



Trabalho Prático II (TP II) - Filas e Métodos de Ordenação

- Submissão com data e hora de entrega disponíveis na plataforma da disciplina. O que vale é o horário do Moodle, e não do *seu*, ou do *meu* relógio!!!
- Clareza, identificação e comentários no código também vão valer pontos. Por isso, escolha cuidadosamente o nome das variáveis e torne o código o mais legível possível.
- O padrão de entrada e saída deve ser respeitado exatamente como determinado no enunciado. Parte da correção é automática, não respeitar as instruções enunciadas pode acarretar em perda de pontos.
- Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.
- A avaliação considerará o tempo de execução e o percentual de respostas corretas.
- Eventualmente serão realizadas entrevistas sobre o trabalho para complementar a avaliação;
- O trabalho é em grupo de até 2 (duas) pessoas.
- Será aceito trabalhos após a data de entrega, todavia com um decréscimo de 0,05 a cada 10min.
- Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.
- Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.
- Códigos ou funções prontas específicas de algoritmos para solução dos problemas elencados não são aceitos
- Não serão considerados algoritmos parcialmente implementados.
- Procedimento para a entrega:.
 1. Submissão: via **Moodle**.
 2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
 3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
 4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
 5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via **Moodle**.
 6. Você deve submeter os arquivos *.h*, *.c* e o *.pdf* (relatório) na raiz do arquivo *.zip*. Use os nomes dos arquivos *.h* e *.c* exatamente como pedido.
 7. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.
- **Bom trabalho!**

Simulação de um Escalonador de Processos

Dentro do Kernel de um sistema operacional, o escalonador decide quais processos terão acesso a CPU e por quanto tempo eles irão executar. Neste trabalho você deverá criar um algoritmo que simule um escalonador simples, levando em consideração a ordem de chegada dos processos, suas prioridades e a quantidade de ciclos de CPU.

O escalonador deve atender primeiro os processos mais prioritários, seguindo a ordem de chegada deles. Além disso, os processos não podem executar por mais que X ciclos na CPU, sendo reinseridos na fila de processos se ainda faltarem ciclos. Neste trabalho, tomaremos $X = 100$.

Imposições e comentários gerais

Neste trabalho, as seguintes regras devem ser seguidas:

- Seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada)
- Um grande número de *Warnings* ocasionará a redução na nota final.

O que deve ser entregue

- Código fonte do programa em C (**bem indentado e comentado**).
- Documentação do trabalho (relatório¹). A documentação deve conter:
 1. **Implementação:** descrição sobre a implementação do programa. Não faça “*print screens*” de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação.
 2. **Impressões gerais:** descreva o seu processo de implementação deste trabalho. Aponte coisas que gostou bem como aquelas que o desagradou. Avalie o que o motivou, conhecimentos que adquiriu, entre outros.
 3. **Análise:** deve ser feita uma análise dos resultados obtidos com este trabalho.
 4. **Conclusão:** comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
 5. **Formato:** PDF.

Como deve ser feita a entrega

Verifique se seu programa compila e executa na linha de comando antes de efetuar a entrega. Quando o resultado for correto, entregue via *Moodle* até a data disponível na plataforma de entrega um arquivo **.ZIP** com o nome e sobrenome do aluno. Esse arquivo deve conter: (i) os arquivos *.c* e *.h* utilizados na implementação, (ii) instruções de como compilar e executar via terminal, e (iii) o relatório em **PDF**.

Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados para representar uma fila de processos. O TAD deverá implementar, pelo menos, as seguintes operações:

1. **criarFila:** aloca um TAD `FilaProcessos`.
2. **destroiFila:** desaloca um TAD `FilaProcessos`.

¹Exemplo de relatório: <https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsgmcdg>.

3. `adicionaFila`: adiciona um processo na fila.

