UNIVERSIDADE DE ÉVORA
ESCOLA DAS CIÊNCIAS E TECNOLOGIAS
ENGENHARIA INFORMÁTICA

SISTEMAS OPERATIVOS I

Gestor de Processos

RELATÓRIO

ELABORADO POR:
JACQUELINE ÉVORA Nº25200

JOÃO AIVECA Nº26175

MARLENE OLIVEIRA Nº25999

Ano Lectivo 2011/2012

UNIVERSIDADE DE ÉVORA
ESCOLA DAS CIÊNCIAS E TECNOLOGIAS
ENGENHARIA INFORMÁTICA

OBJECTIVO

Este trabalho tem como objectivo a elaboração de um simulador para a gestão de processos de acordo com o modelo de 5 estados.

A simulação efectuada assemelhar-se-á a um sistema operativo simples que usa o modelo de 5 estados e um sistema com 3 filas para atender os pedidos aos diversos dispositivos de I/O.

Implementar em C um programa que tem como input uma descrição do sistema e de vários processos.

Cada processo alterna períodos (bursts) de uso da CPU com pedidos a dispositivos. Cada processo terminará também num "burst" de CPU. O programa deve gerar um output com o trace dos processos, indicando em cada instante qual o seu estado, e apresentar a relação entre o tempo de CPU e o tempo de espera de cada processo.

UNIVERSIDADE DE ÉVORA
ESCOLA DAS CIÊNCIAS E TECNOLOGIAS
ENGENHARIA INFORMÁTICA

ESTRUTURA DO NOSSO PCB (*Process Control Block*) E DAS NOSSAS FILAS

struct PCB{

int pid -> Id do processo

char estado -> Inicial do estado R-Ready, U - Running (porque Ready tem a mesma inicial),...

int t_wait -> Tempo em espera

int t_cpu -> Tempo do processo em CPU até ser interrompido por quantum.

int t_cpu_total -> Tempo do processo em CPU desse tempo de CPU, contando todos os quantums.

int t_cpu_somado -> Tempo do processo em todas as utilizações do CPU

int current -> Índice dos tempos que indica qual está a ser processado actualmente, seja de CPU ou num dispositivo.

int timeInDisp -> Tempo decorrido num dispositivo

int instI -> Instante de entrada para new

int * tempos -> Tempos de execução segundo o input

int maxT -> Número de valores no array de tempos

};

struct fila{

struct PCB *p -> Elemento que se encontra à cabeça da fila.

struct fila *next -> Apontador para o próximo elemento.

int size -> Inteiro correspondente ao tamanho da fila.

int tdisp -> Tempo do dispositivo (inútil no caso em que a fila corresponde a um estado.

};

Funções que permitem a manipulação das Filas:

- **void initUSERDISP(struct fila * fi, int tdisp)** -> Inicializa uma fila de dispositivos com os valores default.
- **struct PCB* removeUSER (struct fila * fi, int tdisp)** -> Remove um elemento de uma fila (o inteiro serve para propagar o tempo do dispositivo, no caso da fila corresponder a um dispositivo).
- **void initUSER(struct fila * fi)** -> Inicializa uma fila (que neste caso, corresponderá a um estado) com os valores default.
- **void insertFIFO (struct PCB *p, struct fila * fi, int tdisp)** -> Insere processos numa fila.
- **int isEmpty(struct fila *fi)** -> Verifica se uma fila se encontra vazia.
- **struct PCB* getFirst(struct fila *f)** - > Obtém o elemento que se encontra no topo da fila.
- **int size (struct fila *fi)** -> Devolve o tamanho da fila.

Funções do Ciclo Principal:

- **main(int argc, char * argv[])** -> Método main. É neste método que é lido o input do utilizador.
- **void mainCycle(int nprocessos, int * processos [])** -> Função principal do código. Inicializa um array de apontadores para processos que são usados para prints – tal impede a necessidade de os procurar fila a fila. O estado 'x' é usado neste array, que indica que o processo ainda não esteve em new. Ao ser inicializado, um processo passa para a fila de new e é guardado no array. Então, executam-se as tarefas necessárias: as rotinas dos dispositivos (para que o tempo avance nos mesmos), seguidas da passagem de processos em new para ready; finalmente, corremos o scheduler para saber se algum processo teve timeout ou terminou. Os prints são efectuados neste ponto, o que implica que para cada unidade de tempo i, o mostrado no output é i-1, ou seja, a operação que foi executada neste ciclo. Após estas operações, corremos (com run()) no CPU 1 unidade de tempo. Terminamos se todos os processos, neste ponto, estão em exit.
- **void scheduler()** -> Função que trata da admissão de processos no estado Running.
- **void newToReady()** -> Equivalente à transição ADMIT do modelo de cinco estados. Permite a passagem de processos da fila NEW para a fila READY (tal admissão só é possível se o número de processos na fila READY for menor ou igual a dois).
- **void blockedToReady(int disp)** -> Equivalente à transição EVENT OCCURS do modelo de cinco estados. Permite a passagem de processos da fila BLOCKED para a fila READY, depois dos dispositivos terem sido verificados por ordem.
- **void runningToReady()** -> Equivalente à transição TIMEOUT do modelo de cinco estados. Permite a passagem de processos da fila RUNNING para a fila READY.
- **void runningToBlocked()** -> Equivalente à transição EVENT WAIT do modelo de cinco estados. Permite a passagem de processos da fila RUNNING para a fila BLOCKED (tal ocorre quando existem chamadas de IO).
- **void runningToExit()** -> Equivalente à transição RELEASE do modelo de cinco estados. Permite a passagem de processos da fila RUNNING para a fila EXIT.
- **void readyToRunning()** –> Equivalente à transição DISPATCH do modelo de cinco estados. Permite a passagem de processos da fila READY para a fila RUNNING.
- **struct PCB * newProcess(int pID, int * tempos)** -> Cria um processo.
- **void terminate(struct PCB * p [])** -> Termina o programa quando todos os processos já tiverem concluído a sua execução.
- **void printProcesses(struct PCB * p [], int nproc)** -> Imprime output.

- **void run() ->** Executa um burst de CPU (1 unidade).
- **void disp(struct PCB * p, int disp)** -> Permite efectuar a gestão de processos nas filas de dispositivos.
- **int * toArrayInt(char * values, int * convertedR)** -> Converte o input dado pelo utilizador (que inicialmente é guardado numa string) para um array de inteiros.

Output do Algoritmo:

```
<0> <u> <x> <x>
<1> <u> <x> <x>
<2> <u> <r> <x>
<3> <u> <r> <r>
<4> <r> <u> <r>
<5> <r> <u> <r>
<6> <r> <b> <u>
<7> <r> <b> <u>
<8> <r> <b> <u>
<9> <r> <b> <u>
<10> <u> <b> <r>
<11> <b> <r> <u>
<12> <b> <r> <u>
<13> <r> <r> <u>
<14> <r> <u> <b>
<15> <r> <u> <b>
<16> <r> <u> <b>
<17> <u> <b> <b>
<18> <b> <b> <b>
<19> <b> <b> <u>
<20> <b> <b> <u>
<21> <b> <b> <e>
<22> <b> <b> <e>
<23> <b> <b> <e>
<24> <b> <u> <e>
<25> <b> <u> <e>
<26> <b> <u> <e>
<27> <b> <u> <e>
<28> <b> <u> <e>
<29> <b> <b> <e>
<30> <b> <b> <e>
<31> <u> <b> <e>
<32> <e> <b> <e>
<33> <e> <b> <e>
<34> <e> <u> <e>
<35> <e> <u> <e>
<36> <e> <u> <e>
<37> <e> <u> <e>
<38> <e> <e> <e>

<1.285714>

<1.214286>

<0.555556>

Processos terminados.
```

CONCLUSÃO

Dado ter-mos um input fixo, nós tiramos partido deste facto. A implementação do nosso algoritmo poderia ser alterada de forma a aceitar inputs mais variados; por exemplo, já que o enunciado garante que são 3 os dispositivos, temos sempre 3 filas inicializadas. Tal podia ser resolvido (já que o formato do input compreende o número de dispositivos) para um array blocked de filas, com cada índice do array a corresponder a um dispositivo.

Como tal não se verifica, o algoritmo corre apenas para 3 dispositivos. Outro problema prende-se com o malloc limitado a 3 caracteres no toArrayInt; isto implica que inputs maiores que 999 num dos tempos podem causar problemas. Tal podia ser resolvido com mallocs/frees que alocassem memória de forma dinâmica.

