

Fundamentos de Sistemas Operativos



Jorge Pais
2016-2017

Fundamentos de Sistemas Operativos

1. Introdução

O sistema operativo é o programa que mais utilização tem em qualquer computador. Seja qual for a área de funcionalidade de um utilizador do computador, desde a área meramente lúdica (jogos, mails, skype, etc) até à área de programador de sistemas (desenvolvimento de aplicações), todos eles utilizam um programa designado por sistema operativo. Os sistemas operativos mais conhecidos e utilizados são o UNIX, Linux e o Microsoft Windows.

Um computador é uma arquitetura hardware que engloba os módulos designados por microprocessador, memória e portos de entrada e saída, mostrados na figura 1.1. Um exemplo duma arquitetura é o arduino que foi um microsistema com que o aluno já trabalhou na disciplina de Computação Física da LEIM.

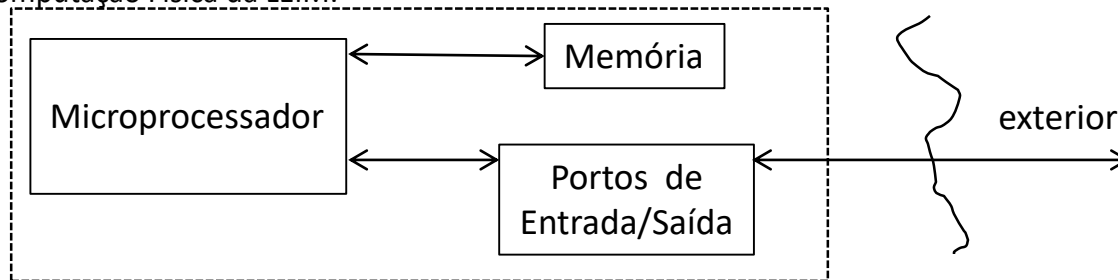


Figura 1.1: Uma arquitetura computacional básica.

A programação ao nível da linguagem máquina de um microprocessador é difícil e consome muito tempo. O mesmo se passa ao nível da configuração, gestão e partilha dos componentes hardware. Desta forma, o objetivo principal do programa designado por *sistema operativo* é facilitar e maximizar a gestão dos recursos hardware ao criar um nível de abstração ao utilizador designado por *máquina virtual*.

A utilização da *máquina virtual* permite manipular os *dispositivos hardware* de uma forma muito mais simples e intuitiva para os vários tipos de utilizadores tal como o ilustrado na figura 1.2.

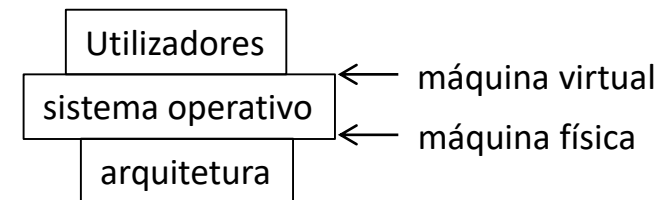


Figura 1.2: Os vários níveis de abstração.

Fundamentos de Sistemas Operativos

2. Tipos de Sistemas Operativos

2.1. Sistemas Operativos Mono-Utilizador

Os sistemas operativos mono-utilizador foram aqueles que inicialmente foram utilizados nos primeiros computadores pessoais, tais como o CPM da Digital Research ou o MS/DOS da Microsoft, estes já dos anos 80.

Estes sistemas operativos eram simples e tiveram o impacto de tornar credível que os computadores poderiam ser utilizados nas mais diversas funções por utilizadores heterogéneos. Tiveram o impacto de desmistificar o uso dos computadores que até então eram propriedade dos especialistas em informática.

2.2. Sistemas Operativos Multi-Utilizador

O primeiro sistema operativo multi-utilizador com grande divulgação e uso foi o Unix para computadores de médio porte que serviam de suporte a aplicações multi-utilizador, tal como a edição de notícias, etc.

Como o Unix é um sistema operativo proprietário, então foi desenvolvido o Linux como um sistema operativo com um modelo semelhante, mas *open source*, o que permitiu o desenvolvimento de muitas aplicações inovadoras e foi adotado em muitas universidades americanas e europeias.

Depois vem o sistema operativo Windows 3.1 da Microsoft que começou por ser uma evolução gráfica do MS/DOS mas que depois degenerou em múltiplas versões multi-utilizador e multi-processos, tais como o Windows 95 e 98, Windows XP, Windows Vista, Windows 7, 8 e 10.

É sobre estes sistemas operativos multi-utilizador, multi-processo e multi-programados que vai acentar o estudo da disciplina de Fundamentos de Sistemas Operativos

Fundamentos de Sistemas Operativos

3. Estudo de um Sistema Operativo

Um sistema operativo pode ser decomposto em camadas funcionais, cada camada é responsável pela gestão eficiente de um recurso físico da máquina hardware. Estas camadas são um modo de definir a concepção de um sistema operativo duma maneira modular e estruturada, embora as camadas não sejam inteiramente disjuntas.

A abordagem ao sistema operativo vai ser feita de forma modular seguindo a hierarquia apresentada na figura 3.1.



Figura 3.1: Abordagem modular e hierárquica a um sistema operativo.

Na disciplina de FSO vamos utilizar a linguagem Java para desenvolvimento de aplicações. Assim, o sistema operativo e cada uma das camadas funcionais vão ser exploradas utilizando a API disponibilizada pela linguagem de programação. A partir desta secção o estudo de cada camada vai ser direccionada e explorada a partir da linguagem Java que acena na Java Virtual Machine ficando o código desenvolvido portátil para vários hardware alvo, desde computadores até telemóveis.

3.1. Gestão de Processos e Tarefas

Por gestão de processos entende-se a partilha da máquina física por um conjunto de programas independentes ou interdependentes. Um processo é um programa que pode ser executado a partir do sistema operativo e que desenvolve uma determinada atividade no computador, por exemplo os editores de texto (Word, Ultraedit, NotePad, etc), os compiladores de linguagens de programação (C, C++, Java, etc), os editores de imagem (Photoshop, Paint, Illustrator, etc), os programas de comunicação (Messenger, Skype, etc). Todos estas aplicações são processos que partilham os recursos físicos duma máquina e que do ponto de vista dos utilizadores dum computador estão a ser executados em paralelo.

Um processo tem vários estados de funcionamento, o diagrama de estados de um processo está ilustrado na figura 3.2.

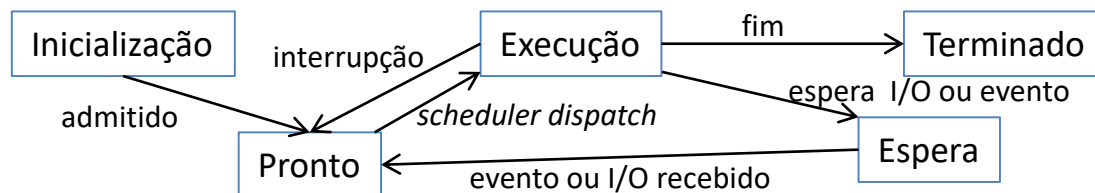


Figura 3.2: Diagrama de estados de um processo.

Inicialização – o processo foi criado mas espera a atribuição de recursos.

Pronto – espera a atribuição do processador.

Execução – está na posse da máquina física.

Espera - espera algum evento ou comunicação.

Terminado – a execução terminou e falta assinalar ao processo pai este acontecimento.

Fundamentos de Sistemas Operativos

3.1.1. Um processo em Java

Em Java, um processo pode ser criado utilizando a classe `ProcessBuilder`. Um processo é uma instância da classe `ProcessBuilder` tendo como parâmetro de entrada o “nomeProcesso” do tipo `String`. No parâmetro “nomeProcesso” consta o nome do processo executável mais o respetivo caminho (path). Por exemplo:

```
ProcessBuilder pb= new ProcessBuilder(nomeProcesso); // cria um processo Win32 passado no parâmetro de entrada .
```

Em Java, um processo é lançado para execução utilizando o seguinte método:

```
Process p= pb.start(); // executa um processo
```

O código completo numa classe denominada “`lançarEEsperarProcessoWin32`” em Java para lançar um ou mais processos está descrito na figura 3.1.1.1.

```
public class lançarEEsperarProcessoWin32{
    public static void main(String[] args) {
        // maneira de executar a partir do Java processos Win32
        ProcessBuilder pb= new ProcessBuilder("c:\\Windows\\system32\\mspaint.exe");
        Process p= null;
        try {
            // executar o processo mspaint
            p = pb.start();
            System.out.println("MsPaint executando.");
        } catch (IOException e) { System.err.println(e.getMessage()); }
        If (p!= null) {
            try {
                // esperar pelo fim do processo mspaint
                p.waitFor();
                System.out.println("MsPaint terminou.");
            } catch (InterruptedException e) { System.err.println(e.getMessage()); }
        }
    }
}
```

Figura 3.1.1.1 – Classe em Java para lançar processos de sistema Win32.

Fundamentos de Sistemas Operativos

Um programa Java pode ser executado a partir do sistema operativo, desde que convertido num ficheiro .JAR executável. Para fazer esta operação, é necessário um programa Java que tenha um método run() que é a base da aplicação para convertê-lo num ficheiro com extensão .JAR executável. Por exemplo, em eclipse a produção do ficheiro .JAR é realizado através da seguinte sequência de comandos:

1. Selecionar File-> Export->Runnable JAR file ;
2. Clicar em NEXT;
3. Definir a Launch configuration dentro das opções disponibilizadas e definir o caminho(*path*) e nome do ficheiro executável com extensão .JAR;
4. Clicar em Finish.

