Prof Otávio Gomes (otavio.gomes@unifei.edu.br)

Roteiro 02

Nano kernel e escalonador

1) A partir da análise do código-fonte disponível no arquivo ***ECOS13_Lab02_1_Kernel.c***, temos:

- O código reagenda processos que retornam **REPEAT** adicionando-os novamente ao buffer. No entanto, isso pode causar um loop infinito se o buffer só contiver processos que retornem **REPEAT**. Seria útil implementar uma lógica para lidar com essa condição, possivelmente limitando o número de reagendamentos.
- O código utiliza ponteiros estáticos para processos adicionados ao *buffer*. Isso é adequado para o exemplo, mas em uma aplicação real deve ser considerada. Deve ser considerada, também, a alocação dinâmica e a liberação de memória para processos que podem ser criados e destruídos em tempo de execução pode ser necessário.
- Para tornar o kernel mais flexível, pode-se permitir que os processos tenham diferentes prioridades ou tipos de reagendamento. Isso poderia ser alcançado expandindo a estrutura *process* para incluir essas propriedades.

2) A partir das informações apresentadas e do código-fonte fornecido, faça:

a) Implemente tratamentos para todas as ocorrências de um **return FAIL**.

b) Implemente uma limitação para o reagendamento de processos. Para evitar *loops* infinitos causados por processos que são continuamente reagendados, pode-se introduzir um limite de reagendamento para cada processo. Para isso, uma sugestão é expandir a estrutura **process** para incluir um contador de reagendamentos e um limite máximo permitido para cada processo:

```
typedef struct {
   ptrFunc func;
   int rescheduleCount;
   int rescheduleLimit;
} process;
```

Nesta abordagem é interessante modificar a função **kernelLoop** para incrementar **rescheduleCount** cada vez que um processo é reagendado, e só permitir o reagendamento se esse contador estiver abaixo de **rescheduleLimit**.

c) Implemente uma variável **count** para limitar execuções do *loop*. Tendo como objetivo limitar o número total de iterações do loop do kernel e evitar uma execução indefinida. Utilize a variável **count** efetivamente. O valor máximo de iterações deverá ser definido pelo usuário no processo de inicialização (**kernelInit**).

d) Aprimore a flexibilidade de execução com prioridades de processos. Para isso é necessário introduzir prioridades para os processos e permitir que o kernel trate diferentes processos de maneira apropriada, baseando-se na sua urgência ou importância.
Com o objetivo de aprimorar o escalonador e permitir diferentes políticas de escalonamento (como Round-Robin, Prioridades, FIFO, etc.), é possível implementar uma seleção dinâmica de algoritmos de escalonamento.
Para isso desenvolva **duas funções** (ponteiro de funções) que permitam que o kernel escolha dinamicamente o algoritmo de escalonamento com base na configuração ou condições de sistema. Implemente funções de escalonamento específicas que podem ser trocadas em tempo de execução, por exemplo FIFO e SJF (*Shortest-Job First)*. Estas funções substituirão a chamada *kernelLoop.*
Seguem algumas orientações:

- Para implementar as funções **FIFO** (*First In, First Out*) e **SJF** (*Shortest Job First*) será necessário modificar a estrutura *process* para incluir uma nova variável denominada ***tempo_restante***: `int tempo_restante;`
- A implementação **FIFO** não é afetada pela adição da variável *tempo_restante*, pois essa estratégia simplesmente executa os processos na ordem em que foram adicionados, independentemente de seu tempo restante.
- Para a implementação de **SJF**, é necessária uma lógica que selecione o processo com o menor *tempo_restante* para execução. Dado que o buffer é circular e os processos não estão necessariamente ordenados por *tempo_restante*, é necessário percorrer o *buffer* para encontrar o próximo processo a ser executado. AO ser encontrado, este processo será trocado com aquele que ocupa a posição inicial (start) e a execução poderá ser realizada normalmente.