IMPLEMENTAÇÃO

Para início deste relatório, vale dizer que com a codificação foi divida em etapas, dando uma noção do quanto a implementação estava avançando e precavendo que etapas mais avançadas não rodassem por causa de etapas mais básicas.

Um fato a ser destacado é que, na mesma ordem que está escrita, é que a etapa seguinte conserva a lógica das etapas anteriores.

Então as etapas ficaram:

- Ler o arquivo;
- Passar o arquivo como input pelo terminal (ARGC e ARGV);
- Salvar as tarefas em Struct's;
- Criar uma lista ligada com as as Struct's;
- Ordenar fila encadeada;
- Criar lista(array) com os tempos em que as tarefas aparecerão;
- Rate propriamente dito;
- Escrever o resultado em um arquivo;

Agora sabendo quais são, pode-se dar início ao aprofundamento das etapas:

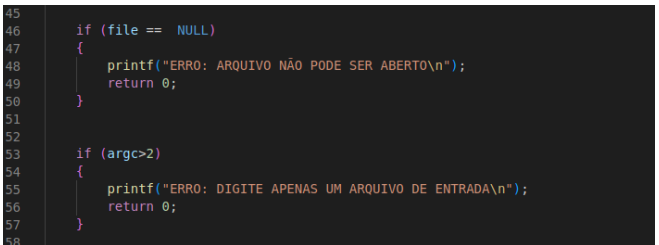
Ler O Arquivo

Primeiramente foi pego o arquivo "input.txt" de exemplo disponibilizado pelo professor Erico para a realização da atividade. Estando este arquivo já dentro da mesma pasta do programa, foi criado um ponteiro de nome "file" do tipo FILE e utilizando a função da biblioteca <stdio.h> "fopen()", que retorna um ponteiro para o file, e como segundo argumento o char "r", indicando que o arquivo deseja ser lido, assim o arquivo "input.txt" pôde ser aberto.

Para a leitura propriamente dita do arquivo, foi utilizada a função "fscanf()", que tem como primeiro argumento o ponteiro tipo FILE, que indica qual arquivo deseja-se ler. Salvando o [TOTAL TIME] em uma variável tipo int chamada "tempo total". Depois fazendo um loop "for" que roda até o final do arquivo EOF, foi possível pegar o [TASK NAME] [PERIOD] [CPU BURST].

Passar o arquivo como input pelo terminal (ARGC e ARGV)

A primeira coisa a ser feita foi o tratamento de erro, caso o arquivo fosse inexistente ou caso fosse escrito mais de um arquivo na hora de rodar no terminal. Respectivamente nos "if" a seguir:



O Arquivo passado foi lido da seguinte forma, sendo representado arvg[1], e a lógica anterior foi mantida.

```
43
44      file = fopen(argv[1], "r");
45
```

Salvar as tarefas em Struct's;

Foi criada uma struct com os seguintes valores:

```
typedef struct node
{
    char nome[20];
    int periodo;
    int burstOriginal;
    int burstAtual;
    int countDeadLine;
    int countExecution;
    int countKilled;
    int flag;
    struct node *prox;
} node;
```

A struct foi declara antes da Main() e como foi usado o typedef, "node" virou variável global;

E a partir de uma nova função insert() as struct's os valores dentro da Struct estavam conseguindo ser salvos corretamente, sendo nome, período e burstOriginal respectivamente os [TASK NAME] [PERIOD] [CPU BURST]. Os demais valores foram criados pensando no resto da execução do código e serão usados nas próximas etapas.

Criar uma lista ligada com as as Struct's;

Usando um ponteiro head tipo node e o ponteiro "prox" dentro das struct's, foi criada uma lista ligada (ainda dentro da função insert())

```
186     node *nova, *aux;
187     aux = (node *)malloc(sizeof(node));
188     nova = (node *)malloc(sizeof(node));
189     strcpy(nova->nome, neWnome);
190     nova->periodo = neWperiodo;
191     nova->burstOriginal = neWburstOriginal;
192     nova->burstAtual = 0;
193     nova->countDeadLine = 0;
194     nova->countExecution = 0;
195     nova->countKilled = 0;
196     nova->flag = 0;
197
```

Como o ponteiro head é um tipo node e uma variável universal, a função insert() nao precisa retornar nada, sendo do tipo void.