O código completo duma classe denominada “lançarEEesperarProcessoJAVA” em Java executar um ou mais processos está descrito na figura 3.1.1.2.

```
public class lancarEEesperarProcessoJAVA{
    public static void main(String[] args) {
        // maneira de executar a partir do Java um processo JAVA
        Process p= null;
        try {
            p= Runtime.getRuntime().exec("java -jar C:\\Users\\jpais\\Desktop\\T1.jar ");
            System.out.println("T1 executando.");
        } catch (IOException e) { e.printStackTrace(); }
        if (p!= null) {
            try {
                // esperar pelo fim do processo T1
                p.waitFor();
                System.out.println("T1 terminou.");
            } catch (InterruptedException e) { System.err.println(e.getMessage()); }
        }
    }
}
```

Figura 3.1.1.2 – Classe em Java para lançar processos JAVA.

Fundamentos de Sistemas Operativos

3.1.2. Comunicação entre processos em Java

A comunicação em JAVA entre processos independentes (IPC – Inter-Process Communication) na mesma máquina pode ser realizado com as seguintes tecnologias:

- Memória partilhada;
- Sockets por TCP (*Transfer Control Protocol*);
- Sockets por UDP (*User Datagram Protocol*).

A comunicação entre máquinas diferentes é realizada pelas tecnologias:

- Sockets por TCP (*Transfer Control Protocol*);
- Sockets por UDP (*User Datagram Protocol*).

Função das técnicas a passagem da informação entre processos pode demorar mais ou menos tempo, os tempos para cada uma das técnicas são da seguinte ordem de grandeza:

Memória partilhada e 1 core de cpu:	1,5 microsegundos;
Memória partilhada e mais de 1 core de cpu:	0,7 microsegundos;
TCP e 1 core de CPU :	30 microsegundos;
TCP e mais de 1 core de CPU :	22 microsegundos;
UDP e 1 core de CPU:	5 microsegundos;
UDP e mais de 1 core de CPU:	8 microsegundos.

Fundamentos de Sistemas Operativos

3.1.3. Comunicação entre processos com Memória Partilhada

A comunicação em JAVA entre processos independentes (IPC – Inter-Process Communication) na mesma máquina pode ser realizado de forma eficiente através de memória partilhada que em JAVA é designada por memória mapeada.

A memória mapeada em Java tem um modelo que mapeia numa área de memória virtual o conteúdo de um ficheiro. Esta memória pode ser manipulada como sendo um buffer normal sem necessidade das operações de leitura e escrita explícitas sobre ficheiro. O conteúdo desta memória pode ou não estar disponível na memória principal do processo, portanto o tempo de acesso aos dados pode variar um pouco com a utilização da memória mapeada.

A classe que suporta o conceito de memória mapeada é o tipo `MappedByteBuffer` que tem a seguinte definição:

```
public abstract class MappedByteBuffer extends ByteBuffer
```

Uma instância do tipo `MappedByteBuffer` representa um buffer de acesso direto com conteúdo igual ao de um ficheiro.

As instâncias de buffers do tipo `MappedByteBuffer` são criados através do método `FileChannel.map` e existem até as instâncias deixarem de existir no processo.

Qualquer processo JAVA pode alterar o conteúdo de um `MappedByteBuffer`.

Se o `MappedByteBuffer` for truncado ou fechado por um processo, a partir daqui, todos os processos deixam de aceder ao `MappedByteBuffer`, gerando-se uma exceção de acesso.

A manipulação de um `MappedByteBuffer` é exatamente igual à de um `Buffer`. Apenas adiciona os seguintes métodos:

`MappedByteBuffer force()` – obriga a atualização da memória secundária com o conteúdo do buffer.

`boolean isLoaded()` – indica se o conteúdo do buffer está em memória principal.

`MappedByteBuffer load()` – carrega o conteúdo do buffer em memória principal.

Fundamentos de Sistemas Operativos

Exercicio: Implemente uma classe que permita a comunicação por memória partilhada entre processos JAVA no mesmo computador.

Resolução:

A memória partilhada em JAVA é uma memória mapeada de um ficheiro que tem de ser conhecido dos vários processos através do path e nome do ficheiro.

A classe tem o nome `processoM` e tem a composição mostrada na figura 3.1.3.1.

```
public class ProcessoM {  
    // ficheiro  
    File ficheiro;  
    // canal que liga o conteúdo do ficheiro ao Buffer  
    FileChannel canal;  
    // buffer  
    MappedByteBuffer buffer;  
    // dimensão máxima em bytes do buffer  
    final int BUFFER_MAX= 30;  
  
    // construtor onde se cria o canal  
    ProcessoM(){ }  
  
    // recebe uma mensagem convertendo-a numa String  
    String receberMensagem() {}  
  
    // envia uma String como mensagem  
    void enviarMensagem(String) {}  
  
    // fecha o canal de comunicação  
    void fecharCanal() {}  
  
    // método run do processo  
    void run() {}  
}
```

Figura 3.1.3.1 – Estrutura da classe `ProcessoM`.

Fundamentos de Sistemas Operativos

```
ProcessoM(){
    // cria um ficheiro com o nome comunicacao.dat
    ficheiro = new File("comunicacao.dat");

    //cria um canal de comunicação de leitura e escrita
    try {
        canal = new RandomAccessFile(ficheiro, "rw").getChannel();
    } catch (FileNotFoundException e) {e.printStackTrace(); }

    // mapeia para memória o conteúdo do ficheiro
    try {
        buffer = canal.map(FileChannel.MapMode.READ_WRITE, 0, BUFFER_MAX);
    } catch (IOException e) { e.printStackTrace(); }
}

// recebe uma mensagem convertendo-a numa String
String receberMensagem() {
    String msg=new String();
    char c;
    buffer.position(0);
    while (c= buffer.getChar()) != '\0'
        msg += c;
    return msg;
}
```

1 de 2

```
// envia uma String como mensagem
void enviarMensagem(String msg) {
    char c;
    buffer.position(0);
    for (int i= 0 ; i< msg.length() ; ++i){
        c= msg.charAt(i);
        buffer.putChar(c);
    }
    buffer.putChar('\0');
}

// fecha o canal entre o buffer e o ficheiro
void fecharCanal() {
    try {
        canal.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2 de 2

Fundamentos de Sistemas Operativos

3.2. Processos leves ou Tarefas

Os processos leves ou tarefas executam-se só num computador e são partes de um único processo designado por processo-pai que os lança e executa. Os processos leves cooperam dentro dum processo-pai para realizar uma funcionalidade desejada em pseudo-parallelismo num processador de um core ou em parallelismo num processador com múltiplos-cores. Os processos leves concorrem e partilham os recursos do computador e podem partilhar os dados do processo-pai.

Como simplificação do texto, ao conceito de processo leve vamos abreviar e apenas designar como **Tarefa**.

3.2.1. Tarefa em Java

Um processo pode ser desenhado de modo a conter múltiplas tarefas. Uma tarefa é a implementação de uma atividade dentro de um processo tendo uma estrutura de dados e código independente das outras tarefas. Uma tarefa em Java pode ser implementada como uma classe Java que deriva da classe Thread herdando assim todos os métodos desta classe.

A classe Thread é acedida através da biblioteca *java.lang.Thread* e tem a declaração *public class Thread extends Object implements Runnable*.

Uma classe que derive da class Thread é uma tarefa que se executa dentro duma aplicação ou processo-pai. A máquina virtual Java permite a execução de várias tarefas (Threads) dentro de um processo.

Há duas maneiras para definir uma classe do tipo Thread, que de seguida se explica.

Fundamentos de Sistemas Operativos

1. Uma das maneiras é a seguinte:

```
public class controlarRobot1 extends Thread {  
    ArrayList<String> comandos;  
    controlarRobot1() {  
        comandos= new ArrayList<String>();  
    }  
    run() {  
        /* interpretar e executar os comandos */  
    }  
}
```

O modo para criar e lançar uma instância da classe controlarRobot1 é a seguinte:

```
    controlarRobot1 cr1= new controlarRobot1();    // cria uma instância da classe controlarRobot1  
    cr1.start();                                   // executa o método run() da classe controlarRobot1
```

2. A outra maneira alternativa é:

```
public class controlarRobot2 implements Runnable {  
    ArrayList<String> comandos;  
    controlarRobot2() {  
        comandos= new ArrayList<String>();  
    }  
    run() {  
        /* interpretar e executar os comandos */  
    }  
}
```

A maneira para criar e lançar uma instância da classe controlarRobot2 é a seguinte:

```
controleRobot2 cr2= new controlarRobot2();           // cria uma instância da classe controlarRobot2  
new Thread(cr2).start();                             // executa o método run() da classe controlarRobot2
```

Fundamentos de Sistemas Operativos

3.2.2. Sincronização entre Tarefas

A comunicação entre tarefas exige sincronização, porque o tempo de processamento de cada tarefa e a sua execução é imprevisível num computador, as tarefas executam-se de forma assíncrona. Portanto, para se conseguir êxito na cooperação entre as várias tarefas é preciso haver pontos de sincronismo nas atividades. Um ponto de sincronismo, é um ponto no código em que uma tarefa espera por outra tarefa em relação às atividades a executar.

Quando duas ou mais tarefas acedem nas suas atividades a recursos não partilháveis, como por exemplo, dispositivos de entrada/saída, canais de comunicação, ficheiros ou dados em memória, estes dispositivos devem ser protegidos do acesso simultâneo das várias tarefas. Uma das técnicas de programação para proibir o acesso simultâneo a um recurso físico é a técnica designada por exclusão mútua.

- Exclusão Mútua

O acesso ao robot utiliza o canal de comunicação bluetooth que é um recurso único no computador e se existirem duas tarefas no nosso computador que queiram aceder ao canal bluetooth este tem de ser feito em exclusão mútua. Por exemplo, analise o código da figura 3.2.2.1..

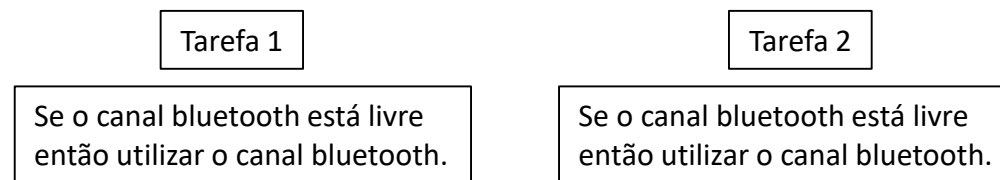


Figura 3.2.2.1 – Tarefas concorrentes a aceder a um canal bluetooth e ao mesmo recurso físico.

O código da figura 3.2.2.1. está errado porque permite que dois processos acedam ao canal bluetooth simultaneamente porque a execução de cada tarefa é independente da outra e o teste do canal bluetooth estar livre pode ser feito simultaneamente e ambas as tarefas avançarem para a utilização do canal bluetooth.

Fundamentos de Sistemas Operativos

O problema do acesso simultâneo de duas tarefas ao canal bluetooth estende-se a todos os recursos não partilháveis que incluem praticamente todos os periféricos, ficheiros para escrita e as áreas de dados que se pretendem modificar. Os recursos não partilháveis só podem ser acedidos por uma tarefa de cada vez, sendo o programa desenvolvido que deve de garantir esta exclusão mútua.

Se redesenharmos o pseudo-código das tarefas 1 e 2 da figura 3.2.2.1 para o código da figura 3.2.2.2.

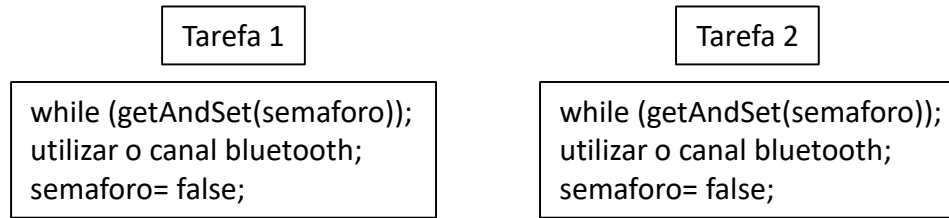


Figura 3.2.2.2 – Tarefas concorrentes a aceder a um canal bluetooth e ao mesmo recurso físico em exclusão mútua.

A exclusão mútua ao canal bluetooth é garantida pela variável “semaforo” que tem de ser conhecida pelos duas tarefas conjuntamente com a instrução `getAndSet(AtomicBoolean)` existente na máquina virtual Java. Esta instrução devolve o valor lógico da variável passada como parâmetro de entrada e afeta a variável com o valor `true`.

Com esta manipulação duma variável booleana garante-se que só uma tarefa acede ao canal bluetooth de cada vez, chamando-se a esta técnica de exclusão mútua entre tarefas num acesso a um recurso não-partilhável.

- Semáforos em Java

O conceito de semáforo foi introduzido por Dijkstra em 1965 e foi a primeira entidade que permitia a comunicação entre processos. A operação sobre um semáforo “s” que permite o incremento do seu valor inteiro designava-se originalmente por Dijkstra como *V(s)* (*abreviatura da palavra holandesa Verhogen que significa em português Incrementar*), em terminologia anglo-saxónica por *signal(s)* e em java por `s.release()` sendo s do tipo `semaphore`. A operação sobre um semáforo “s” que permite o decremento do seu valor inteiro quando $s > 0$, originalmente por Dijkstra designava-se por *P(s)* (*abreviatura da palavra holandesa Proberan que significa Testar*), em terminologia anglo-saxónica por `wait(s)` e em Java por `s.acquire()`.

Fundamentos de Sistemas Operativos

O acesso simultâneo de dois processos ao canal bluetooth resolvido em Java ficaria com o pseudo-código da figura 3.1.4.3.

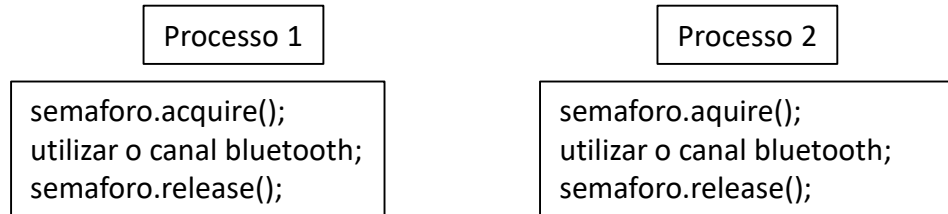


Figura 3.2.2.3 – Processos concorrentes em Java a aceder ao mesmo recurso físico em exclusão mútua.

O código da figura 3.2.2.3. pode provocar um deadlock nos dois processos caso o valor do semáforo tenha sido inicializado com zero unidades. Pois cada uma das tarefas quando executam o método `acquire` são suspensas e ficam na lista de espera do semáforo até que outro processo faça um `release`.

Quando os processos ficam à espera dum recurso e o recurso não é libertado por falha algoritmica ou outra falha qualquer, a esta situação designa-se por deadlock. E o efeito sentido pelo utilizador é a aplicação onde os processos estão inseridos falhar ou até mesmo parar a sua execução.

Para o código da figura 3.2.2.3 funcionar, no ato de criação do semáforo, este tem de ser criado com o valor inicial de 1 para não haver deadlock nas tarefas 1 e 2. Em java o código de inicialização de um semáforo com 1 unidade e o lançamento dos processos 1 e 2 é ilustrado na figura 3.2.2.4.

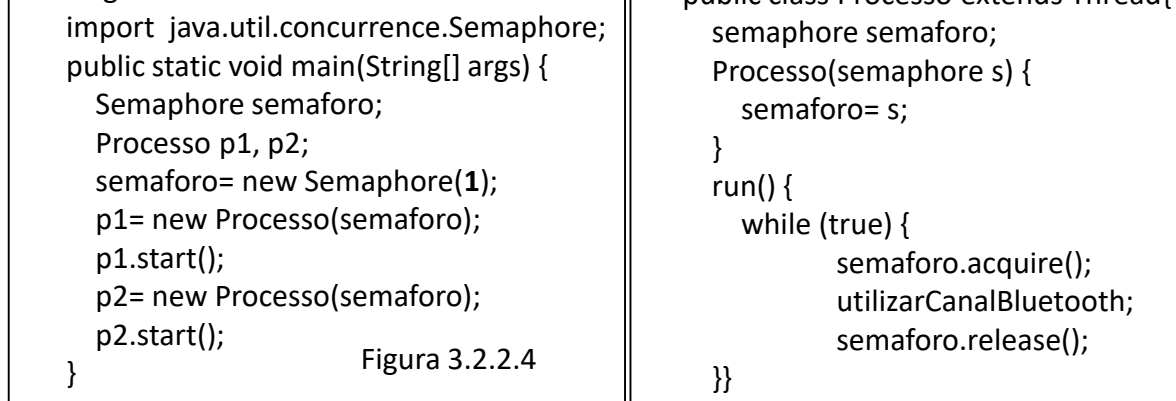


Figura 3.2.2.4

Fundamentos de Sistemas Operativos

Outra situação típica de deadlock quando vários processos utilizam os mesmos recursos não-partilháveis ou esperam uns pelos outros para completar certas acções, na figura 3.2.2.5 ilustra-se outro exemplo com dois semáforos e com duas tarefas.

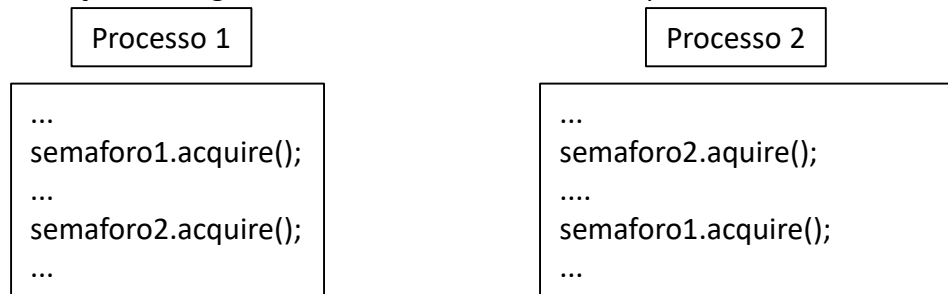


Figura 3.2.2.5 – Processos concorrentes em deadlock.

Considerando que os valores iniciais dos semáforos *semaforo1* e *semaforo2* têm o valor unitário, 1, e quando completam a primeira execução de *acquire* ficam reduzidos ao valor zero, 0, então na segunda execução do *acquire* os dois processos ficam suspensos em cada um dos semáforos à espera que alguém faça release. Como não existe mais nenhum processo, a execução dos dois processos é suspensa ocorrendo uma situação de deadlock.

A partilha de um semáforo é definida pelo seu valor inicial, se o valor for **unitário** apenas **um** processo o partilha e se o valor do semáforo for **n** então **n** processos podem partilhar o semáforo ou o recurso abrangido pelo semáforo.

A ilação mais importante destes exemplos é que o **deadlock** pode ocorrer como resultado de uma sequência incorreta de execuções de *acquire* sobre vários semáforos.

Existem sistemas de prova que permitem avaliar qual a probabilidade de acontecer deadlock numa determinada aplicação multi-processo. Estes sistemas de prova demonstram que seja qual for o fio de execução de um programa multi-processo para um dado semáforo inicializado com o valor N, é respeitada a condição no tempo:

$$\text{Quantidade(acquire)} \leq \text{Quantidade(release)} + N$$

garantindo-se assim que não existe deadlock.

A verificação desta condição para todos os semáforos numa aplicação multi-processo é uma tarefa bastante complexa, mas 16 que hoje já se faz na produção e certificação de código para a indústria de microprocessadores e aeronáutica entre outras.

Fundamentos de Sistemas Operativos

Exercício: Considere um buffer circular de N posições no qual vários processos produtor pretendem colocar mensagens do tipo String e vários processos consumidor pretendem ler as strings. Pretende-se que desenhe três classes, uma classe BufferCircular, uma classe Produtor e uma classe Consumidor.

A classe BufferCircular com uma interface que possa ser utilizada pelos processos Produtor e Consumidor garantindo a integridade e não perda de informação na manipulação do Buffer. A solução para a classe BufferCircular é a seguinte:

