

# EP2 - MAC0422 - 2015

## main, utils, mem\_mgr e page\_mgr

Renato Lui Geh e Ricardo Fonseca

# Linha de input

Assim como no EP1, há uma string `cmd` que é a linha de input do usuário. Em seguida dividimos `cmd` em tokens tais que cada par de token  $(t_i, t_{i+1})$ , quando concatenados produzem uma subsequência de `cmd`. Ou seja, a concatenação  $(t_0, \dots, t_k)$  resulta em `cmd`. Isso é feito na função `extract_args`. O token  $t_i$  equivale ao código `args_table[i]`. Note que  $t_0$  equivale ao próprio comando, e não o primeiro argumento do comando.

# Prompt

Foi usado `getline` para o prompt, assim como foi feito no EP1.

# evaluate\_str

C e C++ não permitem usar `switch` com `cstrings` (`char*`). Portanto para podermos comparar o comando num `switch` precisamos transformar a string em um `unsigned int`. Para isso usamos `evaluate_str`.

```
/* Transforma string em um unsigned integer. */  
constexpr unsigned int evaluate_str(const char* str, int h = 0) {  
    return !str[h] ? 5381 : (evaluate_str(str, h+1)*33) ^ str[h];  
}
```

O que fazemos é passar por cada caractere  $c_i$  da string e transforma-lo em um número inteiro não negativo  $k_i$  tal que ele seja dependente do último caractere, ou seja, dada uma transformação  $T : \text{char} \rightarrow \mathbb{Z}^+$  então  $k_i = T(c_i) | T(c_{i-1})$  (leia-se  $k_i$  é a transformação de  $c_i$  dada a transformação de  $c_{i-1}$ ).

# Parsing

A função `parse` lê o arquivo de trace e cria os processos e memória a serem simuladas.

## Escrevendo bytes

Para escrevermos a memória byte a byte, foi usada a função `fwrite` da `cstdio`. Para garantirmos que o que realmente escrevemos é um byte, usamos um `unsigned char`. Para representar o número -1, foi escolhido 255 para representar memória livre. Portanto, no total, podemos ter 255 processos rodando.

```
void __write_bytes(FILE* stream, int i, int f, unsigned char val)
{
    fseek(stream, i, SEEK_SET);

    while (i++ <= f)
        fwrite(&val, sizeof(val), sizeof(val), stream);
}
```

Aqui pode-se ver que o intervalo  $[i, f]$  (ambos extremos inclusivos) é escrito com o valor `val`.

# Gerenciador de Memória

Para facilitar o trabalho da implementação do nosso Gerenciador de Memória, criamos dois arquivos: `utils.c` e `utils.h`. Nesses arquivos foram criados as structs para representar `mem_node`, `size_node`, funções para auxiliar algoritmos e escrever bytes no arquivo de saída.

Listas duplamente ligadas com cabeça foram criadas para armazenar e manipular os blocos de memória mais facilmente. Para o algoritmo Quick Fit usamos uma lista de listas para separar os diferentes tamanhos de espaços livres na memória.

## mem\_node e size\_node

Para manipular a memória virtual, utilizamos duas structs diferentes, uma representando um bloco na memória virtual e outro que é apenas usado para o algoritmo Quick Fit

`mem_node`: Possui 6 campos: (`char t`) para o tipo do bloco, (`int i`) para a posição início da memória do bloco, (`int s`) para tamanho do bloco, (`int pid`) para o identificador do processo utilizando o bloco, (`mem_node *n`) e (`mem_node *p`) como ponteiros para os próximos nós da lista.

`size_node`: Lista de listas de tamanhos de blocos para o algoritmo Quick Fit. Possui 4 campos. (`int s`) para tamanho dos blocos da lista que o nó aponta, (`mem_node *f`) como ponteiro para uma lista ligada sem cabeça com os blocos livres de tamanho `s`, (`size_node *n`) e (`size_node *p`) como ponteiros para os próximos nós da lista.



# Algoritmos

Para ser mais fácil de se escolher qual gerenciador usar, criamos a variável `manager`, que é um ponteiro para função. Os argumentos de linha de comando são lidos e a função é atribuída a `manager` em seguida.

O gerenciador pelo método Quick Fit utiliza uma lista de listas, que é criado apenas no caso dele ser escolhido ao rodar o programa.

# Listas Ligadas

Em todos os gerenciadores implementados foram usadas a lista com a cabeça `v_mem_h`. Para a memória física total utiliza-se `t_mem_h`

`v_mem_h`: guarda todos os blocos de memória livres ou ocupados na memória virtual. O algoritmo Quick Fit utiliza de forma um pouco diferente essa lista.

`t_mem_h`: guarda todos os blocos de memória ocupando a memória física total.

# First Fit (FF)

Este gerenciador usa diretamente a lista da memória virtual. Quando o  $t\_secs$  chega no instante  $t_0$  de um processo, se há algum espaço livre que ele caiba, o processo é atribuído àquela parte da memória virtual. O bloco de memória é mudado para P.