Ordenar fila encadeada

Depois de criada com sucesso, era necessário ordenar a fila, pelo período, criando assim a prioridade solicitada pelo algoritmo, então a ordem foi imposta logo assim que a fila foi criada (ainda dentro da função insert()):

```
197
198     if (p->prox == NULL)
199     {
200         nova->prox = p->prox;
201         p->prox = nova;
202     }
203     else if (nova->periodo < p->prox->periodo)
204     {
205         nova->prox = p->prox;
206         p->prox = nova;
207     }
208     else
209     {
210         aux = p->prox;
211         while (aux->prox != NULL && nova->periodo > aux->prox->periodo)
212         {
213             aux = aux->prox;
214         }
215         nova->prox = aux->prox;
216         aux->prox = nova;
217     }
218     tamanho += time / neWperiodo;
219 }
```

Verificando se era o primeiro da fila – primeiro if – era apenas adicionado normalmente na fila. Caso não fosse o primeiro nó adicionado, verificava-se se ele seria o primeiro da fila (se tivesse o menor período) – segunda condicional (primeiro “else if”). Caso não fosse nem o primeiro nem o menor, a lista era percorrida, com a ajuda de um ponteiro aux tipo node(aux de auxiliar) até que o período do novo nó a ser alocado fosse menor que o período da fila próx.

Uma nota importante é que a função imprima() serve apenas para com debug do código para verificar se até esta etapa estava funcionando.

Criar lista(array) com os tempos em que as tarefas aparecerão

Para rodar o código de rate foi criada uma lista de inteiro com todos os múltiplos dos períodos de todas as tarefas, sem repetição caso períodos diferentes de tarefas diferentes tivessem o mesmo múltiplo, incluindo o zero e o tempo total. Isso foi feito com a função addmdc() que retorna um ponteiro para um inteiro.

```
240
241     for (p = le->prox; p != NULL; p = p->prox)
242     {
243         int N = time / p->periodo;
244 
245         for (int i = 0; i < N + 1; i++)
246         {
247             for (int j = 0; j < length; j++)
248             {
249                 if (m[j] == p->periodo * i)
250                 {
251                     flag = 1;
252                 }
253             }
254             if (flag == 0)
255             {
256                 m[in] = p->periodo * i;
257                 in++;
258             }
259             flag = 0;
260         }
261     }
262     m[in] = time;
```

O inteiro N vê quantas múltiplos do período cabem no array até o tempo total;

O inteiro “in” marca quantos multiplos já entraram, dessa forma nenhum será sobrescrito e ele adiciona o tempo total (representado por time) no último idex do array, por isso que “tamanho” se inicializa com 1 e não com zero (representado por length dentro da função addmdc());

O primeiro for na linha 241 serve para rodar a lista encadeada e pegar os períodos de todas as tarefas;

O segundo “for” da linha 245 serve para adicionar os múltiplos do período da tarefa da vez até o tempo total;

O terceiro “for” da linha 247 verifica se o multiplos já foi adicionado, caso tenha sido ele coloca a variável flag do tipo inteiro como 1;

O múltiplo só é adicionado no array caso a flag seja exatamente igual a 0;

Depois de criada, foi usada outra função, bubbleSort, para ordenar esse array;

Rate propriamente dito;

```
91      for (int i = 0; i < tamanho - 1; i++)
92      {
93          int dif = m[i + 1] - m[i];
94          int all;
95          int try = t - m[i];
96          int try1;
97          for (p = le->prox; p != NULL; p = p->prox)
```

Toda essa etapa ocorre dentro da função rate();

O “for” da linha 91 vai rodar o array dos múltiplos, que está representando os tempo em que as tarefas surgiram novamente;

A variável “dif” do tipo inteiro representa a diferença entre o tempo(múltiplo) atual e o próximo, dizendo quantas unidades de execução(u.e) se tem até que seja necessário verificar se uma tarefa de maior prioridade entrou.

```
97          for (p = le->prox; p != NULL; p = p->prox)
98          {
99              if (m[i] % p->periodo == 0)
100             {
101
102                 p->burstAtual = p->burstOriginal;
103             }
104
105             if (try < p->burstAtual && try > try1)
106             {
107                 can = FALSE;
108                 try1=try;
109             }
```

O “for” da linha 97 ira rodar as tarefas que estão representadas pelos nós da lista encadeada;

O “if” da linha 99 verifica se tempo atual (o múltiplo do array) é múltiplo do período da tarefa atual, caso seja a variável burstAtual do tipo inteiro do nó no momento, reinicia seu valor para a sua outra variável burstOriginal;

O “if” da linha 105 verifica se ainda da pra executar a tarefa atual, pegando a diferença entre o tempo(múltiplo) atual e o tempo total), caso ele não consiga, a variavel can do tipo inteiro terá o valor FALSE (que foi definido no inicio do codigo como 0). Por questões de execução para saber o quanto de u.e foi perdido na hora que o tempo total acabou, foram criadas as variáveis try e try1, que guardam o maior valor que eles não conseguirão rodar. Tem que ser o maior valor para que no final só seja printado uma vez o que a tarefa não conseguiu executar.