O input poderia também ser lido de um ficheiro.

Além destas melhorias, julgamos que a nossa implementação é suficiente para realizar a gestão dos processos, tal como sugerida no enunciado.

Anexos

Código do ficheiro trabalho.h:

```
typedef PCB;

typedef fila;


struct PCB{

        int pid; //id do processo

        int pc;

        char estado; //inicial do estado R-Ready, U - Running (porque Ready tem a mesma inicial),...

        int t_wait; //tempo em espera

        int t_cpu; //tempo do processo em CPU até ser interrompido por quantum.

        int t_cpu_total; //tempo do processo em CPU desse tempo de CPU, contando todos os quantums.

        int t_cpu_somado; //tempo do processo em todas as utilizações do CPU

        int current; //índice dos tempos que indica qual está a ser processado actualmente, seja de CPU ou num dispositivo.

        int timeInDisp; //tempo decorrido num dispositivo

        int instI; //instante de entrada para new

        int * tempos; //tempos de execução segundo o input

        int maxT; //número de valores no array de tempos


};

struct fila{

        struct PCB *p;

        struct fila *next;

        int size;

        int tdisp;

};
```

Código do ficheiro trabalho.c:

```
#include "trabalho.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 2


struct fila *new;

struct fila *ready;

struct fila *running;

struct fila *exitU;


struct fila *d1;

struct fila *d2;

struct fila *d3;


void initUSERDISP(struct fila * fi, int tdisp){ //Inicia uma fila de dispositivos com os valores default.

	fi->size=0;

	fi->p=NULL;

	fi->next=NULL;

	fi->tdisp=tdisp;

}

struct PCB* removeUSER (struct fila * fi, int tdisp){

	struct PCB *v; //elemento que vai ser removido. Este temporário é para o guardar, para retornar.

	if(isEmpty(fi)){

		return NULL; //não é possível remover duma fila vazia!
```

```
	}

	else{

		v=fi->p; //elemento a remover

		fi->size--; //fila decrementa o tamanho, já que perdeu 1 elemento

		if(tdisp!=-1) //se fôr fila de dispositivos, tem de manter o tempo de dispositivo guardado à cabeça.

			fi->next->tdisp=tdisp; //coloca o tempo de dispositivo na nova cabeça

		*fi = *fi->next; //Avança na fila, ficando a nova cabeça o elemento seguinte, efectivamente perdendo-se

					//a antiga cabeça, sendo "removida".


		return v;

	}

}


void initUSER(struct fila * fi){ //inicializa uma fila (não de dispositivos) com valores default

	fi->size=0;

	fi->p=NULL;

	fi->next=NULL;

}


void insertFIFO (struct PCB *p, struct fila * fi, int tdisp){ //insere o processo p na fila fi


	while( fi->next )

		fi=fi->next; //procura a cauda da fila para garantir comportamento FIFO


	fi->p=p; //coloca o novo elemento na cauda fa fila antiga
```

```
	struct fila * cauda=(struct fila*) malloc(sizeof(struct fila)); //cria a nova cauda sem elemento

	if(tdisp!=-1) //se for fila de dispositivos, propaga o tempo do dispositivo à cauda da nova fila

		initUSERDISP(cauda, tdisp);

	else

		initUSER(cauda);

	fi->next=cauda; //coloca a cauda da fila ligada ao elemento novo inserido

}

int isEmpty(struct fila *fi){ //se fi está vazia retorna 1, else retorna 0

	if((fi->p==NULL)&&(fi->next==NULL)){

		return 1;

	}

	return 0;

}


struct PCB* getFirst(struct fila *f){ //obtém o topo da fila

	return f->p;

}

int size ( struct fila *fi){//devolve o tamanho da fila


	int i = 0;

	struct fila *temp=fi;

	while(!isEmpty(temp)){

		i++;

		temp=temp->next;}

	return i;

}
```

```
void newToReady(){
	/*ADMIT*/
	if(ready->size < MAX || ready->size == MAX){
		struct PCB *temp = removeUSER(new,-1);
		temp->estado = 'r';
		temp->t_cpu=0;
		insertFIFO(temp,ready, -1);
	}
	else{
		printf("Não é possível admitir novos processos na fila ready.\n");
	}
}

void blockedToReady(int disp){

	/*Event Occurs*/

	struct PCB *temp;
	if(disp==1)
		temp = removeUSER(d1, d1->tdisp);
	if(disp==2)
		temp = removeUSER(d2, d2->tdisp);