```
public class BufferCircularString {
    final int dimensaoBuffer= 4;
    String[] bufferCircular;
    int putBuffer, getBuffer;
    // o semáforo elementosLivres indica se há posições livres para inserir Strings
    // o semáforo acessoElemento garante exclusão mútua no acesso a um elemento
    // o semáforo elementosOcupados indica se há posições com Strings válidas
    Semaphore elementosLivres, acessoElemento, elementosOcupados;

    public BufferCircularString(){
        bufferCircular= new String[dimensaoBuffer];
        putBuffer= 0;
        getBuffer= 0;
        elementosLivres= new Semaphore(dimensaoBuffer);
        elementosOcupados= new Semaphore(0);
        acessoElemento= new Semaphore(1);
    }
}
```

1 de 2

```
public void inserirElemento(String s){
    try {
        elementosLivres.acquire();
        acessoElemento.acquire();
        bufferCircular[putBuffer]= new String(s);
        putBuffer= ++putBuffer % dimensaoBuffer;
        acessoElemento.release();
    } catch (InterruptedException e) {}
    elementosOcupados.release();
}

public String removerElemento() {
    String s= new String();
    try {
        elementosOcupados.acquire();
        acessoElemento.acquire();
    } catch (InterruptedException e) {}
    s= bufferCircular[getBuffer];
    getBuffer= ++getBuffer % dimensaoBuffer;
    acessoElemento.release();
    elementosLivres.release();
    return s;
}
```

2 de 2 17

Fundamentos de Sistemas Operativos

A solução para as classes Produtor e Consumidor são as seguintes:

```
public class Produtor extends Thread {
    BufferCircularString bufferCircular;
    String myString;
    Semaphore haTrabalho, livreMyString, ocupadaMyString, acessoMyString;
    Produtor(BufferCircularString bc, Semaphore ht) {
        bufferCircular= bc;
        haTrabalho= ht;
        myString= new String();
        LivreMyString= new Semaphore(1);
        ocupadaMyString= new Semaphore(0);
        acessoMyString= new Semaphore(1);
    }
    public void setString(String s){
        try {
            livreMyString.acquire();
            acessoMyString.acquire();
        } catch (InterruptedException e) {}
        myString= s;
        acessoMyString. Release();
        ocupadaMyString.release();
    }
    public void run() {
        while (true) {
            try {
                ocupadaMyString.acquire();
                acessoMyString.acquire();
            } catch (InterruptedException e) {}
            bufferCircular.inserirElemento(myString);
            acessoMyString. Release();
            livreMyString.release();
            haTrabalho.release();
        }
    }
}
```

```
public class Consumidor extends Thread {
    String myString;
    BufferCircularString bufferCircular;
    Semaphore livreMyString, ocupadaMyString, haTrabalho, acessoMyString;
    public Consumidor(BufferCircularString bc, Semaphore ht) {
        bufferCircular= bc;
        haTrabalho= ht;
        myString= new String();
        livreMyString= new Semaphore(1);
        ocupadaMyString= new Semaphore(0);
        acessoMyString= new Semaphore(1);
    }
    public String getString() {
        try { ocupadaMyString.acquire();
            acessoMyString.acquire();
        } catch (InterruptedException e) {}
        String s= new String(myString);
        acessoMyString.release();
        livreMyString.release();
        return s;
    }
    public void run() {
        while (true) {
            try { haTrabalho.acquire();
                livreMyString.acquire();
            } catch (InterruptedException e) {}
            String s= new String(bufferCircular.removerElemento());
            try { acessoMyString.acquire(); }
            catch (InterruptedException e) {}
            myString= s;
            acessoMyString.release();
            ocupadaMyString.release();
        }
    }
}
```

Fundamentos de Sistemas Operativos

- Monitores

Há métodos que devido à sua especificidade e implementação devem ser executados em exclusão mútua num sistema multi-processo.

Uma solução consiste na criação de um semáforo por cada método para garantir exclusão mútua no acesso a um método. Por exemplo,

```
Semaphore AcessoExclusaoMutua1= new Semaphore(1);
...
void metodo1 throws InterruptedException() {
    AcessoExclusaoMutua1.acquire();
    // código do método
    ...
    AcessoExclusaoMutua1.release();
}

Semaphore AcessoExclusaoMutua2= new Semaphore(1);
...
void metodo2 throws InterruptedException() {
    AcessoExclusaoMutua2.acquire();
    // código do método
    ...
    AcessoExclusaoMutua2.release();
}
...
```

Fundamentos de Sistemas Operativos

Outra solução para o acesso em exclusão mútua a métodos é o uso de monitores.

Os monitores têm como objetivo garantir que métodos sejam executados em exclusividade por múltiplos processos e sem a complexidade inerente ao uso de semáforos que são fonte de erros na implementação de aplicações.

Os monitores nasceram na década de 70 e existem as seguintes variantes:

Monitor de Hoare - o processo que executa o acordar (processo que executa um dos métodos `notify()` ou `notifyAll()` em java) é bloqueado na fila interna do estado “pronto para execução”(ready) e o processo que acorda (processo que executou o método `wait()`, em java, estando bloqueado) é colocado em execução.

Monitor de Lampson ou Brinch-Hasson – o processo que executa o acordar (`notify()` ou `notifyAll()` em Java) continua em execução e o processo que foi desbloqueado ou os processos que foram desbloqueados são colocados na fila interna de processos do estado “pronto para execução” (ready).

Monitores em Java

Os monitores em Java são do tipo Lampson e são suportados pela própria linguagem. Cada objecto em Java tem associado um monitor que é implementado a partir duma flag lock. O acesso a esta flag lock é permitida através do uso da diretiva `synchronized`. Os vários tipos de sincronização existentes são:

1. Sincronização ao método – com a utilização de `synchronized` antes da definição do método. Este método é executado apenas por um processo.
2. Sincronização ao bloco – com a utilização de `synchronized` aplicada a um bloco de código. Este bloco de código é definido como um conjunto de instruções que tem de ser executado em exclusividade. A utilização do método `wait()` serve para garantir que só um processo executa o bloco de código. Os métodos `notify()` e `notifyAll()` notificam outros processos que o bloco está disponível. Para o exemplo da folha anterior, o código dos métodos `metodo1` e `metodo2` utilizando monitores Java ficaria,

```
synchronized void metodo1() {  
    // código do método 1  
    ...  
}
```

```
synchronized void metodo2() {  
    // código do método 2  
    ...  
}
```

Fundamentos de Sistemas Operativos

A classes Produtor e Consumidor com **synchronized no método**:

```
public class Produtor extends Thread {
    BufferCircularString bufferCircular;
    String myString;
    Semaphore haTrabalho, livreMyString, ocupadaMyString;
    public Produtor(BufferCircularString bc, Semaphore ht) {
        bufferCircular= bc;
        haTrabalho= ht;
        myString= new String();
        LivreMyString= new Semaphore(1);
        ocupadaMyString= new Semaphore(0);
    }
    public synchronized void setString(String s){
        try { livreMyString.acquire();
        } catch (InterruptedException e) {}
        myString= s;
        ocupadaMyString.release();
    }
    private synchronized void inserirString(String s) {
        try{ ocupadaMyString.acquire();
        } catch (InterruptedException e) {}
        bufferCircular.inserirElemento(s);
        livreMyString.release();
    }
    public void run() {
        while (true) {
            inserirString(myString);
            haTrabalho.release();
        }
    }
}
```

1 de 2

```
public class Consumidor extends Thread {
    String myString;
    BufferCircularString bufferCircular;
    Semaphore livreMyString, ocupadaMyString, haTrabalho;
    public Consumidor(BufferCircularString bc, Semaphore ht) {
        bufferCircular= bc;
        haTrabalho= ht;
        myString= new String();
        livreMyString= new Semaphore(1);
        ocupadaMyString= new Semaphore(0);
    }
    public synchronized String getString() {
        try { ocupadaMyString.acquire();
        } catch (InterruptedException e) {}
        String s= new String(myString);
        livreMyString.release();
        return s;
    }
    private synchronized void obterString() {
        try { livreMyString.acquire();
        } catch (InterruptedException e) {}
        myString= bufferCircular.removerElemento();
        ocupadaMyString.release();
    }
    public void run() {
        while (true) {
            try { haTrabalho.acquire();
            } catch (InterruptedException e) {}
            obterString();
        }
    }
}
```

2 de 2

21

Fundamentos de Sistemas Operativos

A classe Produtor com **synchronized** no bloco:

```
public class Produtor extends Thread {
    BufferCircularString bufferCircular;
    String myString;
    Semaphore haTrabalho, livreMyString, ocupadaMyString;
    public Produtor(BufferCircularString bc, Semaphore ht) {
        bufferCircular= bc;
        haTrabalho= ht;
        myString= new String();
        LivreMyString= new Semaphore(1);
        ocupadaMyString= new Semaphore(0);
    }
    public void setString(String s){
        try { livreMyString.acquire();
        } catch (InterruptedException e) {}
        synchronized (myString) {
            myString= s;
            myString.notify();
        }
        ocupadaMyString.release();
    }
    public void run() {
        while (true) {
            try{ ocupadaMyString.acquire();
            } catch (InterruptedException e) {}
            synchronized(myString) {
                try{ myString.wait();
                } catch (InterruptedException e){}
                bufferCircular.inserirElemento(myString);
            }
            livreMyString.release();
            haTrabalho.release();
        }
    }
}
```

Fundamentos de Sistemas Operativos

4. Sistema de ficheiros

Um sistema de ficheiros de uso geral é um programa que é parte do sistema operativo e que tem por função gerir as entidades designadas por ficheiros. Um ficheiro é uma coleção de bytes que pode ser um programa, um texto, um conjunto de fotografias, uma música, etc. Um ficheiro é a entidade que é guardada e manipulada pelo sistema de ficheiros.

O suporte físico de um ficheiro pode ser um disco rígido, uma pen drive, uma fita magnética, etc, ou seja, qualquer dispositivo físico que permita o armazenamento permanente de bits. Os suportes físicos são organizados em blocos de dimensão fixa , por exemplo, 1kbyte, 2kbytes, 4 kbytes, etc. Estes blocos são tantos quantos a dimensão do dispositivo físico utilizado. No dispositivo físico também tem de existir uma tabela que descreve o estado dos blocos, ocupado ou livre. A tabela de descrição dos blocos bem como a informação constante nos blocos que constituem um ficheiro são geridas pelo sistema de ficheiros.

Desta forma, o sistema de ficheiros deve disponibilizar as seguintes funcionalidades:

1. Criar e eliminar um ficheiro;
2. Leitura e escrita de um ficheiro;
3. Acesso a um ficheiro através de um nome lógico.
4. Gerir o espaço de memória ocupado pelo ficheiro no dispositivo físico de suporte para que a organização física do ficheiro seja abstrata ao utilizador.
5. Proteger os ficheiros contra falhas do sistema de suporte.
6. Segurança no acesso aos ficheiros.

Fundamentos de Sistemas Operativos

4.1 Suporte físico

O suporte físico mais comum de um sistema de ficheiros é um disco rígido que tem a estrutura interna apresentada na figura 4.1.1.



Figura 4.1.1 : Estrutura física de um disco rígido para armazenamento de dados binários.

Na figura 4.1.1, à esquerda observa-se as várias partes para controlo de um disco rígido. Ao centro, pode observar que um disco rígido pode ser constituído por vários pratos ou discos. À direita, observa a divisão física de cada prato ou disco que é dividido em pistas e cada pista é dividida em sectores. A unidade elementar de armazenamento de informação é o sector que pode armazenar 1Kbyte, 2Kbytes, 4Kbytes, etc, consoante a dimensão de cada disco.

Fundamentos de Sistemas Operativos

4.2 Definições

Um ficheiro é uma entidade abstrata identificada por um nome e que fisicamente contém valores binários organizados em bytes e que ocupa num disco o conjunto de sectores necessários para o armazenamento total do ficheiro. Por exemplo um ficheiro com 1Mbyte de dimensão para ser armazenado num disco com sectores de 4Kbytes ocupa no disco $10^6/4 \cdot 10^3 = 250$ sectores. Um sector físico geralmente dá lugar à noção abstrata de *cluster* que define a unidade mínima de ocupação de um ficheiro.

Os possíveis atributos de um ficheiro necessários à sua descrição são:

1. Nome do ficheiro – um conjunto de caracteres;
2. Extensão – um conjunto de 3 caracteres que define o tipo do ficheiro;
3. Protecção - Permissões de acesso;
4. Palavra Chave – conjunto de caracteres que permite o acesso ao ficheiro;
5. Dono ou criador do ficheiro – o utilizador proprietário do ficheiro;
