state; // Estado do Processo
    private int priority; // Prioridade do Processo
    int baseRegister; // Primeira Posição do Processo
    int boundsRegister; // Ultima Posição do Processo
    int pc; //Valor do Program Counter

    public Process(int id, String state){
        this.id = id;this.state = state;
    }
    // Salvar o Contexto
    ContextData context = new ContextData(id);

    public int getId(){return id;}
    public int getPc(){return pc;}
    public String getState(){return state;}
    public int getPriority(){return priority;}
    public int getBaseRegister(){return baseRegister;}
    public void setBaseRegister(int value){baseRegister = value;}
    public int getBoundsRegister(){return boundsRegister;}
    public void setBoundsRegister(int value){boundsRegister = value;}
    public void setPc(int pc){this.pc = pc;}
    public void setState(String state){this.state = state;}
    public void setPriority(int priority){this.priority = priority;}

    // Salvando o Contexto Atual
    public void saveContextData(int r1,int r2, int r3, int r4, int r5, int r6, int r7, int r8){
        context.save(r1, r2, r3, r4, r5, r6, r7, r8);
    }
    public int getRegister(String register){
        return context.getRegister(register);
    }

}
```

Como mostrado no Código acima, temos às seguintes variáveis:

- **id:** esse é o identificador único do Processo.
- **state:** o state serve para sabermos como está o Status do Processo no sistema, onde o mais simples são 3 status diferentes: NEW,RUNNING,FINISH, mas pode ter outros.
- **priority:** serve para sabermos a prioridade do Processo na Fila de Processos do Process Control Block, onde é a posição do Processo na Fila.
- **baseRegister:** esta variável salva qual a primeira posição da Partição de Memória onde esse Processo se encontra.
- **boundsRegister:** esta variável salva qual a última posição possível da Partição de Memória onde esse Processo se encontra.
- **pc:** este é o Program Counter, que vai lendo às posições de memória e fazendo às Funções assembly que se encontram nessas Posições.
- **context:** esta é a inicialização do ContextData dentro do Processo.

Os métodos do Processo que podem ser utilizados por outras Classes são:

- **getId():** Método que retorna o id do Processo.
- **getState(),setState(String state):** Métodos de gerência do Status.
- **getPriority(),setPriority(int priority):** Métodos de gerência da Prioridade.
- **getPc(),setPc(int pc):** Métodos de gerência do Program Counter.
- **getBaseRegister(),setBaseRegister(int value):** Métodos de gerência do baseRegister.
- **getBoundsRegister(),setBoundsRegister(int value):** Métodos de gerência do boundsRegister.
- **saveContextData(int r1,...,int r8):** este Método chama o Método dentro de ContextData para salvar às posições de registradores vindos do Process Control Block.
- **getRegister(String register):** retorna o valor da posição de memória salva em um registrador, para podermos verificar se está salvando direito às posições.

Agora, depois dessas classes, temos como construir nosso Process Control Block para esse tipo de Memória:

```java
class PCB{
    Queue<Process> queue;
    Memory mem;
    int r1,r2,r3,r4,r5,r6,r7,r8;

    public PCB(Memory mem){
        queue = new LinkedList<>();
        this.mem = mem;
        mem.getMemory().add(0,queue);
    }

    public void createProcess(int id){
        Process process = new Process(id,"NEW");
        process.setPriority(queue.size());
        process.setBoundsRegister(mem.getPartition(id));
        process.setBaseRegister(mem.getPartition(id)-7); // 7 porque são 8 posições por part.
        process.setPc(process.getBaseRegister());
        insertOnMemory(process.getBaseRegister(), process.getBoundsRegister(), id);
        queue.add(process);
    }

    public void insertOnMemory(int registerBase, int boundsRegister, int id){
        int j = 0 ;
        for(int i = registerBase ; i <= boundsRegister ; i++){
            String value = "Lendo Linha " + j + " Processo " + id;
            j++;
            mem.getMemory().set(i,value);
        }
    }
```