O processo permanece na memória até terminar.

## Next Fit (NF)

Este gerenciador também usa diretamente a lista da memória virtual. Quando o `t_secs` chega no instante `t0` de um processo, se há algum espaço livre que caiba o processo, o processo é atribuído àquela parte da memória virtual. Porém usamos um apontador de nó especial `v_last` que marca a última posição vista na lista, dessa forma para futuros usos da função, ela começa do último bloco criado.

# Quick Fit (QF)

Ao escolher o algoritmo Quick Fit, o dicionário é criado dinamicamente por nosso algoritmo, que faz o seguinte:

- Pega o valor disponível (inicialmente toda a memória virtual disponível).
- Divide em 4 esse espaço, e aproxima até o maior múltiplo de 2 menor que esse valor.
- Se o resultado for maior que o limite inferior (definido por nós como 16, de acordo com o tamanho das páginas), separa  $3/4$  do tamanho total recebido pela função como espaço livre nas listas (para cada  $1/3$  é criado um bloco de espaço livre desse tamanho próximo de  $1/4$  do tamanho total). Em seguida chama a função recursivamente para o restante do espaço.
- Caso o resultado seja igual ou menor que o limite inferior, separa o maior número de intervalos com tamanho limite inferior como espaço livre na lista do quick fit.

## QF - Continuação

Quando o `t_secs` chega no instante `t0` de um processo, ele procura o menor bloco livre que seja maior ou igual que o tamanho necessário do processo (percorre a lista de `size_node`), e depois marca como ocupado esse bloco, e põe no começo da lista `v_mem_h` esse processo (essa lista ligada não possui ordenação para processos, apenas tem todos os blocos que estão sendo ocupados por processos).

# ff\_nf\_free

Tanto para o gerenciador de memória FF quanto NF utilizamos a mesma função para quando um processo termina. De forma simples ela transforma o bloco como livre, e com a ajuda das listas duplamente ligadas, verifica se o bloco estava entre algum outro bloco livre, eficientemente unindo-os em um só bloco pronto para ocupar outro processo.

## qf\_free

Para o gerenciador de memória QF utilizamos a função `qf_free` para quando um processo termina. Ela transforma o bloco do processo como livre, e utilizando nossa lista de listas, disponibiliza como livre o bloco na sua respectiva lista (a que possui outros blocos com seu mesmo tamanho).



# Paginação

Para paginação, rodamos cada processo e se no tempo  $t_i$  houver um acesso, vemos se a memória a ser acessada está na memória física. Se não estiver, page fault e usamos um dos algoritmos de substituição de página. Senão não fazemos nada, já que estamos apenas simulando e não realmente lendo a memória.

## Not Recently Used Page (NRU)

Foi usado um vetor `virt_refs::internal` para representar cada página da memória virtual. Nesse vetor estão os dados de acesso de cada página. Dado um elemento  $e_i$ , se  $e_i = -1$ , então ele não foi acessado recentemente. Senão, se  $e_i = 0$ , então ele foi acessado neste ciclo. Senão, então  $e_i > 0$  e portanto ele foi acessado  $e_i$  segundos atrás. Se  $e_i$  for maior ou igual ao limite `INTERRUPT_DT` que representa o tempo decorrido após cada *reset*, então  $e_i = -1$ . A função `nru_repl` procura por  $e_i, \dots, e_j$  que sejam, juntos, do tamanho requisitado e que tenham menor peso. Definimos peso como:

$$w_i = 1, \text{ se } e_i \geq 0$$

$$w_i = 0, \text{ se } e_i < 0$$

Portanto, queremos  $\min(\sum_{k=i}^j w_k)$ .

# First-In, First-Out (FIFO)

Neste algoritmo de substituição de página, apenas temos uma fila `page_queue` e toda vez que ocorre um *page fault*, pegam-se as  $n$  primeiras páginas ocupadas (e portanto mais "velhas") tal que a soma dos tamanhos delas sejam a menor soma possível e que sejam também maior ou igual ao tamanho requerido.

## Second-Chance Page (SC)

Muito parecido com nosso FIFO, o algoritmo SC vai usar uma fila `page_queue`, porém ao ocorrer uma page fault, antes de retirar automaticamente a página do topo (as mais "velhas"), ele verifica o elemento  $e_i$  da página. Caso ele for  $\geq 0$ , faz ele virar  $-1$  e põe a página no fim da fila. Caso o contrário apenas remove que nem o algoritmo FIFO.

# Observações

Nosso algoritmo Quick Fit divide a memória virtual até tamanhos maiores ou iguais que o limite inferior, o que gera um problema de desperdício de memória, pois se a memória não for múltiplo dele (16 no caso), é possível que até 15 bytes sejam desperdiçados e nunca utilizados, mas comparado com a memória virtual disponível não é de grande impacto esse valor.