实验三 遍历进程地址空间

班级: 07112303 学号: 1120231863 姓名: 左逸龙

一、 实验目的

1. **理解虚拟内存的具体形态** 将教材中抽象的“虚拟地址空间”概念具象化。通过编写程序观察进程内部内存的提交 (Committed) 、保留 (Reserved) 和空闲 (Free) 状态，深入理解进程对内存的实际使用模式。
2. **熟悉 Windows 系统调用** 学习使用 Windows API 进行系统级编程。本次实验主要使用到了 `VirtualQueryEx` 遍历内存，以及 `GetSystemInfo` 和 `GlobalMemoryStatusEx` 等函数，我们可以通过查阅官方文档掌握这些 API 的参数规范和调用方法。
3. **分析内存布局与权限** 观察不同内存区域的保护属性 (如只读、可读写、可执行) ，理解操作系统如何利用这些属性实现代码与数据的隔离，以及如何通过工作集 (Working Set) 机制实现内存共享与资源优化。

二、 实验内容

本次实验旨在 Windows 10/11 环境下，使用 C++ 开发一个控制台应用程序，功能类似于简化的内存分析工具。主要功能模块如下:

1. **获取系统基本信息** 程序启动后，首先查看当前系统的内存情况。我们可以了解系统页大小 (Page Size) ，用户地址空间的起始与终止位置，以及当前物理内存的总容量与负载情况。
2. **扫描指定进程的内存** 这是实验的核心功能。针对用户输入的 PID，程序将对该进程的虚拟地址空间进行全量遍历。对于每一块连续的内存区域，程序需解析并输出其起始地址、区域大小、当前状态以及内存保护属性。
3. **查看性能指标** 结合 PSAPI 库，查询目标进程的物理内存占用情况 (工作集) ，区分私有内存与共享内存的占比，从而分析进程的实际资源消耗。

三、 实验步骤

本次实验使用 Visual Studio Code 作为开发环境，基于 C++ 语言实现。详细的代码实现可见附件 `main.cpp`。

3.1 准备工作与库配置

在实验初期，我们注意到实验所需的部分关键 API (如 `GetProcessMemoryInfo`) 并不包含在标准的 `kernel32.lib` 中，而是位于 `PSAPI` (Process Status API) 库。因此，除了引入必要的头文件外，还需在代码中显式链接 `psapi.lib`，以避免编译链接错误。

```
#pragma comment(lib, "psapi.lib") // 关键：链接静态库

#include <windows.h>
#include <psapi.h>
#include <iostream>
// ... 其他头文件
```

3.2 定义数据结构与辅助函数

由于 Windows API 返回的内存状态码均为数值常量 (如 `0x1000` 代表 `MEM_COMMIT`)，不便于直观阅读。为此，我们定义了 `RegionInfo` 结构体用于存储解析后的信息，并实现了 `StateToStr` 和 `ProtectToStr` 辅助函数，将数值状态映射为易读的字符串 (如 “Committed”, “Read-Write”) 。

```
struct RegionInfo {
    void* base;
    SIZE_T size;
    DWORD state;
    DWORD protect;
    DWORD type;
};

// 辅助函数：将状态宏转换为字符串
std::string StateToStr(DWORD state) {
    switch (state) {
        case MEM_COMMIT: return "Committed";
        case MEM_RESERVE: return "Reserved";
        case MEM_FREE: return "Free";
        default: return "Unknown";
    }
}

// ProtectToStr 实现逻辑类似，处理 PAGE_READONLY 等标志位
std::string ProtectToStr(DWORD prot) {
    switch (prot & 0xFF) {
        case PAGE_READONLY: return "R--";
        case PAGE_READWRITE: return "RW-";
        case PAGE_WRITECOPY: return "RWC";
        case PAGE_EXECUTE: return "--X";
        case PAGE_EXECUTE_READ: return "R-X";
        case PAGE_EXECUTE_READWRITE: return "RWX";
        case PAGE_EXECUTE_WRITECOPY: return "RWXC";
        default: return "---";
    }
}
```

