Atividades 3: Espaço de Endereçamento e API de Memória

Atividade 3.1 - Espaço de Endereçamento Virtual (Capítulo 13)

Objetivo: Observar o comportamento do espaço de endereçamento virtual e a relação entre memória virtual e física.

1. Leia o manual do comando `free`: `man free`.
2. Execute `free -m` para ver a quantidade de memória total e livre.
3. Crie um programa chamado `memory-user.c` que:
   - Recebe como argumento a quantidade de MB de memória a ser alocada.
   - Aloca a memória dinamicamente (com `malloc`).
   - Percorre a memória continuamente ou por um tempo determinado.
4. Enquanto o programa estiver em execução, rode `free -m` em outro terminal e observe a variação do uso de memória.
5. Teste com diferentes quantidades de memória e observe o comportamento do sistema.
6. Utilize o comando `ps aux` para descobrir o PID do processo.
7. Use `pmap <PID>` e explore as flags como `-X` para visualizar detalhes.
8. Execute novamente o programa com diferentes quantidades de memória e observe a saída do `pmap`.

Perguntas:

- A memória total e livre apresentada pelo comando `free` corresponde às suas expectativas?
- O que muda nas estatísticas de memória enquanto o programa está rodando? E após ser finalizado?
- Quantas regiões distintas são exibidas no `pmap`? Que tipos de segmento aparecem além de código, heap e stack?

Atividade 3.2 - Alocação de Memória em C (API de Memória)

Objetivo: Compreender e identificar erros comuns em alocação de memória usando `malloc` e `free` com ferramentas de depuração.

1. Crie o programa `null.c` que define um ponteiro como `NULL` e tenta acessá-lo. Compile e execute.
2. Compile novamente usando a flag `-g` para habilitar informações de depuração.
3. Rode o programa com `gdb` e identifique o erro.
4. Execute com `valgrind --leak-check=yes` e observe a saída.
5. Crie um programa que aloque memória com `malloc()` e não libere com `free()`. Use `valgrind` para detectar vazamento.
6. Crie um array com `malloc` e acesse além do limite (ex.: `data[100]`). Observe o erro com `valgrind`.
7. Libere a memória com `free()` e tente acessar o valor. Use `valgrind` para identificar o erro.
8. Faça um `free()` incorreto, passando um ponteiro inválido (ex.: `ptr+1`). Avalie com as ferramentas.
9. Crie um vetor dinâmico que se expande com `realloc()` quando novos elementos são adicionados.

Perguntas:

- O que acontece ao acessar um ponteiro `NULL`? Como o `gdb` e o `valgrind` ajudam a identificar o problema?
- Ao alocar memória e não liberar, o que o `valgrind` aponta?
- O que ocorre ao acessar memória fora dos limites alocados? E ao liberar e usar logo em seguida?
- O uso de `realloc()` em uma estrutura dinâmica (vetor) funcionou como esperado?