6. Directoria – é uma flag que indica se o ficheiro é uma directoria ou um ficheiro de dados;
7. Tipo de acesso – indica o tipo de acções realizadas por quem acede ao ficheiro, só de leitura ou leitura e escrita;
8. Invisível – flag que indica se o ficheiro é ou não invisível nas listagens;
9. Ficheiro de Sistema – flag que indica se o ficheiro é normal ou de sistema;
10. Ficheiro de Arquivo – flag que indica se o ficheiro é normal ou de arquivo;
11. Acesso Aleatório – indica o tipo de acesso sequencial ou aleatório;
12. Ficheiro Temporário – indica que é um ficheiro persistente ou temporário;
13. Flag de lock – indica se o acesso ao ficheiro foi trancado por algum processo;
14. Primeiro sector – indica o primeiro sector onde começa a informação do ficheiro;
15. Data e hora da criação
16. Data e hora da última alteração
17. Data e hora do último acesso
18. Dimensão corrente
19. Dimensão máxima

Fundamentos de Sistemas Operativos

4.3 Volumes

Um disco pode ser linearizado, ver figura 3.2.3.1, em pistas e sectores que por sua vez podem descrever um ou mais volumes, como o mostrado na figura 3.2.3.2.

Pista 0

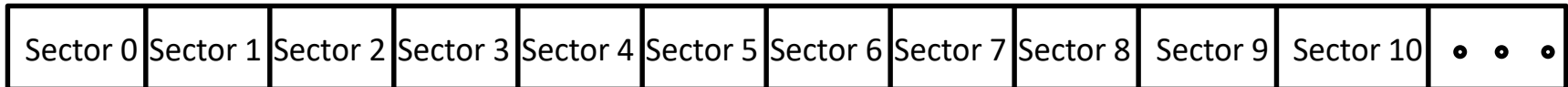


Figura 4.3.1: Linearização de uma pista de um disco



Figura 4.3.2: Disco com vários volumes.

A partição Master Boot Record de um disco contém quatro descritores de partições, cada descritor contém:

1. Active flag – indica se a partição está ativa ou não;
2. Início absoluto da partição – qual o sector onde começa a partição.
3. Dimensão da partição – a partir desta informação define-se o sector de fim da partição.
4. Tipo de partição.

A partição *Extended* pode conter várias partições lógicas. Cada partição lógica tem um *extended boot record* que a descreve. A última partição lógica é preenchida com zeros.

Fundamentos de Sistemas Operativos

4.4 Estrutura duma partição

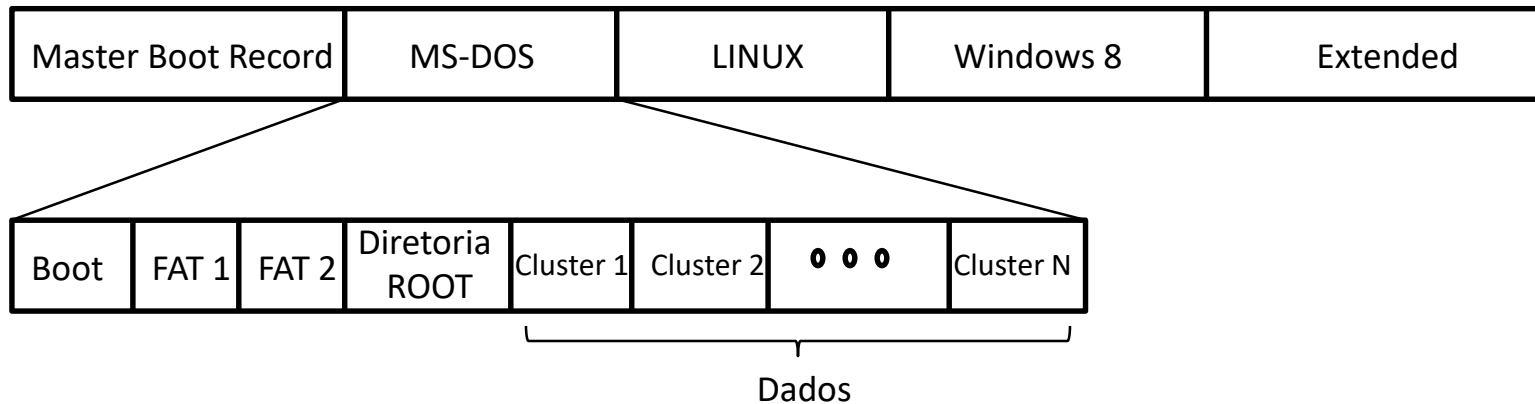


Figura 4.4.1: Estrutura duma partição.

Boot – É um setor com o código de arranque do sistema operativo e com as definições do volume.

FAT1 – File Allocation Table é uma tabela com informação acerca da ocupação e tipo dos clusters. Cluster é um bloco de informação.

FAT2 – É uma duplicação da FAT1 para garantir coerência na informação e aumentar a tolerância a falhas.

Diretoria ROOT – dimensão fixa com 512 entradas por omissão.

Clusters – Blocos de informação com uma dimensão múltipla do sector.

Fundamentos de Sistemas Operativos

4.5 Estrutura dum diretoria

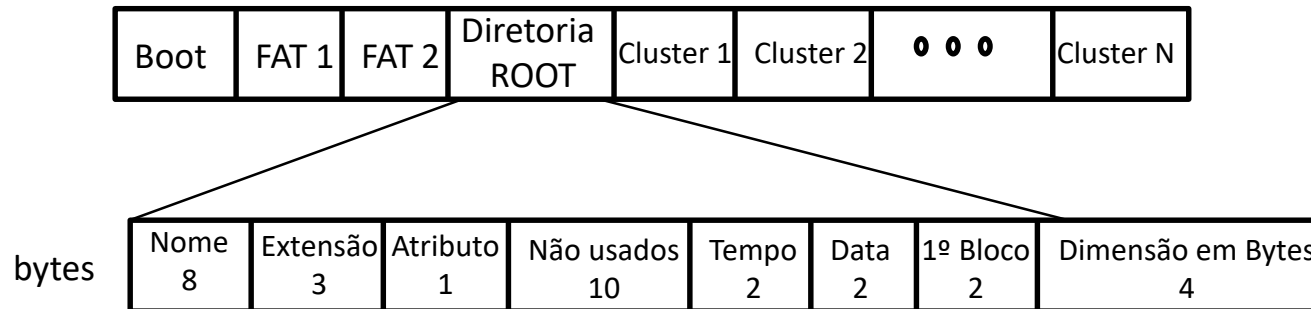


Figura 4.5.1: Estrutura dum diretoria.

Nome – Conjunto de 8 caracteres.

Extensão – Conjunto de 3 caracteres.

Atributo – Este carácter tem os seguintes valores e significados: A – Arquivo ; D – Diretoria; V – Nome do volume; S – *System Files*; H – *Hidden File*; R – *Read-Only*.

Tempo – hora da criação

Data – data da criação

1º Bloco - Informação de onde começa.

Dimensão em bytes – $2^{32} = 4\text{Gbytes}$ dimensão máxima de um ficheiro em FAT.

Fundamentos de Sistemas Operativos

4.6 Estrutura dum FAT

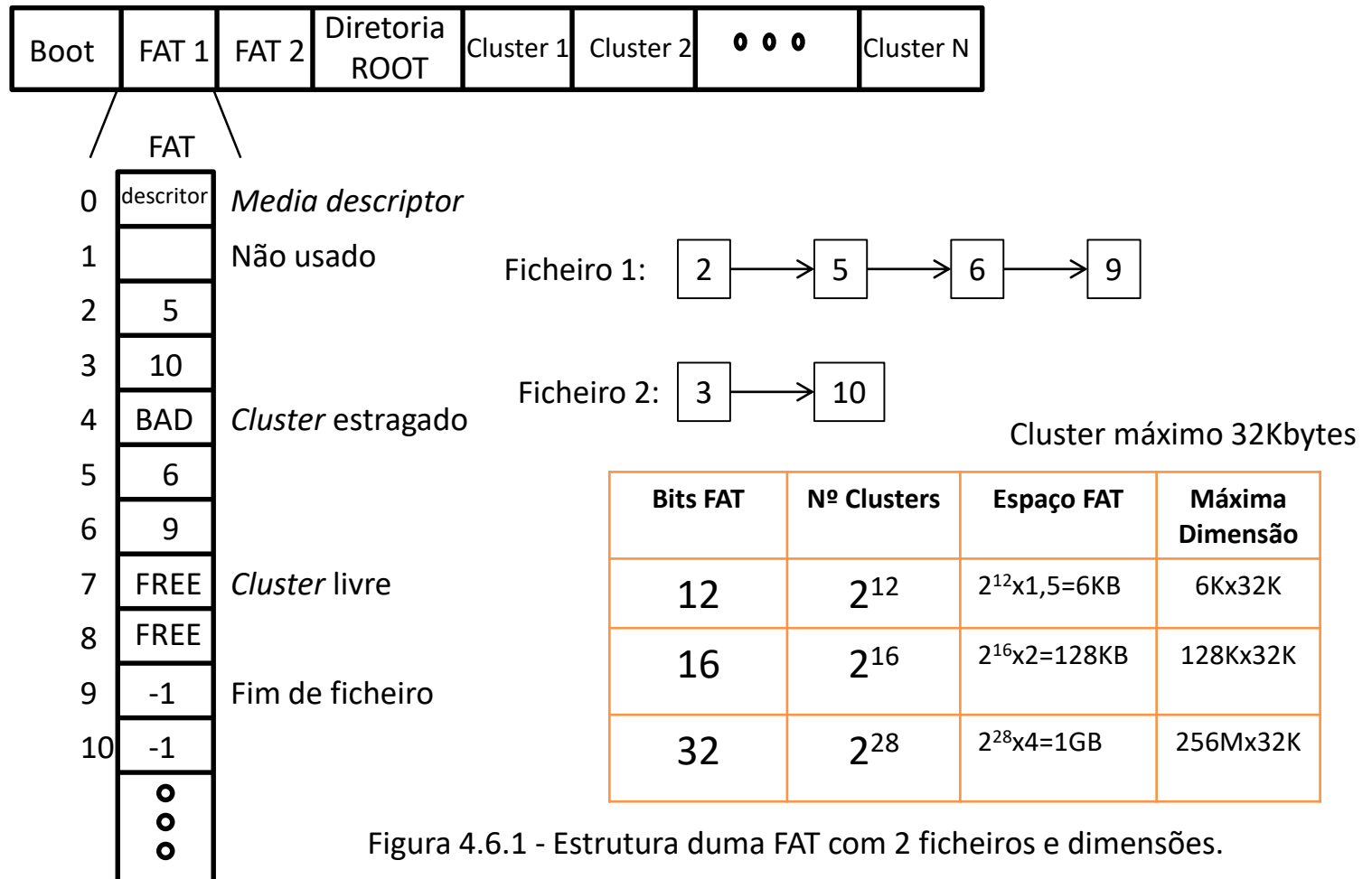


Figura 4.6.1 - Estrutura dum FAT com 2 ficheiros e dimensões.

Fundamentos de Sistemas Operativos

4.7 Manipulação de Ficheiros em Java através de Input e Output Streams

Uma stream é uma entidade lógica que se comporta como um canal de leitura ou escrita com acesso a ficheiros ou a dispositivos de Entrada/Saída (*I/O devices*).

A hierarquia de classes para Leitura de ficheiros e dispositivos de entrada/saída está ilustrada na figura 3.3.1.

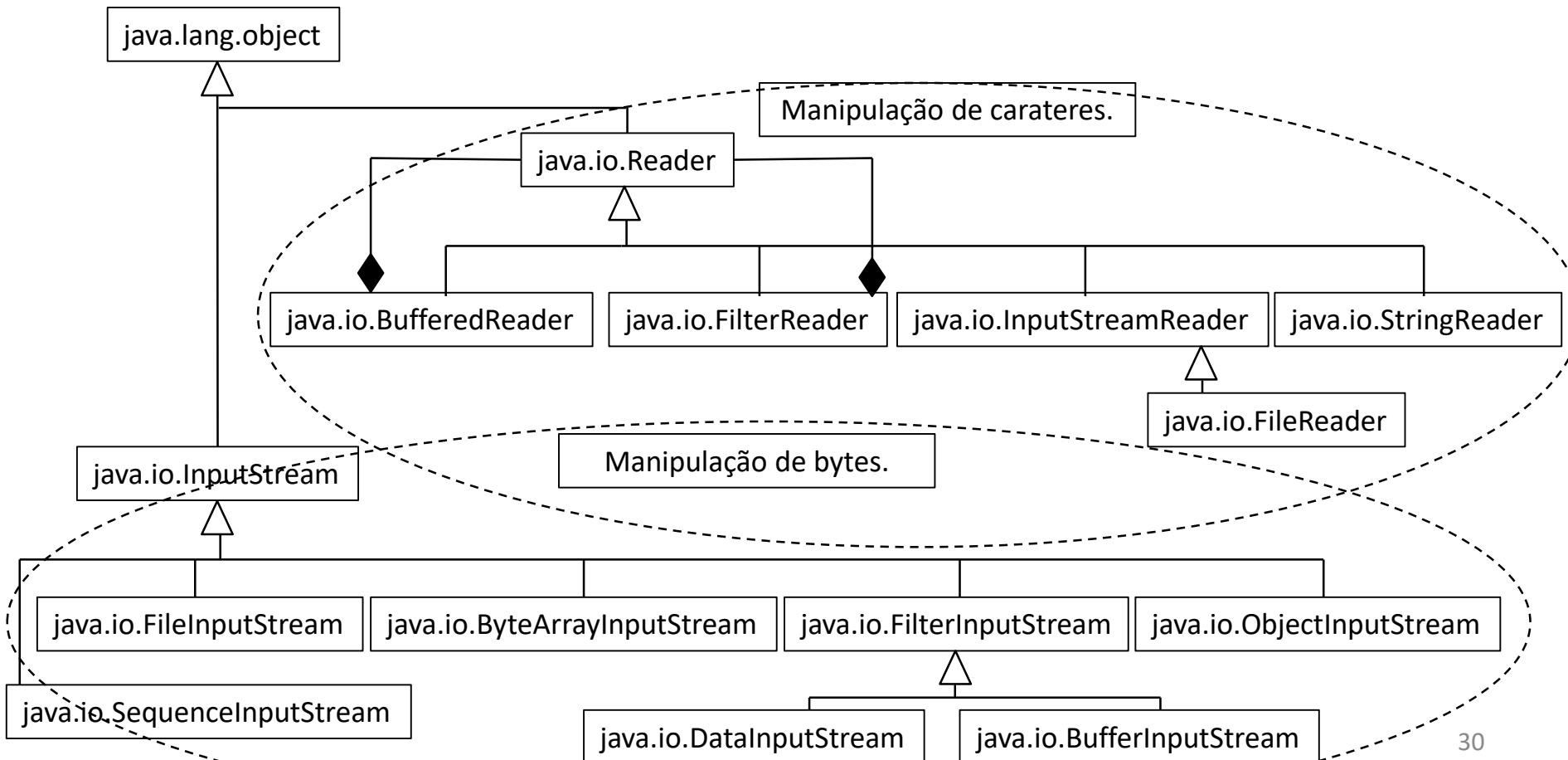


Figura 4.7.1 –Hierarquia de classes para Leitura.

Fundamentos de Sistemas Operativos

A hierarquia de classes para escrita em ficheiros e em dispositivos de entrada/saída está ilustrada na figura 4.7.2.

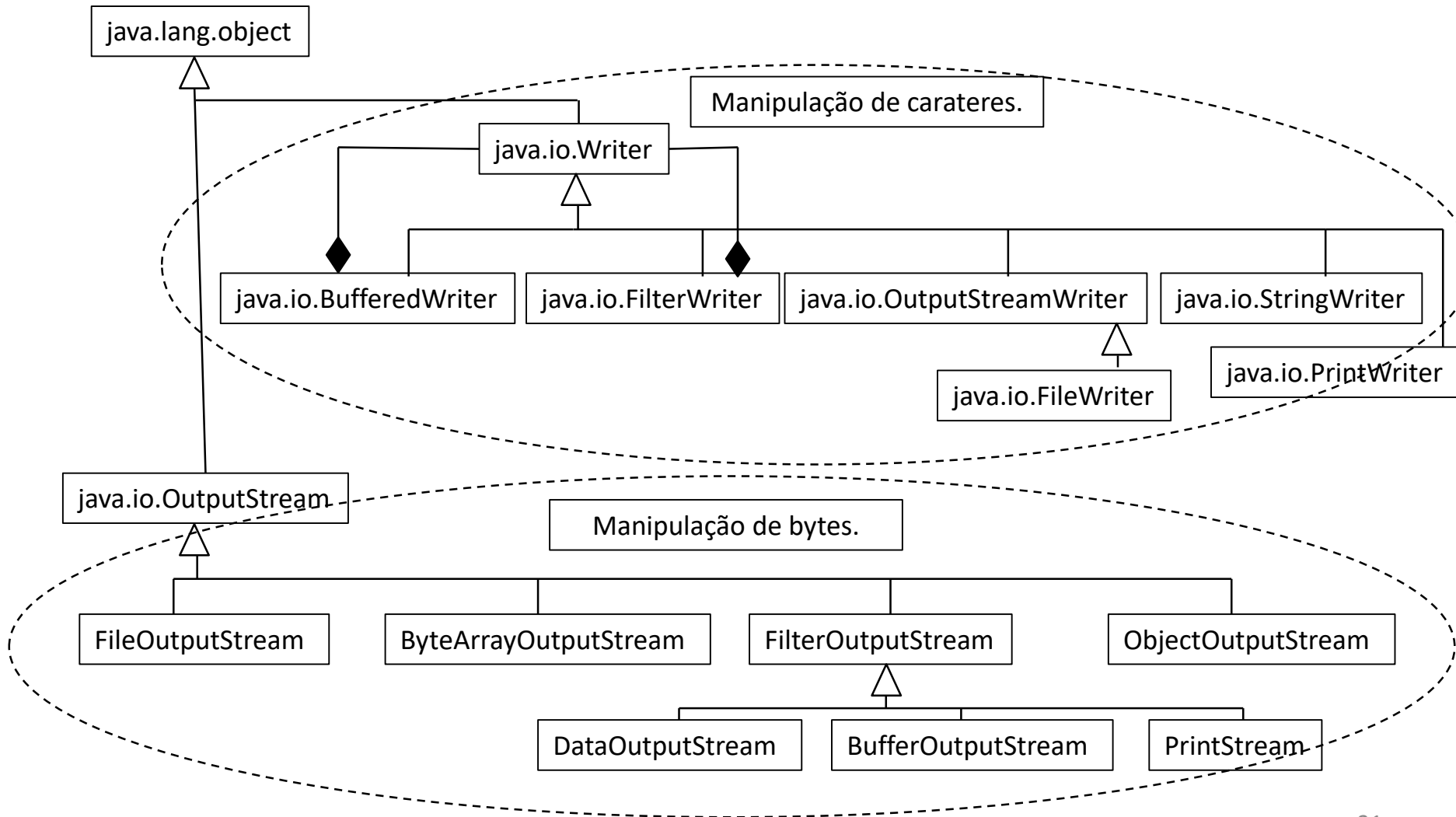


Figura 4.7.2 –Hierarquia de classes para Escrita.

Fundamentos de Sistemas Operativos

Exercício 1: Construa uma classe ListarDiretorias que dispõe dos seguintes métodos:

i) void Dir(String pathname) – se o pathname for uma diretoria deve listar todos os ficheiros dentro da diretoria e referir as subdiretorias. Se o pathname for um ficheiro deve listar o ficheiro e a sua dimensão.

ii) void Dir() – lista os ficheiros da diretoria corrente;

Resolução da alínea 1.i)