	if(disp==3)
		temp = removeUSER(d3, d3->tdisp);

	temp->estado = 'r';
```

```
	insertFIFO(temp,ready, -1);


}


void runningToReady(){


	/*Timeout*/

	struct PCB *temp = removeUSER(running, -1);

	temp->estado = 'r';

	insertFIFO(temp,ready,-1);


}



void runningToBlocked(){


	/*Event Wait*/


	struct PCB *temp = removeUSER(running, -1);

	temp->estado = 'b';

	if(temp->tempos[temp->current]==1)

		insertFIFO(temp,d1, d1->tdisp);

	if(temp->tempos[temp->current]==2)

		insertFIFO(temp,d2, d2->tdisp);

	if(temp->tempos[temp->current]==3)

		insertFIFO(temp,d3,d3->tdisp);

	temp->current++; //salta
```

```
}




void runningToExit(){


        /*Release*/

        struct PCB *temp = removeUSER(running,-1);

        temp->estado = 'e';

        insertFIFO(temp,exitU, -1);


}


void readyToRunning(){


        /*Dispatch*/


        struct PCB *temp = removeUSER(ready,-1);

        temp->estado = 'u';

        insertFIFO(temp,running, -1);


}


struct PCB * newProcess(int pID, int * tempos){ //cria um processo

        struct PCB *p = malloc(sizeof(struct PCB));

        p->pid =pID;

        p->timeInDisp=0;
```

```
	p->instI = tempos[0];

	p->estado = 'n';

	p->t_cpu=0;

	p->t_cpu_total=0;

	p->t_cpu_somado=0;

	p->tempos=tempos;

	int count=0;

	while(tempos[count]!=-1)

		count++;

	p->maxT=count-1;

	p->current=1; //SALTA VALOR DE INICIALIZAÇÃO (índice 0)

	return p;

}


void terminate(struct PCB * p []){ //Termina

	printf("\n<%f> \n<%f> \n<%f> \n",(float)p[0]->t_wait/(float)p[0]->t_cpu_somado,

					(float)p[1]->t_wait/(float)p[1]->t_cpu_somado,

					(float)p[2]->t_wait/(float)p[2]->t_cpu_somado);

	printf("Processos terminados.\n");

	exit(1);

}


void printProcesses(struct PCB * p [], int nproc){ //imprime output

	int i;

	for(i=0; i<nproc; i++){

		printf("<%c> ", p[i]->estado);

		}
```

```
	puts("\n");

	}


void run(){ //executa um burst de CPU (1 unidade)

	if(isEmpty(running))

		return;

	struct PCB *temp = getFirst(running);




	temp->t_cpu++;

	temp->t_cpu_total++;

	temp->t_cpu_somado++;


}

void scheduler(){

	if(isEmpty(running))

		return;

	struct PCB *temp = getFirst(running);

	if(temp->t_cpu_total==temp->tempos[temp->current])

		//processo já terminou o CPU actual

		if(temp->current==temp->maxT){

			//sendo o último, termina

			runningToExit();

			temp->t_cpu=0;

			temp->t_cpu_total=0;

			if(!isEmpty(ready))

				readyToRunning();
```

```
                return;}

        else{

                //não sendo, passa a blocked aguardando evento

                temp->current++;

                runningToBlocked();

                temp->t_cpu=0;

                temp->t_cpu_total=0;

                if(!isEmpty(ready))

                        readyToRunning();

                return;

                }

    if(temp->t_cpu%4==0&&temp->t_cpu!=0){

                //verifica quantum

                temp->t_cpu=0;

                if(!isEmpty(running))

                        runningToReady();

                if(!isEmpty(ready))

                        readyToRunning();

                return;

                }

}


void disp(struct PCB * p, int disp){

    p->timeInDisp++;

    p->t_wait++;

    //avança tempo nos dispositivos

    if(disp==1)
```

```
                if(p->timeInDisp==d1->tdisp){

                        p->timeInDisp=0;

                        blockedToReady(1);}


        if(disp==2)

                if(p->timeInDisp==d2->tdisp){

                        p->timeInDisp=0;

                        blockedToReady(2);}


        if(disp==3)

                if(p->timeInDisp==d3->tdisp){

                        p->timeInDisp=0;

                        blockedToReady(3);}
}


void mainCycle(int nprocessos, int * processos []){

        int i=0;

        int pID=0; //incrementa-se para garantir pids únicos

        struct PCB * prcPrint [nprocessos]; //para efeitos de print

