UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
http://www.icmc.usp.br/

Professor: Rodrigo Fernandes de Mello (mello@icmc.usp.br)
Alunos PAE: Lucas Pagliosa (lucas.pagliosa@usp.br)
Ricardo Fuzeto (ricardofuz@usp.br)
Yule Vaz (yule.vaz@gmail.com)
Monitores: Victor Forbes (victor.forbes@usp.br)

Trabalho 4: Simulador de Escalonador de Processos

1 Prazos e Especificações

O trabalho descrito a seguir é individual e não será tolerado qualquer tipo de plágio ou cópia em partes ou totalidade do código. Caso seja detectada alguma irregularidade, os envolvidos serão chamados para conversar com o professor responsável pela disciplina.

A entrega deverá ser feita única e exclusivamente por meio do sistema run.codes no endereço eletrônico `https://run.codes`. Sejam responsáveis com o prazo final para entrega: o run.codes disponibiliza o prazo máximo de submissão dos trabalhos, está programado para não aceitar submissões após este prazo e não serão aceitas entregas fora do sistema.

Leia a descrição do trabalho com atenção e várias vezes, anotando os pontos principais e as possíveis formas de resolver o problema. Comece a trabalhar o quanto antes para que você não fique com dúvidas e que consiga entregar o trabalho a tempo.

2 Descrição do Problema

Um *processo* é um *software* em execução. Para o Sistema Operacional (SO), é importante a distinção entre estes dois conceitos: um processo possui mais informações que o descrevem, já que se trata de um *software* em execução, e, portanto, possui informações como o ponto atual de execução e o valor de cada variável declarada, por exemplo. No entanto, um SO pode permitir que diversos processos sejam executados ao mesmo tempo. No entanto, sabe-se que a quantidade de núcleos de um processador é limitada: dado um processador com $\mathcal{X}$ núcleos, o SO pode executar, no máximo, $\mathcal{X}$ processos ao mesmo tempo, mesmo que o SO possua $2\mathcal{X}$ processos "em execução".

Esse comportamento só é possível porque o SO permite que cada processo execute por um breve momento, chamado *quantum*. A rotina do SO que cuida do "revezamento" entre os processos, decidindo quais podem ser executados em um dado núcleo do processador, é chamada de *escalonador de processos*.

De acordo com a Figura 2, podemos ver que os processos não executam ao mesmo tempo. O núcleo do processador não é capaz de executar dois processos ao mesmo tempo, e por isso eles devem "revezar" para serem executados. No entanto, nós, usuários, enxergamos os processos como executados ao mesmo tempo. Isso acontece porque o *quantum* de um SO geralmente é pequeno (alguns milissegundos), então os processos são alternados frequentemente. No entanto, para garantir que os processos tenham porções justas de tempo em execução, o escalonador deve utilizar algoritmos para *decidir* quais processos devem ser executados a cada instante. Alguns dos algoritmos mais conhecidos para realizar esta decisão são:

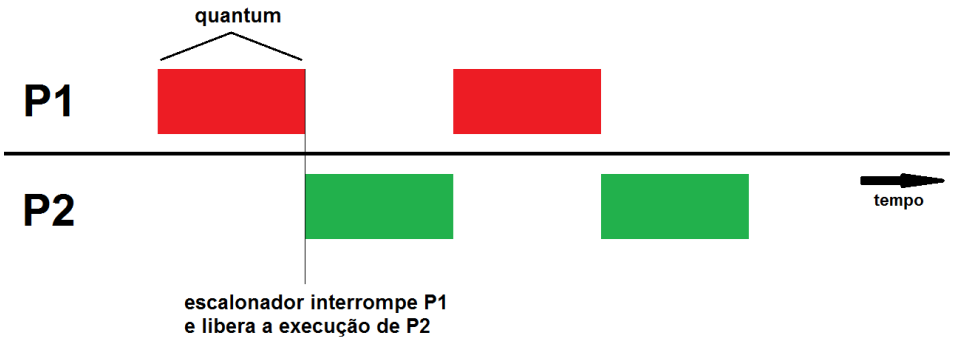

Figura 1: Exemplo de funcionamento de um escalonador para um processador de 1 núcleo. O processo P1 começa executando, tendo um *quantum* de execução. Quando esse tempo acaba, o escalonador decide qual o processo a ser colocado em execução, neste caso P2.