```java
    // Leitura de uma Linha de cada Processo
    private int adder(int relative, int baseReg){
        return relative + baseReg;
    }
    private boolean comparator(int value, int boundsReg){
        if(value <= boundsReg){
            return true;
        }
        return false;
    }

    public void lerLinha(int valor){
        Process process = queue.poll();
        int value = adder(valor, process.baseRegister);
        if(comparator(value, process.boundsRegister)){
            System.out.println("Posição " + value + " | Processo " + process.getId() + " : " + mem.getMemory().get(value) + "\n");
            process.saveContextData(value,-1, -1, -1, -1, -1, -1, -1);
        }else{
            System.out.println("Posição " + value + " Não existe no Processo " + process.getId());
        }
        queue.add(process);
    }
```

```java
    // TESTES
    public int queueSize(){return queue.size();}
    public void testeFila(){
        while(queue.size() > 0){System.out.println("Processo " + queue.poll().getId());}
    }
    public void testeRegisters(){
        Process aux = queue.poll();
        System.out.println("Processo " + aux.getId());
        System.out.println("Base Register: " + aux.getBaseRegister());
        System.out.println("Bounds Register: " + aux.getBoundsRegister());
        System.out.println("Prioridade: " + aux.getPriority());
    }
    public void testeContextData(){
        Process aux = queue.poll();
        System.out.println("Processo: " + aux.getId());
        System.out.println("ContextData: " + aux.getRegister("r1") + "\n");
    }

}
```

Agora explicando melhor, o Process Control Block deve poder:

I. Criar um Processo
II. Adicionar um Processo em uma Partição de Memória
III. Gerenciar os Processos que estão em uma Fila de execução
IV. Modificar a Posição Relativa(vindo do Programa) em uma Posição de Memória real presente dentro da Partição do Processo
V. Poder parar e continuar um Processo depois, salvando o Contexto do Processo

O programa que eu fiz de exemplo possui às seguintes variáveis:

- **queue:** esta é a Fila de Execução dos Processo dentro do PCB
- **mem:** esta é a Memória que foi inicializada fora do PCB
- **r1,...,r8**: estes são os Registradores que vão ser usados na execução dos Processos

O Métodos do Process Control Block fazem o seguinte:

- **PCB(Memory mem):** este é o construtor do Process Control Block, que recebe a memória como Parâmetro para ser usada pelos Processos, além disso salva a Fila de Execução na primeira posição da Memória para usarmos ela e fazer com que todas às Partições tenham tamanhos exatos(por exemplo sem isso a Primeira Partição seria da Posição 0 até a Posição 7 da Memória)
- **createProcess(int id):** este Método é o Método que criamos os Processos, onde no meu exemplo não estou lidando com leitura de funções assembly e sim em leitura dentro de uma Partição, o Processo é iniciado com:
    - O status do Processo é iniciado como **NEW**
    - A prioridade do Processo é o tamanho da Fila,por exemplo se não tiver valores dentro da Fila, a Prioridade do Processo é 0.
    - Definimos o **baseRegister** e o **BoundsRegister** a partir da Partição escolhida, como por exemplo se for o Processo 1, o meu programa vai definir a Partição 1 como a Partição desse processo.
    - O valor do Program Counter vai ser o valor da Primeira posição da Partição, por que é onde ele vai começar a leitura
    - Ele vai iniciar todas às posições da memória de cada Processo com valores criados no próximo Método que irei explicar, chamado de **insertOnMemory.**
    - E por fim, ele vai adicionar o processo dentro da Fila de Processos
- **insertOnMemory(int registerBase, int boundsRegister, int id):** Este Método foi criado simplesmente no intuito de termos o que ler da memória, porque eu não criei e nem precisa para o exemplo a leitura de um arquivo, onde eu simplesmente adiciono em cada linha da Partição de cada Processo informações como a linha lida e qual o Processo que esta linha está dentro.

Os Próximos Métodos servem para pegarmos o valor de uma Posição externa e mudarmos ela para uma Posição real dentro de cada Processo, porque já que estamos mexendo com uma Partição que possui posições específicas não é qualquer posição que ela aceita, como por exemplo:

Digamos que o programa pediu a posição 5 da memória mas estamos trabalhando com uma Partição que inicia na Posição 9 até a 16, dessa forma pegamos o valor de

entrada e somamos com a primeira posição da Partição e depois verificamos se esse valor não passa do valor limite da Partição que estamos trabalhando, como abaixo:

5 + 9 = 14 < 16? verdadeiro, portanto acessamos a posição 14 dentro da Partição.

Para isso foi criado os Seguintes Métodos:

- **adder(int relative, int baseReg):** calculamos um valor relativo de entrada e somamos com o valor do RegisterBase da Partição.
- **comparator(int value, int boundsReg):** esse Método verifica se o valor de entrada é menor ou igual ao limite da Partição e retorna um Booleano.
- **lerLinha(int valor):** este é o Método principal de leitura da memória, onde ele utiliza os Métodos **adder** e **comparator** para podermos ler uma Linha de um Processo que está sendo retirado da Fila de Processos, onde além de ler uma Linha e mostrar na Tela do Terminal, ele salva a posição lida no Registrador 1 do Processo(como exemplo) e adiciona novamente para a Fila, fazendo com que quando esse processo for chamado de novo ele vai ler outro Processo toda vez.

Existe uns últimos Métodos que servem para testar como às coisas estão acontecendo na Leitura dos Processos, como:

- **queueSize():** somente para vermos o tamanho da Fila
- **testeFila():** somente para vermos os processos dentro da Fila
- **testeRegisters():** verificar os dados do Processo dentro da Fila, sendo o Processo aleatório
- **testeContextData():** serve para verificarmos os dados salvos no ContextData do Processo.

Não elaborei de forma Concorrente porque não era preciso, onde deixo bem claro como funciona sem precisar colocar Threads.

b) Memória de Partição Dinâmica

Há algumas grandes diferenças para a Partição Fixa, como que a Memória é dividida em Frames de vez de somente serem posições únicas como Programado na Partição fixa, onde em cada Frame é adicionado uma Parte do Processo, onde vão sendo adicionada se tiver espaço dentro da memória, não precisando deixar todos os Frames unidos em um único ponto.

Primeiramente, a mudança feita na Memória, deixando ela Dinâmica:

```java
class Memory {
    private int size = 16;
    private List<Object> memory;

    public Memory() {
        memory = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            memory.add(i, null);
        }
    }

    public void teste() {
        for (int i = 0; i < memory.size(); i++) {
            System.out.println("Position " + i + " : " + memory.get(i));
        }
        System.out.println("\n");
    }

    public List<Object> getMemory() {
        return memory;
    }

    public int getMemSize() {
        return size+1;
    }

}
```

Neste exemplo deixei a Memória menor, para não ficar muito grande os Resultados.
As Variáveis não mudaram muito da Primeira parte, sendo somente que o tamanho da Memória agora são de 16 Posições, que em nossa memória Dinâmica são chamadas de **Frames.**

O ContextData não foi modificado porque mexer com os Registradores fica da mesma forma como na memória de Partições fixas.

Agora vamos para às Modificações reais:

Temos abaixo o novo Processo do Sistema:

```java
class Process {
    private int id;
    private String state;
    private int priority;
    private int pageNumbers = 4;
    private Map<Integer, Integer> pagination = new HashMap<>();
    int pc;
    private int baseRegister;
    private int boundsRegister;

    public Process(int id, String state){
        this.id = id;this.state = state;
    }
    ContextData context = new ContextData(id);

    public int getId(){return id;}
    public int getPc(){return pc;}
    public String getState(){return state;}
    public int getPriority(){return priority;}
    public void setPc(int pc){this.pc = pc;}
    public void setState(String state){this.state = state;}
    public void setPriority(int priority){this.priority = priority;}