```
public class ListarDiretorias {
    File ficheiro;

    void Dir(String nome) {
        ficheiro = new File(nome);
        if (!ficheiro.exists()){
            System.out.println("O " + ficheiro + " não existe.");
            return;
        }
        if (!ficheiro.canRead()) {
            System.out.println("O " + ficheiro + " não aceita leitura.");
            return;
        }
        if (ficheiro.isDirectory()) {
            System.out.println("A diretoria " + ficheiro + " contém:");
            String[] ficheiros = ficheiro.list();
            for (int i = 0; i < ficheiros.length; i++) {
                // concatenar o nome da diretoria ao nome do ficheiro
                File subFicheiro = new File(ficheiro + "\\" + ficheiros[i]);
```

1 de 2

```
        if (subFicheiro.isDirectory())
            System.out.format("%-45s %s \n", ficheiros[i], "[dir]");
        else
            System.out.format("%-50s %-9s %s \n",
                               ficheiros[i], subFicheiro.length(), " bytes");
    }
}
else { // é um ficheiro, mostrar o seu conteúdo
    BufferedReader leitor = null;
    try {
        leitor = new BufferedReader(new FileReader(ficheiro));
    } catch (FileNotFoundException e2) {}
    String line;
    try {
        while ((line = leitor.readLine()) != null)
            System.out.println(line);
    } catch (IOException e) {}
    try {
        leitor.close();
    } catch (IOException e) {}
}
}
```

2 de 2

32

Fundamentos de Sistemas Operativos

Exercício2 : Acrescente à classe ListarDiretorias o seguinte método:

i) void DirAll(String pathname) – se o pathname for uma diretoria deve listar todos os ficheiros dentro da diretoria e os ficheiros dentro das subdiretorias referindo as subdiretorias. Se o pathname for um ficheiro deve listar apenas o ficheiro e a sua dimensão.

Faça a **sua resolução** da alínea 2.i) dentro das caixas.

Fundamentos de Sistemas Operativos

- Anexo 2: Robot Lego-Mindstorms



O robot lego-mindstorms tem uma estrutura modelar, mas a configuração a usar tem duas rodas de tração independentes na frente e uma roda independente e livre na traseira. O robot suporta até 4 sensores e pode ter 3 atuadores podendo ser ligado a um computador através duma ligação USB ou através duma ligação wireless de bluetooth, tal como o ilustrado na figura 1.

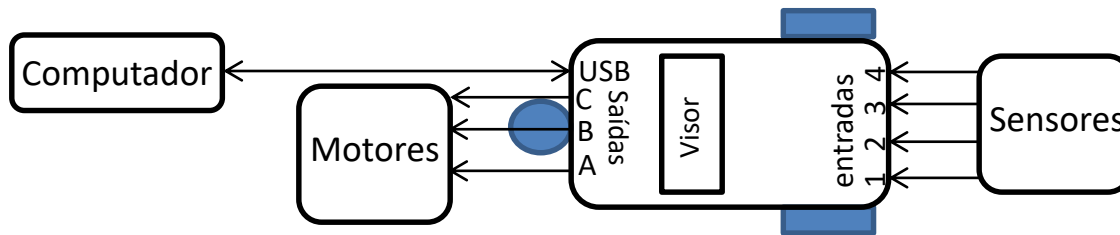


Figura 1: Estrutura do robot Lego-Mindstorms

- Modelo Computador-Robot

O modelo robot-computador está ilustrado na figura 2 e é um modelo utilizado em robots que interagem com humanos e com o mundo. O robot devido à sua limitada capacidade de autonomia das baterias tem uma potência de processamento e memória limitadas integrando apenas algum tipo de inteligência básica à sua sobrevivência e impossibilitando a integração de toda a sua inteligência no robot. Assim, o robot é um escravo do computador tendo autonomia para ações muito básicas de sobrevivência. E o computador funciona como o “cérebro” do robot, comandando remotamente através de bluetooth sempre os seus movimentos e interação com o mundo exterior.

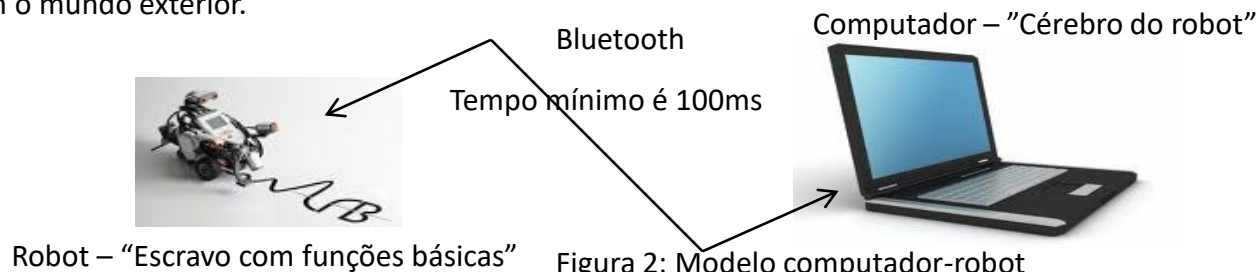


Figura 2: Modelo computador-robot

Fundamentos de Sistemas Operativos

O principal problema do modelo computador-robot da figura 2 é que o cérebro do robot sente ou “vê” o mundo que o rodeia com um atraso equivalente ao tempo da comunicação bluetooth. Por outras palavras, o cérebro sente o mundo não no tempo presente mas num tempo passado. Este problema pode ser minimizado com técnicas de predição em tempo real nas ações.

Os comandos a enviar do computador para o robot são apresentados na seguinte tabela por categorias.

Comunicação	Descrição	Exemplo
<code>boolean OpenNXT(String nomeRobot)</code>	Estabelece o canal de comunicação entre o computador e o robot com o nome passado no parâmetro de entrada do método. O método retorna <i>true</i> em caso de sucesso.	<pre>If (OpenNXT("Guia1")) {} Ou, static final String robot= "Guia2"; If (OpenNXT(robot)) {}</pre>
<code>void CloseNXT()</code>	Fecha o canal de comunicação.	<code>CloseNXT()</code>

Cada robot é identificado por um nome que aparece na primeira linha do visor do robot. Todos os robots têm nomes diferentes.

Depois de uma chamada ao método `OpenNXT` com sucesso, a ligação wireless bluetooth computador-robot fica estabelecida e a partir daqui, o programa em Java está em condições de enviar comandos de controlo para o robot. Para finalizar uma sessão de comunicação deve sempre de chamar o método `CloseNXT`.

Se por qualquer razão o canal de comunicação bluetooth for aberto e não for fechado, apesar do programa Java terminar, o canal de comunicação permanecerá ocupado. Portanto, qualquer tentativa de nova abertura do canal irá dar erro devido à comunicação wireless bluetooth seguir o modelo ponto a ponto.