- **First In, First Out (FIFO)**: os processos que são reconhecidos primeiro pelo escalonador são executados *até o final*. Em geral, este algoritmo utiliza uma *fila* para organizar os processos. Também é conhecido por *First Come, First Served*;
- **Shortest Remaining Time First**: este algoritmo é parecido com o *Shortest Job First*. Os processos executados são os com menor tempo de execução restante;
- **Round-Robin**: este algoritmo apenas arranja todos os processos em uma lista, e trata todos os processos como tendo a mesma prioridade. Assim, um processo não é executando uma segunda vez antes que todos os outros processos executem ao menos uma vez. Pode-se pensar na execução do *Round Robin* como a utilização de uma *lista circular* em que todos os elementos recebem a mesma quantidade de recursos. Pode-se pensar na execução do *Round Robin* como uma lista circular;

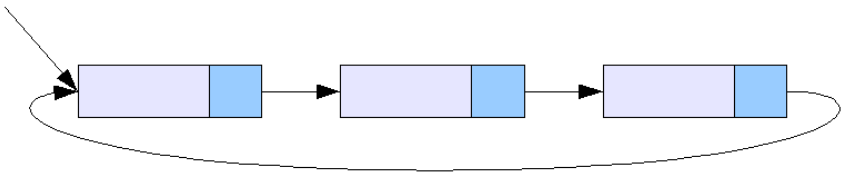

Figura 2: Exemplo de funcionamento do algoritmo *Round Robin*. O primeiro processo executado é o que está na cabeça (head) da lista, seguindo a sequência da lista a partir daí. Após executar o último processo da lista, o algoritmo volta para a cabeça da lista, e estes passos são repetidos até a lista estar vazia.

- **Prioridade Fixa Preemptiva**: este algoritmo divide os processos em diferentes níveis de prioridade, e um processo com nível mais alto de prioridade sempre é executado antes de qualquer processo de prioridade mais baixa.

3 Requisitos do Projeto

O trabalho consiste na implementação de um simulador de escalonador de serviços, na linguagem C. O simulador deve receber uma lista de processos, contendo o código de identificação de um processo p_i, o tempo t_{0i} em que o processo é reconhecido pelo escalonador e o tempo t_{fi} que o processo leva para ser executado. O simulador deve exibir, como saída, o tempo t_{fi} em que o processo p_i tem sua execução finalizada. O simulador deve utilizar o tempo t_{fi} apenas para

controlar a execução da simulação, portanto o escalonador implementado **não deve utilizar o tempo t_{fi} em consideração**.

O código de identificação p_i de um processo deve ser único. Caso o simulador reconheça um novo processo com o mesmo código, este novo processo deve ter seu código alterado para o *próximo código maior disponível*. O código p_i de um processo está no intervalo $[1, \texttt{MAXINT}]$. Os tempos t_{0i} e t_{fi} do processo p_i são do tipo `int`.

Cada processo também possui um nível r_i de prioridade. Dados dois processos p_i e p_j, o algoritmo usado é *Round Robin* caso $r_i = r_j$. No entanto, caso $r_i > r_j$ então o *Round Robin* ainda será usado, no entanto iniciando o ciclo de iterações por p_i. O simulador deve possuir apenas 4 *níveis de prioridade distintos*, e a ordem de execução entre processos de mesmo nível de prioridade é dada por ordem **ascendente** de código. O simulador não interrompe a execução de processos num dado nível de prioridade, no entanto retorna para um nível de prioridade superior caso haja um novo processo naquele nível. O novo processo sempre é reconhecido pelo escalonador antes de decidir o novo processo a ser executado. Portanto, no momento da "chegada" de um novo processo, o escalonador ainda não deve ter escolhido um novo processo, apontando para o processo executado no ciclo anterior.