    // Paginação
    public Map<Integer,Integer> getPagination(){return pagination;}
    public void setPagination(Map<Integer,Integer> pages){this.pagination = pages;}
    public int getPageNumbers(){return pageNumbers;}

    // Registers
    public int getBaseRegister(){return baseRegister;}
    public int getBoundsRegister(){return boundsRegister;}
    public void setBaseRegister(int br){baseRegister = br;}
    public void setBoundsRegister(int br){boundsRegister = br;}

    // ContextData
    public void saveContextData(int r1,int r2, int r3, int r4, int r5, int r6, int r7, int r8){
        context.save(r1, r2, r3, r4, r5, r6, r7, r8);
    }
    public int getRegister(String register){
        return context.getRegister(register);
    }

}
```

Em meu exemplo não apresentei a utilização do baseRegister e do boundsRegister, mas no caso se for rodar um programa, toda vez que acessamos um Frame, temos que pegar a primeira posição do vetor interno do Frame e o tamanho do Vetor.

A maior modificação agora é que temos o Método chamado **getPagination(),setPagination(Map<Integer,Integer> pages)** e a Variável **pagination.** Essas informações são para salvar às Páginas e os Frames específicos, para que quando o Process Control Block for rodar o Processo ele saiba onde ficam às informações daquele Processo em específico, onde toda a informação fica em um Hashmap onde:

(página,frame) = onde a página é uma parte do Processo e o Frame onde ele se encontra na Memória.

Agora temos o nosso Process Control Block:

```java
class PCB{
    Queue<Process> queue;
    Memory mem;
    int r1,r2,r3,r4,r5,r6,r7,r8;
    Map<Integer,Integer> pagination = new HashMap<>();

    public PCB(Memory mem){
        queue = new LinkedList<>();
        this.mem = mem;
        mem.getMemory().add(0,queue);
    }

    public void createProcess(int id){
        Process process = new Process(id,"NEW");
        process.setPriority(queue.size());

        // Verificando Frames Livres
        ArrayList<Integer> list = new ArrayList<>();
        for(int i = 0 ; i < mem.getMemSize() ; i++){
            if(mem.getMemory().get(i) == null){
                list.add(i);
            }
        }
        // Pegando os primeiros Frames para o Processo
        Map<Integer, Integer> auxMap = new HashMap<>();
        for(int i = 0 ; i < process.getPageNumbers() ; i++){
            auxMap.put(i, list.get(i)); //(página,frame)
        }

        // Adicionando os valores de exemplo
        for(Integer page : auxMap.keySet()){
            Integer frame = auxMap.get(page);
            insertOnMemory(page, frame, process.getId());
        }

        process.setPagination(auxMap);
        queue.add(process);
    }
```

```java
    public void insertOnMemory(int page, int frame, int id){
        String[] values = new String[4];
        for(int i = 0 ; i < values.length ; i++){
            values[i] = "Página " + page + " Frame " + frame + " Processo " + id + " Linha " + i;
        }
        mem.getMemory().set(frame,values);
    }

    // Leitura de uma Página
    private int frameLocation(int relative, int frameSize){
        return relative / frameSize;
    }

    private int offset(int value, int frameSize){
       return value - frameSize;
    }