        prcPrint[0]->estado='x'; //default para não inicializado (ainda não entrou na fila new)

        prcPrint[1]->estado='x';

        prcPrint[2]->estado='x';

        while(exitU->size < nprocessos && i<80){

                int j = 0;

                for(j=0; j<nprocessos; j++)

                        if(processos[j][0]==i){

                                //se o tempo actual do ciclo for o de inicialização do processo,
passa-o para a fila de new
```

```
                prcPrint[j]=newProcess(pID, processos[j]);

                insertFIFO(prcPrint[j],new, -1);

                pID++;}


i++;


/*1 - filas de dispositivos para ready*/

if(!isEmpty(d1)){

        disp(getFirst(d1), 1);}

if(!isEmpty(d2))

        disp(getFirst(d2), 2);

if(!isEmpty(d3))

        disp(getFirst(d3), 3);


/*2 - fila de new para ready*/

if(!isEmpty(new))

        newToReady();


/*3 - fila de CPU para ready*/

if(isEmpty(running))

        if(!isEmpty(ready))

                readyToRunning();



//verifica o processo em running

scheduler();

printf("<%d> ",i-1);
```

```
                int k=0;
                for(k=0; k<nprocessos; k++){
                        printf("<%c> ", prcPrint[k]->estado);
                }
                puts("\n");
                //executa 1 unidade de tempo no CPU
                run();
                //termina se exit tem todos os processos
                if(size(exitU)==nprocessos) {
                        terminate(prcPrint);
                        }


        }
}

int * toArrayInt(char * values, int * convertedR ){
        int count=0; //contar posição actual da string
        int nints=0; //contar inteiros convertidos
        convertedR=malloc(sizeof(int)*strlen(values));
        char * t = malloc(sizeof(char)*3); //temporário
        while(count<strlen(values)){
                int t2=0;
                int i=0;
                while( values[count]!=' '&&values[count]!='\0'){
                        t[i]=values[count];
                        count++;} //lê string
```

```
                count++; //salta espaço

                convertedR[nints]=atoi(t); //converte

                nints++;

                t=malloc(sizeof(char)*3); //realoca temporário

                }

        convertedR[nints]=-1; //termina os convertidos com -1 para indicar o fim do array

        return convertedR;



}


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

        //garante input certo

        if(argc<=1){

                printf("Erro de chamada da função.");

                exit(1);}

        //leitura do input geral

        int nprocessos;

        int ndispositivos;

        int temp=0;

        int * input;


        if(argc>=2){

                //transformar string input em valores


                input=toArrayInt(argv[1], input);

                nprocessos=input[0];
```

```
		ndispositivos=input[1];

	}


	//garante veracidade do input

	if(argc!=(2+nprocessos)){


		printf("Falha: nprocessos não corresponde ao número de processos dos argumentos.");

		exit(1);


	}


	//inicializa as filas (vazias)

	new=malloc(sizeof(struct fila));

	ready=malloc(sizeof(struct fila));

	running=malloc(sizeof(struct fila));

	exitU=malloc(sizeof(struct fila));

	d1=malloc(sizeof(struct fila));

	d2=malloc(sizeof(struct fila));

	d3=malloc(sizeof(struct fila));


	initUSER(new);

	initUSER(ready);

	initUSER(running);

	initUSER(d1);

	initUSER(d2);

	initUSER(d3);

	initUSER(exitU);
```

```
	d1->tdisp=input[2];

	d2->tdisp=input[3];

	d3->tdisp=input[4];	//tempos dos dispositivos


//ler inputs seguintes, dados por processo.


	int currentProcess=0;

	int * tempos [nprocessos];

	while(currentProcess<nprocessos){

		int i;

		int * te=toArrayInt(argv[2+currentProcess],te);


		tempos[currentProcess]=te;

		currentProcess++;

		}



/*executa os processos*/

	mainCycle(nprocessos,tempos);

}
```