Para cada um dos níveis de prioridade, o simulador deve realizar a ordenação dos processos em ordem ascendente de p_i.

Para padronizar a contagem do tempo, o simulador deve operar em frequência de **1 quantum**: cada ciclo de atividade do simulador corresponde a 1 *quantum* de execução. Portanto, cada **iteração** do simulador altera o estado dos processos em execução correspondente à passagem de 1 *quantum* de tempo.

4 Arquivos de Entrada e Saída

Os dados do arquivo de entrada definem a lista de processos a ser executada pelo simulador. Dada uma lista $\mathcal{L}$ de processos, cada linha de $\mathcal{L}$ deve conter o código p_i de um processo, seu tempo inicial de entrada no simulador t_{0i}, o volume de *quanta* necessário para ser completamente executado t_{fi}, e o nível de prioridade do processo r_i. O arquivo de entrada terá o seguinte formato:

- `(int)`p_0 `(int)`t_{00} `(int)`t_{f0} `(int)`r_0
- `(int)`p_1 `(int)`t_{01} `(int)`t_{f1} `(int)`r_1
- $\vdots$
- `(int)`p_{i-1} `(int)`t_{0i-1} `(int)`t_{fi-1} `(int)`r_{i-1}
- `(int)`p_i `(int)`t_{0i} `(int)`t_{fi} `(int)`r_i

Após terminar a execução de todos os processos definidos no arquivo de entrada, o simulador deve criar um arquivo de saída, com os dados da simulação. O arquivo de saída possui, em cada linha, os dados da execução de um único processo p_i, contendo: i) o código p_i do processo, atualizado pelo simulador; ii) o ciclo de execução do simulador em que a execução de p_i foi finalizada; e iii) uma quebra de linha. Os processos no arquivo de saída devem ser apresentados em ordem cronológica ascendente (processos que terminam primeiro são exibidos antes).

Por exemplo: dada a seguinte lista $\mathcal{L}$ de processos:

```
47 1 7 2
125 1 8 3
2408 5 3 1
2408 7 14 3
445 12 20 4
307 13 12 3
225 20 10 2
```

A saída produzida deve ser a seguinte:

```
2408 16
47 27
125 30
225 61
307 63
2409 65
445 73
```

5 Dicas

- Defina onde o *loop* de execução do simulador ficará, bem como quais operações serão feitas dentro dele e quais serão invocadas de outro lugar. Isto irá ajudar a manter o código-fonte mais limpo e legível.

6 Desafios

- **Nível de prioridade de tempo real**: o escalonador deve possuir um nível de prioridade especial (identificado como nível zero), onde os processos deste nível possuem prioridade total de execução. Quando um novo processo neste nível de prioridade for reconhecido, deve ser executado imediatamente. Além disso, os processos de nível 0 executam em ritmo *Round Robin* entre si, porém nenhum processo de outro nível pode executar antes que todos os processos de tempo real sejam finalizados.

- **Algoritmo de escalonamento seletivo**: além do algoritmo *Round Robin*, o escalonador também deve poder escolher as ordens de execução utilizando FIFO. Nos casos em que o FIFO for utilizado, o arquivo de entrada terá um caractere "f" na primeira linha, seguido dos dados dos processos nas linhas seguintes. Neste caso, a ordem de execução dos processos será por ordem **ascendente** de código, e o nível de prioridade de cada processo não será informado.

7 Observações importantes

- Programe as impressões na tela EXATAMENTE como exemplificado no decorrer deste documento. Tome cuidado com pulos de linha, tabs, espaços, etc.

- Coloque dentro do zip todos os arquivos de código (`*.h *.c`), o makefile e um arquivo texto com o nome e número USP.