    public void lerLinha(int valor){
        Process process = queue.poll();
        int discoverFrame = frameLocation(valor, process.getPageNumbers());
        int offset = offset(valor, process.getPageNumbers()); // posição dentro do frame
        int accessPosition = process.getPagination().get(discoverFrame); //frame que iremos acessar
        String[] vetorAux = (String[]) mem.getMemory().get(accessPosition);
        System.out.println("Valor na Linha " + offset + " da Posição " + accessPosition + " : " + vetorAux[offset]);
        process.saveContextData(offset, r2, r3, r4, r5, r6, r7, r8);
        queue.add(process);
    }
```

Eu decidi colocar eles em Ordem para facilitar na hora de testar exemplos, mas não é prioridade, porque podemos salvar às Páginas em qualquer Frame da Memória.

Quando criamos um Processo no Método **createProcess(int id),** já verificamos se tem Frames Livres na Memória e às salvamos em um ArrayList para facilitar na hora de escolher os 4 Frames que iremos salvar.

Eu defini que o número de Frames por Processo são 4 para poder ficar melhor de testar, onde são escolhidos em ordem.

Eu adicionei como exemplo a localização da Página,Frame e a Linha sendo lida, como abaixo:

Página x Frame y Processo z Linha p

Essas informações ficam em cada Linha dentro de uma Página dentro de um Frame, onde são salvos em um Vetor, ou seja de forma mais clara:

Frame
|
| -- > Page
|
| -- > Process
|
| -- > Line

Dessa forma, é acessado o vetor dentro da Página e lido a Linha dentro do Vetor

Após definido às Páginas e os Frames no HashMap e criado às Linhas, é salvo o HashMap no Processo.

O processo é adicionado na Fila de Execução.

Agora vem os Novos Método de Transformação e uma Posição Relativa:

- **frameLocation(int relative,int frameSize):** esse Método divide o valor relativo entrado pelo Programa com o tamanho do Frame(no caso o tamanho é 4), o valor resultante vai ser o número da linha da lista de Frames Disponíveis daquele Processo específico que será lido, como no exemplo:
  - Posição relativa 4 = 4 / 4 = 1
  - No HashMap de Frames vai ter na Posição 1 o verdadeiro Frame que vai ser lido

- **offset(int value, int framesize):** Este Método nos ajuda a descobrir qual a Posição dentro da Página que será lido de verdade, onde fazemos um cálculo que subtraímos o valor relativo com o tamanho do Frame:
  - Posição relativa 4 = 4 - 4 = Posição que será lida: 0

- **lerLinha(int valor):** este é o Método Principal, onde ele utiliza os outros dois métodos anteriores. Pegamos o Processo com a Maior prioridade da Fila e calculamos qual vai ser a Posição do Hashmap na Variável **discoverFrame.** descobrimos o valor do offset com a Variável **offset.** A variável **accessPosition** vai receber a Posição do Hashmap onde se encontra o Frame que iremos acessar. Iremos apresentar na Tela o que se encontra no Frame recebido no **accessPosition** e o valor que se encontra guardado na Linha pega pelo **offset.** Após imprimir o valor, iremos salvar no

ContextData(como exemplo no Registrador 1) o valor da posição do **offset** que foi lido da Memória e por fim adicionaremos novamente o Processo para a Fila de processos.

Então finalizando, o Process Control Block recebe do Processo a Lista de Páginas salvas nos Frames e lê dentro da Página o que foi salvo nela.

Resolução da Questão 2

boolean wantp ← false, wantq ← false			
p		**q**	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	while wantq	q3:	while wantp
p4:	wantp ← false	q4:	wantq ← false
p5:	wantp ← true	q5:	wantq ← true
p6:	critical section	q6:	critical section
p7:	wantp ← false	q7:	wantq ← false

Fiz o Teste a Mão e saiu esses Resultados:

A Primeira fase o Programa, começando pelo Processo P, vai iniciar os dois lendo às suas Próprias sessões Não-Críticas e os Estados das Variáveis Globais em False.

Após lido às suas Sessões particulares, às duas Variáveis se tornam true, começando pela wantP(no Processo P) e depois o wantQ(no Processo Q), onde vai escalando de um Processo para o outro.

Os dois Processos iniciam o While, mas somente o do Processo P vai acontecer, devido que eu iniciei o teste pelo Processo P, se fosse iniciado pelo Processo Q o While dele iria ser iniciado.

Como o While do Processo P foi iniciado, ele altera o estado da variável necessário no While de Q para False, fazendo com que o Q saia do While e inicie a sua Sessão Crítica.

Ainda dentro do While do Processo P, é alterado novamente a Variável wantP para True e no Processo Q o estado da Variável wantQ é trocado para False, começando assim um novo ciclo de atuação do Programa, onde agora devido a esse While, os dois Processos acessam a Sessão Crítica de forma individual, auxiliando na resolução da **Exclusão Mútua.**

Processo P	Processo Q	wantP	wantQ
P1: Non-Critical	Q1: Non-Critical	False	False
P2: wantP ← True	Q2: wantQ ← True	True	True
P3: While wantQ	Q3: While wantP	True	True
P4: wantP ← false	Q6: Critical Section	False	True
P5: wantP ← true	Q7: wantQ ← false	True	False

Agora, os dois Processo ficam em um Loop, onde nesse Momento o Processo P pode acessa a Sessão Crítica devido que o wantQ agora é False.

Quando chega na P7 no Processo P, ele vai mudar o estado da Variável wantP para False e no Processo Q ele vai mudar o estado da Variável wantQ para True, fazendo nesse momento com que o Processo Q nunca entre mais dentro do While criado, fazendo ele pular direto para a Sessão Crítica e o Processo P vai reiniciar seu Loop Infinito e ler a Sessão Não Crítica e criando assim um novo Loop dentro do Loop onde cada Processo vai ler a Sessão Crítica em seu Momento.

Abaixo está mostrando duas vezes o Loop de ativação explicado acima.

Processo P	Processo Q	wantP	wantQ
P6: Critical Section	Q1: Non-Critical	True	False
P7: wantP ← False	Q2: wantQ ← True	False	True
P1: Non-Critical	Q6: Critical Section	False	True
P2: wantP ← True	Q7: wantQ ← False	True	False
P6: Critical Section	Q1: Non-Critical	True	False
P7: wantP ← False	Q2: wantQ ← True	False	True
P1: Non-Critical	Q6: Critical Section	False	True
P2: wantP ← True	Q7: wantQ ← False	True	False

Agora respondendo a cada uma das Questões:

a) **Exclusão Mútua**: A exclusão Mútua foi resolvida devido que cada Processo vai ler a Sessão Crítica em seu Momento, evitando que os Dois Processos alterem os dados compartilhados ao mesmo tempo.

b) **Progresso**: Nenhum dos Processos é Parado esperando pelo outro, devido que como eles passam de um Processo para o outro em cada interação eles não se trancam esperando pelo outro, podendo assim serem acionados ao mesmo tempo e lendo diferentes informações em cada acontecimento.

c) **Não Postergação:** Os Processos não precisam esperar muito para que possam acessar a Sessão Crítica, então eles não ficam esperando um terminar a Sessão Crítica para que o outro Processo possa Ler ela também

Resolução da Questão 3

Peguei a ideia do Algoritmo e construí um Código em Java para construir às Funções e poder testar no meu computador, onde o código possui às seguintes estruturas:

- **Classe ListaCriador** = ListaCriador serve para construir Objetos que são às Listas que iremos interagir, essas serão às Listas que vão ser usadas pelo Programa