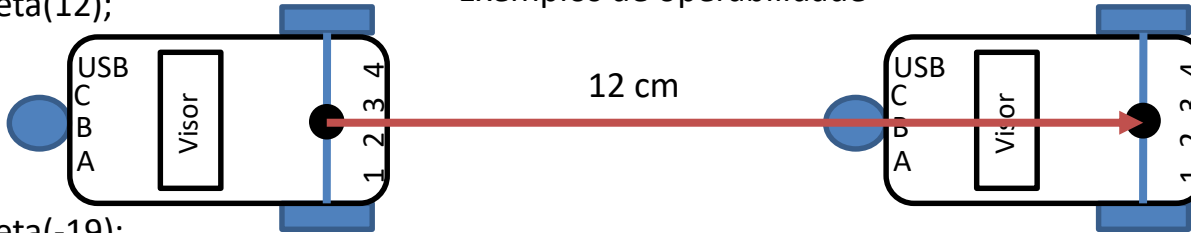
Fundamentos de Sistemas Operativos

Motores	Descrição	Exemplo
<code>void Reta(int distancia)</code>	O robot descreve uma linha reta com a distância dada no parâmetro “distancia” em centímetros. Se o valor da distância for positivo o robot anda em frente, se o valor da distância for negativa o robot anda para trás.	<code>// anda em frente 12 cm Reta(12); // move para trás 19 cm Reta(-19);</code>
<code>void CurvarEsquerda(int raio, int angulo)</code>	O robot curva para a esquerda com o raio dado em centímetros e segundo um dado ângulo em graus.	<code>// curva à esquerda com um raio // de 19cm e um angulo de 90 graus CurvarEsquerda(19, 90);</code>
<code>void CurvarDireita(float raio, int angulo)</code>	O robot curva para a direita com o raio dado em centímetros e segundo um dado ângulo em graus.	<code>// curva à direita com um raio // de 38cm e um ângulo de 45 graus CurvarDireita(38, 45);</code>
<code>void Parar(boolean assincrono)</code>	Pára o robot imediatamente quando o parâmetro de entrada “assincrono” for true. Caso seja false, só pára o robot depois dos outros comandos terem sido executados.	<code>// Pára o robot após o último comando Parar(false); // Pára imediatamente o robot Parar(true);</code>
<code>void AjustarVMD(int offset)</code>	Ajusta a velocidade do motor direito de um valor de velocidade definido em “offset” para que o robot ande a direito quando cumpre o comando Reta().	<code>// Ajusta de +1 a velocidade do // motor direito AjustarVMD(1);</code>
<code>void AjustarVME(int offset)</code>	Ajusta a velocidade do motor esquerdo de um valor de velocidade definido em “offset” para que o robot ande a direito quando cumpre o comando Reta().	<code>// Ajusta de -2 a velocidade do // motor esquerdo AjustarVME(-2);</code>

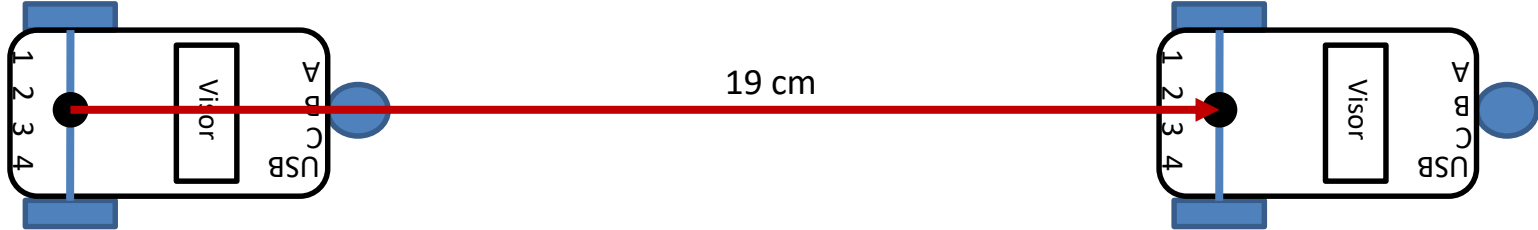
Fundamentos de Sistemas Operativos

- Exemplos de operabilidade

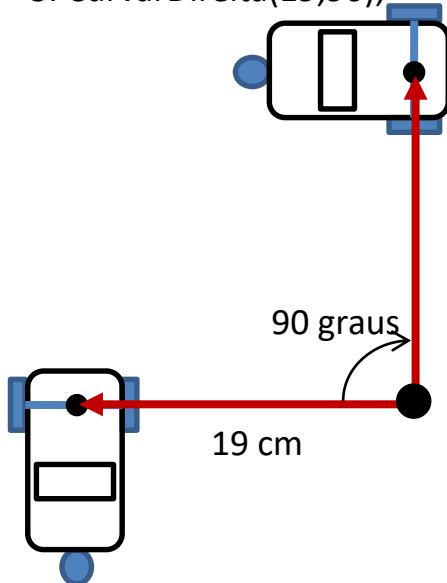
1. Reta(12);



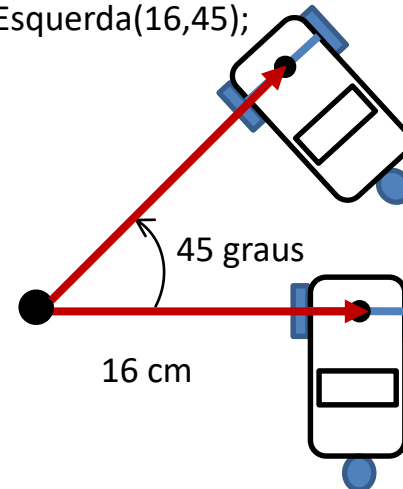
2. Reta(-19);



3. CurvarDireita(19,90);



4. CurvarEsquerda(16,45);



Fundamentos de Sistemas Operativos

Sensores	Descrição	Exemplo
<code>void SetSensorLowSpeed(int input)</code>	Associa um porto de entrada identificado no parâmetro de entrada "input" a um sensor que mede uma distância entre 0 e 255 centímetros utilizando ultrasons. O parâmetro de entrada "input" pode tomar os seguintes valores constantes: S_1, S_2, S_3 ou S_4	// ligar o sensor de distância ao // porto de entrada 4 SetSensorLowSpeed(S_4);
<code>int SensorUS(int input)</code>	Devolve um valor de distância na gama [0, 255] centímetros através da utilização do sensor de ultrasons que está ligado ao parâmetro "input". O parâmetro de entrada "input" pode tomar os seguintes valores constantes: S_1, S_2, S_3 ou S_4	// ler a distância do sensor que está // ligado ao porto 4 int distancia= SensorUs(S_4);
<code>void SetSensorTouch(int input)</code>	Associa um porto de entrada identificado no parâmetro de entrada "input" a um sensor de toque . O parâmetro de entrada "input" pode tomar os seguintes valores constantes: S_1, S_2, S_3 ou S_4	// ligar o sensor de toque ao // porto de entrada 2 SetSensorTouch(S_2);
<code>int Sensor(int input)</code>	Devolve o valor do sensor de toque [0, 1] que está ligado à entrada "input". Devolve 0 se não houver toque e devolve 1 se houver toque. O parâmetro de entrada "input" pode tomar os seguintes valores constantes: S_1, S_2, S_3 ou S_4	// ler o sensor de toque que está // ligado ao porto 2 int toque= Sensor(S_2);

Fundamentos de Sistemas Operativos

Sensores	Descrição	Exemplo
<code>void SetSensorLight(int input)</code>	Associa um porto de entrada identificado no parâmetro de entrada “input” a um sensor de luz. O parâmetro de entrada “input” pode tomar os seguintes valores constantes: S_1, S_2, S_3 ou S_4	// ligar o sensor de distância ao // porto de entrada 4 SetSensorLight(S_2);
<code>void SetSpeed(int percentagem)</code>	O robot cumpre os comandos a 50% da sua velocidade máxima. Com este comando e com o valor do parâmetro “percentagem” que pode ter um valor entre 20 e 100% da velocidade máxima. Os comandos seguintes são cumpridos à velocidade definida por este comando.	// definir o robot para movimentar à velocidade máxima SetSpeed(100);

Fundamentos de Sistemas Operativos

- Bibliografia

Brooks R., Cambrian Intelligence The Early History of the New AI, MIT Press, 1999.

Silberchatz A., Galvin P., Gagne G., Operating System Concepts with Java, John Wiley & Sons, 2009.

Marques J., Ferreira P., Sistemas Operativos, ISBN: 978-972-722-575-5, 2009.

Marques J. A., Guedes P., Fundamentos de Sistemas Operativos, Editorial Presença, 1990.

Lister A. M., Os Sistemas Operativos, Editorial Presença, 1986.