CONTINUA NA PROXIMA PÁGINA…

```
110
111         if (p->burstAtual > 0 && dif > 0)
112         {
113             if (dif > p->burstAtual)
114             {
115                 temp = dif;
116                 dif -= p->burstAtual;
117                 exec = temp - dif;
118                 p->burstAtual = p->burstAtual - exec;
119             }
120             else if (dif <= p->burstAtual)
121             {
122                 temp = p->burstAtual;
123                 p->burstAtual -= dif;
124                 exec = temp - p->burstAtual;
125                 dif = dif - exec;
126             }
127
128             for (node *q = le->prox; q != NULL; q = q->prox)
129             {
130                 all = TRUE;
131                 if (q->burstAtual > 0) all = FALSE;
132             }
```

O “if” da linha 111 verifica se o período atual tem alguma u.e para se executada e se há tempo (representado por “dif”) para executar algo dessa tarefa;

If linha 115 e if linha 120 – dependendo de quem tiver o maior valor (dif ou o burstAtual), esses if’s irão retirar de maneira correta o valor que pôde ser executado;

For linha 128 – vai passar por todos os nós da lista encadeada e verificar se todos os valores, após a execução e a mudança do busrtAtual, são iguais a zero, caso sejam a variável all do tipo int será TRUE, que foi definido como um tipo int de valor 1;

```
134         if (p->burstAtual == 0)
135         {
136             p->countExecution++;
137             fprintf(saida,"[%s] for %d units - F\n", p->nome, exec);
138             if(all==TRUE && p->prox == NULL)fprintf(saida,"idle for %d units\n",  dif);
139         }
140         else if (p->burstAtual > 0 && m[i+1] % p->periodo != 0)
141         {
142             if(can==FALSE)acul+= exec;
143             if (can)fprintf(saida,"[%s] for %d units - H\n", p->nome, exec);
144             else if (acul == try1)fprintf(saida,"[%s] for %d units - K\n", p->nome, acul);
145
146         }
147         else if (m[i+1] % p->periodo == 0)
148         {
149             fprintf(saida,"[%s] for %d units - L\n",  p->nome, exec);
150             p->countDeadLine++;
151         }
152
```

If linha 134 – verifica se após a execução o nó atual da lista, que representa uma tarefa, tem o busrtAtual igual a zero, caso tenha ele adiciona mais 1 no contador countExecution daquele nó e printa uma saída tipo Finished. Apenas com todos os bursts

iguais a zero é possível que o programa passe alguma u.e sem realizar nada(idle), por isso que o print de saída tipo idle está dentro deste if. Além disso, ele está acompanhado de outro if do início da linha 138, que entra caso a variável all seja true e caso a tarefa seja última a da lista, pois só aí se tem certeza que o programa não vai executar mais nenhuma u.e.

Else if linha 140 – esse if entra caso o ainda falte alguma u.e (burstAtual) a ser executada, e printa saída tipo Hold dizendo o quantas u.e. foram executadas. Caso não dê para executar mais a tarefa, ele começa a acumular o quanto foi executado na variável acul para printar a saída killed de uma vez só. Primeiro ele verifica se can é FALSE, pois se for, significa que ele não está em Hold (esperando para terminar de executar) e printa a saída Hold. Caso Acul seja igual a maior diferença que não possa executar (try1), ele printa a saída Killed;

Else if linha 147 – se o próximo tempo(múltiplo) da lista é múltiplo do período da tarefa atual, pois se for, e ele não tiver executado tudo – já que não entrou no primeiro if – ele terá a saída tipo Lost e o countDeadline do nó;

Escrever o resultado em um arquivo;

Foi criada um ponteiro saída tipo file, e a partir do fopen() com um char “w” e usando a função fprintf();

Todos os requisitos foram entregues.