```java
class ListaCriador extends Thread{
    private ArrayList<Integer> lista;
    private int min,max;

    public ListaCriador(int min,int max){
        this.min = min;
        this.max = max;
        lista = new ArrayList<>();
    }

    public ArrayList<Integer> getLista(){return lista;}


    public void run(){
        for(int i = min ; i < max ; i++){
            lista.add(i);
        }
    }

    // Método de Teste
    public void testeSaida(){
        for(int i = 0 ; i < lista.size() ; i++){
            if(lista.get(i) != null){
                System.out.println("Posição " + i + " : " + lista.get(i));
            }
        }
    }
}
```

Como mostrado no código, essa Classe Possui os Seguintes Métodos:

- **ListaCriador(int min,int max):** este Método é o construtor do Objeto, onde ele vai receber os limites da Lista, para podermos ter uma Lista Inicial com um tamanho específico, podendo ser pequena ou grande
- **getLista():** Este é o Método que será mas utilizado dentro das outras Classes, onde ele vai pegar a Lista Completa para poder interagir com ela nos Método pedidos pelo exercício.
- **run():** Esta classe também é uma Thread, onde os valores adicionados dentro das Listas é um Processo
- **testeSaida():** É um Método simples para poder apresentar os dados no Terminal quando testar

- **Classe ListaBloqueio:** Esta é a Classe que era para ser a Struct listaBlq, é nesta Classe que se encontram todos os Métodos necessários para rodarmos os Processos que interagem com às Listas.

