Relatório EP3- MAC0422

Thiago Santos Teixeira -nUSP 10736987

Wander Souza - nUSP 10737011

###

Primeira Parte:

Para a primeira parte do EP, tivemos que criar uma nova `system call` que alterasse a política de alocação de memória do MINIX entre `worst_fit` e `first_fit`. Para isso, começamos tentando entender como funcionavam esses diferentes algoritmos de alocação, lendo assim as referências sobre eles no livro do MINIX. Assim, vimos que o `first_fit` aloca a memória sempre no primeiro espaço de tamanho suficiente, enquanto o `worst_fit` percorre toda a memória, e aloca sempre na maior lacuna possível.

Tendo entendido o funcionamento dos algoritmos, começamos a criar a nossa nova system call `memalloc()`. Para isso, utilizamos o tutorial disponibilizado pelos monitores no EP2 da matéria, portanto, não detalharemos o processo para criá-la, já que isso já foi feito no EP passado. Essa nova chamada será responsável para informar ao sistema qual método de alocação de memória deve ser usado, ou seja, criamos uma variável pública `int mode` em um novo arquivo header `alloc.h`, que adicionamos exclusivamente para ela em `usr/src/servers/pm`, essa recebe um inteiro e será responsável por armazenar qual algoritmo de alocação deve ser usado, **0** caso seja desejado o `first_fit` e **1** caso `worst_fit`.

Com a chamada de sistema pronta, nosso próximo passo foi alterar o código da função `alloc_mem()` no arquivo `alloc.c` em `usr/src/servers/pm`. Ela é a principal responsável pela alocação de memória no MINIX e recebe como parâmetro o tamanho do espaço desejado para ser alocado, e realiza por padrão o `first_fit`. Tendo isso em mente, percebemos que era simples adicionar o `worst_fit` como opção, isso já que, o código para alocar a memória em si seria praticamente o mesmo, a única mudança seria na escolha da lacuna.

Logo, adicionamos uma condição que checa o valor de `mode`, caso o método desejado seja o `first_fit`, o MINIX realiza seu código padrão, caso seja o `worst_fit` ele percorre toda a memória em busca da maior lacuna, e quando é encontrada roda exatamente o mesmo código de alocação, mas recebendo como ponteiro o maior buraco ao invés do primeiro.

Para verificar o `effective UID` do usuário que realizou a chamada, usamos da chamada `geteuid()` da biblioteca `unistd.h` no arquivo `/usr/src/lib/posix/_memalloc.c` e passamos seu retorno por message para a misc.c no pm, que apenas termina a `system_call` caso o valor recebido seja 0, ou seja, caso o usuário seja o root.

Segunda parte:

A segunda parte do EP consistia de implementar um utilitário para imprimir um mapa da memória e a memória livre no sistema, para isso, nos inspiramos fortemente no código do top, em `/usr/src/commands/simple/top.c`.

Começamos com a implementação da parte mais simples, que era imprimir a quantidade de memória livre no sistema. Para isso, apenas adaptamos o código da função `print_memory()` do top.c, que já fazia isso de maneira muito prática e colocamos em nosso arquivo.

Após isso, começamos a montar nossa função `print_table`, que seria a responsável por imprimir o mapa da memória como desejado. Nela, percorremos a process table do minix e para cada processo, pegamos seu `pid`, posição do `.text` e `.stack` e por fim o tamanho da `.stack`. Isso já que, a posição do `.text` na memória será também a posição de início do processo, e a soma da posição da `.stack` com seu tamanho será sua posição final, como pode ser visto na imagem abaixo:

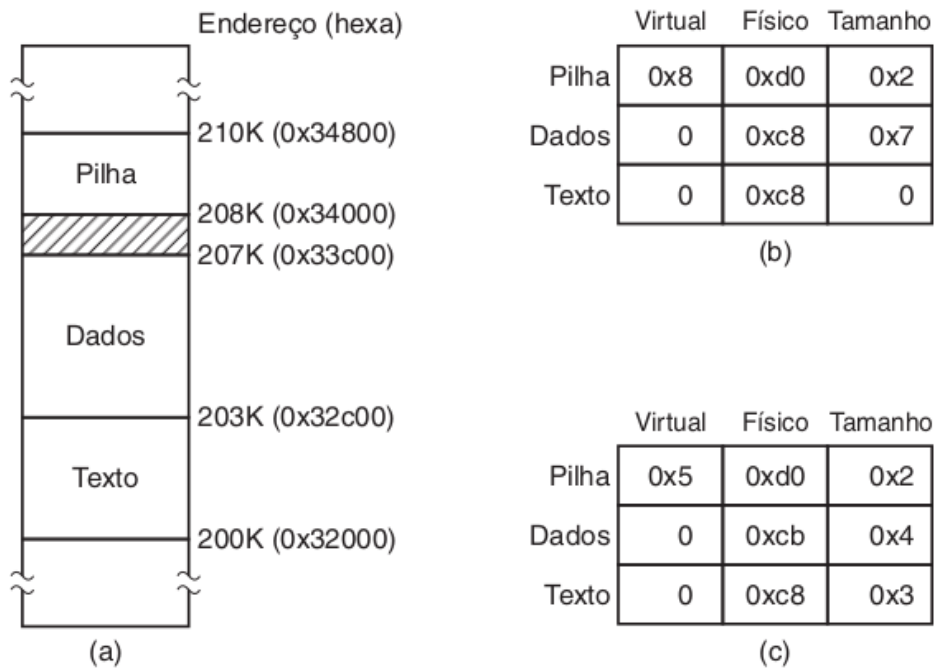

Figura 4-33 (a) Um processo na memória. (b) Sua representação de memória para espaços I e D combinados. (c) Sua representação de memória para espaços I e D separados.

Logo, tendo esses valores e o `pid` de cada processo, bastava imprimir os resultados separados por `\t` que teríamos a tabela desejada.