4. `escalonador`: função que vai atender os processos seguindo a fila criada. Se um processo exceder o máximo de ciclos por processo, ele deve executar a quantidade máxima de ciclos e ser reinserido na fila com sua prioridade incrementada em 1.

5. `imprimeLog`: imprime na tela a ordem de atendimento dos processos, com seus ids e prioridades.

Além disso, para ordenar os processos de acordo com a ordem de chegada e prioridade é necessário utilizar um algoritmo de ordenação modificado. Para este trabalho o método de ordenação utilizado será o *insertion sort* que deverá ser implementado dentro da função `adicionaFila`.

Ao aluno que fizer a ordenação inicial utilizando um algoritmo de *MergeSort* modificado, será garantido 1,0 (um) ponto extra no trabalho. Para isso o algoritmo deve funcionar corretamente e seguir ao menos o modelo do algoritmo original, mantendo a noção de divisão e conquista. É necessário que somente a ordenação inicial seja feita dessa forma, as inserções posteriores na fila devem utilizar *insertion sort*.

Fica a cargo do aluno a implementação do TAD `FilaProcessos`, porém é sugerida a criação de um TAD `Processo` para simplificar a implementação do TAD `FilaProcessos`.

O TAD deve ser implementado utilizando a separação da interface no `.h` e implementação `.c` discutida em sala, bem como as convenções de tradução. Caso a operação possa dar errado, devem ser definidos retornos com erro, tratados no corpo principal.

O código-fonte deve ser modularizado corretamente em três arquivos: `tp.c`, `filaprocessos.h` e `filaprocessos.c`. O arquivo `tp.c` deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo `filaprocessos.h`. A separação das operações em funções e procedimentos está a cargo do aluno, porém, **não deve haver acúmulo** de operações dentro de uma mesma função/procedimento.

Atenção! Somente a primeira ordenação leva em conta o instante de chegada. A reinserção ordenada na fila de um processo que já foi atendido deve levar em conta apenas a prioridade do processo.

Entrada

A entrada é dada por meio do terminal e é composta de vários conjuntos de teste. Para facilitar, a entrada será fornecida por meio de arquivos.² A primeira linha de um conjunto de testes contém um inteiro N que representa a quantidade de processos a serem lidos. Portanto, as N linhas seguintes contêm, cada uma, a descrição de um processo. Um processo é descrito por quatro valores ID , T , P e C que representam, respectivamente, o identificador, o tempo de chegada, a prioridade e a quantidade de ciclos de um processo. Esses processos fazem parte de um conjunto de processos que chegaram num mesmo intervalo de tempo e não estão ordenados.

Saída

Para cada teste seu programa deve imprimir ao final o log de atendimento dos processos, indicando a ordem em que os processos utilizaram a CPU, sua prioridade naquele instante e por quanto tempo eles ficaram na CPU.

Exemplo de um caso de teste

Exemplo da saída esperada dada uma entrada:

Entrada	Saída
3 1256 0.02 4 70 1546 0.04 2 250 2548 0.05 2 60	Log CPU: 1546 2 100 2548 2 60 1546 3 100 1256 4 70 1546 4 50

²Para usar o arquivo como entrada no terminal, utilize `./executavel < nome_do_arquivo_de_teste`.

O seu trabalho deve simular os ciclos de uso da CPU por meio de loops vazios.

Diretivas de Compilação

As seguintes diretivas de compilação devem ser usadas (essas são as mesmas usadas no `run.codes`).

```
$ gcc -c aluno.c -Wall
$ gcc -c tp.c -Wall
$ gcc aluno.o tp.o -o exe
```

Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta *valgrind*. O *valgrind* é um *framework* de instrumentação para análise dinâmica de um código e é muito útil para resolver dois problemas em seus programas: **vazamento de memória e acesso a posições inválidas de memória** (o que pode levar a *segmentation fault*). Um exemplo de uso é:

```
1 gcc -g -o exe *.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

```
1 ==xxxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.