```
class ListaBloqueio extends Thread{
    private Semaphore mutex;
    ListaCriador lista1;
    ListaCriador lista2;

    public ListaBloqueio(ListaCriador lista1, ListaCriador lista2){
        this.mutex = new Semaphore(1);
        this.lista1 = lista1;
        this.lista2 = lista2;
    }

    /**
     * Adiciona o Item na Lista de Itens de entrada
     * @param lista
     * @param item
     */
    public void insere(ListaCriador lista, int item){
        if(lista.getLista().contains(item) == false){
            lista.getLista().add(item);
        }
    }

    /**
     * Retira o elemento descobrindo seu index
     * @param lista
     * @param index
     * @return Integer
     */
    public int retira(ListaCriador lista, int value){
        int position = lista.getLista().indexOf(value);
        lista.getLista().remove(position);
        return value;
    }
```

```java
/**
 * Passa um valor do Final da Lista 1 para a Lista 2
 * @param lista1
 * @param lista2
 * @return boolean
 */
public boolean passa(ListaCriador lista1, ListaCriador lista2){
    int position = lista2.getLista().size()-1;
    if(position >= 0){
        insere(lista1, lista2.getLista().get(position));
        retira(lista2, lista2.getLista().get(position));
        return true;
    }
    return false;
}

/**
 * Método para Rodar em Loop os Métodos
 * @param lista1
 * @param lista2
 */
public void procLista(ListaCriador lista1, ListaCriador lista2){
  int aux = 10;
  while(aux > 0){
    passa(lista1, lista2);
    aux--;
  }

}


public void run(){
    try{
        mutex.acquire();
        procLista(lista1, lista2);
    }catch (InterruptedException e){
        e.printStackTrace();
    }finally{
        mutex.release();
    }
}
```

```java
  // Método de Teste
  public void teste(){
      for(int i = 0 ; i < lista1.getLista().size() ; i++){
          System.out.println("Lista 1: " + " Posicao " + i + " : " + lista1.getLista().get(i));
      }
      System.out.println("\n");
      for(int i = 0 ; i < lista2.getLista().size() ; i++){
          System.out.println("Lista 2: " + " Posicao " + i + " : " + lista2.getLista().get(i));
      }
      System.out.println("\n");
  }

}
```

Está é a Classe Principal, onde os Método Funcionam da Seguinte forma:

- **ListaBloqueio(ListaCriador lista1, ListaCriador lista2):** este Método vai receber duas Listas Externas, onde iremos mexer com elas, porque essas Listas não podem ser criadas dentro dessa Classe senão outros Processos não poderão mexer nelas,além disso é nesse Método construtor que vai ser iniciado o Semáforo.
- **insere(ListaCriador lista,int item):** Este Método verifica se o item existe dentro da Lista que queremos inserir e se não existir iremos adicionar o item na Lista, para que não tenha cópias de um elemento já inserido.
- **retira(ListaCriador lista, int value):** Este Método eu modifiquei ele para que se receba o elemento que queremos retirar da Lista de entrada, porque dessa forma sabemos qual elemento retirar, verificando a posição dele dentro da Lista.
- **passa(ListaCriador lista1,ListaCriador lista2):** Este é o Método que iremos adicionar um item da lista2 dentro da lista1 e depois removendo esse item de dentro da lista2 para não existir dois elementos iguais nas duas listas, retornando um true, mas não utilizei esse retorno no sistema.
- **procLista(ListaCriador lista1, ListaCriador lista2):** Esse Método serve para rodarmos a passagem de um elemento para outra lista em um número específico de vezes, onde eu defini como 10 vezes em um Loop While, onde ele chama o Método **passa.**
- **run():** Este é o Método vindo da Classe Thread que irá utilizar o Semáforo para ser rodado o Método **procLista** devido que mais de um Processo irá utilizar às Listas defini que o Método **procLista** é a Sessão Crítica da Thread, onde o Semáforo **mutex** vai controlar qual Processo irá chamar o Método.
- **teste():** Método para simplesmente podermos testar os valores da Thread

E por fim, a Classe de Teste chamada **Solution,** assim como às da questão 1, onde eu crio duas Listas e início duas Threads diferentes, onde cada uma chama às listas de uma forma diferente, onde cada vez que eu rodo o Programa, os valores das Listas são enviados para Listas Diferentes de formas Diferentes:

```java
public class Solution{
    Run | Debug
    public static void main(String[] args){
        // Listas Criadas
        ListaCriador lista1 = new ListaCriador(0,3);
        ListaCriador lista2 = new ListaCriador(4,7);

        // Adicionando valores como Processo
        lista1.start();
        lista2.start();
        System.out.println("\nLista 1\n");
        lista1.testeSaida();
        System.out.println("\nLista 2\n");
        lista2.testeSaida();

        // Programa
        ListaBloqueio fase1 = new ListaBloqueio(lista1, lista2);
        ListaBloqueio fase2 = new ListaBloqueio(lista2, lista1);
        fase1.start();
        fase2.start();

        System.out.println("\nApós os Processos: \n");

        System.out.println("\nProcesso 1\n");
        fase1.teste();
        System.out.println("\nProcesso 2\n");
        fase2.teste();

        System.out.println("\n As Listas \n");
        System.out.println("\nLista 1\n");
        lista1.testeSaida();
        System.out.println("\nLista 2\n");
        lista2.testeSaida();

    }
}
```

Resolução da Questão 4

1) Quando um arquivo é aberto, precisamos verificar se temos Blocos disponíveis para esse Programa e quem faz isso é o **Volume Control Block,**depois, o Arquivo é adicionado em um **File Control Block** que vai gerenciar o Arquivo para Blocos Livres no Sistema, onde vai existir uma referência da existência desse Arquivo na **Estrutura de Diretórios** e o File Control Block do Arquivo vai ser copiado para a **Tabela de Arquivos Abertos do Sistema** e se for usado por um Processo vai ser adicionado na **Tabela de Arquivos Abertos do Processo** dentro do Process Control Block, onde caso ele encerre de forma inesperada tenha como saber recuperar ele.
2)
    a) No Sistema de Alocação Encadeada, os Blocos do Arquivo estão espalhados pelos Blocos que existem no Sistema, onde cada Bloco possui o Endereço do Próximo Bloco que deve ser lido em seguida,portanto se quisermos acessar o Quinto Bloco desse Arquivo devemos começar lendo desde o Primeiro e irmos acessando o Próximo Bloco pelos endereços de cada Bloco até chegarmos no Quinto Bloco do Programa, sendo algumas vezes bastante demorado por que tem que passar por todos os Blocos anteriores.
    b) No Sistema FAT temos uma Tabela com os Endereços dos Blocos do Arquivo,onde nesse caso a FAT é um Bloco na Memória com a Lista de Endereços dos Blocos do Arquivo onde cada Endereço possui o Endereço do Próximo Bloco que deve ser acessado, Portanto iremos acessar o Primeiro Endereço na Lista de Endereços da FAT e irmos pegando o Próximo endereço até o Quinto Bloco da Lista, onde pegamos esse endereço e acessamos o Bloco, este é o Quinto Bloco do Arquivo.
    c) No Sistema de Alocação Indexada,teremos um Bloco de índices com todos os Endereços de um Arquivo, onde esse é o Bloco que iremos ler e pegar o Quinto endereço na Lista de Endereços e iremos acessar ele imediatamente no Conjunto de Blocos que existem, sendo essa forma de alocação mais Rápida e direta do que às outras.