3.3 获取系统边界

在进行内存遍历前，需要确定遍历的地址范围。通过调用 `GetSystemInfo` 函数，我们可以获取用户空间的有效起始地址（`lpMinimumApplicationAddress`）和结束地址（`lpMaximumApplicationAddress`）。这一步骤确保了遍历操作在合法的用户空间范围内进行。

```
void PrintSystemInfo() {
    SYSTEM_INFO si{};
    GetSystemInfo(&si); // 获取系统页大小和地址范围

    // ... 调用 GlobalMemoryStatusEx 获取物理内存信息

    std::cout << "Page size: " << si.dwPageSize
              << "  Min addr: " << si.lpMinimumApplicationAddress
              << "  Max addr: " << si.lpMaximumApplicationAddress << "\n";
}
```

3.4 核心遍历逻辑

程序采用“递推查询”的方式遍历整个地址空间：

1. 以 `MinAddr` 作为起始查询地址。
2. 调用 `VirtualQueryEx` 获取当前地址所在内存块的详细信息（`MEMORY_BASIC_INFORMATION`）。
3. 保存查询结果。
4. **关键步骤**：将查询地址累加当前区域的大小（`RegionSize`），作为下一次查询的基址。
5. 重复上述步骤，直至地址超出 `MaxAddr`。

```
std::vector<RegionInfo> EnumerateRegions(HANDLE process) {
    std::vector<RegionInfo> regions;
    SYSTEM_INFO si{};
    GetSystemInfo(&si);

    auto addr = reinterpret_cast<std::uintptr_t>(si.lpMinimumApplicationAddress);
    auto maxAddr = reinterpret_cast<std::uintptr_t>(si.lpMaximumApplicationAddress);
    MEMORY_BASIC_INFORMATION mbi{};

    // 循环遍历用户空间
    while (addr < maxAddr) {
        // 若查询失败（如权限不足或越界），则终止遍历
        if (VirtualQueryEx(process, reinterpret_cast<void*>(addr), &mbi, sizeof(mbi))
== 0)
            break;

        RegionInfo info{mbi.BaseAddress, mbi.RegionSize, mbi.State, mbi.Protect,
mbi.Type};
        regions.push_back(info);

        // 跳转至下一块内存区域
        addr += mbi.RegionSize;
    }
    return regions;
}
```

3.5 主函数与权限处理

在 `main` 函数中，程序根据用户输入的 PID 获取目标进程句柄。在实验过程中，我们发现必须在 `OpenProcess` 时显式请求 `PROCESS_QUERY_INFORMATION` 和 `PROCESS_VM_READ` 权限。若权限掩码设置不足，操作系统将拒绝后续的内存查询操作。

```
int main() {
    // ... 输入 PID ...

    // 请求查询与读取权限，否则 VirtualQueryEx 将调用失败
    HANDLE process = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
pid);

    if (!process) {
        std::cerr << "OpenProcess failed: " << GetLastError() << "\n";
        return 1;
    }

    PrintSystemInfo();
    auto regions = EnumerateRegions(process);
    // ... 输出结果 ...
    CloseHandle(process);
    return 0;
}
```

四、 实验结果及分析

本次实验选取了一个正在运行的进程 (PID 34708) 作为分析对象，程序的输出结果见附件 `output.txt`，具体结果分析如下：

4.1 地址空间的分布特征

观察输出结果，最显著的特征是**地址空间的稀疏性**。系统的最小应用地址为 `0x10000` (64KB)，这部分低地址空间的保留旨在防止空指针解引用引发的系统级错误。在整个地址空间中，绝大部分区域处于 `Free` 状态。例如，在地址 `0x2028ba20000` 处存在一个大小约为 135GB 的空闲块。这一现象证实了 64 位系统提供了极大的虚拟寻址能力，而实际被提交 (Committed) 或保留 (Reserved) 的内存仅占极小比例。

4.2 内存保护与安全性机制

通过分析 `Prot` (保护属性) 字段，可以清晰地观察到代码与数据的隔离机制：

- 标识为 `R-X` (可读可执行) 的区域通常对应代码段 (.text)，存放可执行指令。
- 标识为 `RW-` (可读写) 的区域通常对应数据段或堆栈，用于存放变量。

- 实验中几乎未观察到 `RWX` (可读写可执行) 区域，这表明操作系统默认启用了 DEP (数据执行保护) 机制，有效防止了缓冲区溢出攻击。
- 此外，部分区域标记为 `RWC` (Read-Write-Copy)，即“写时复制”。这验证了操作系统在处理进程派生或 DLL 加载时的优化策略：通过延迟物理内存分配，实现资源的高效利用。

4.3 物理内存占用分析

通过 `GetProcessMemoryInfo` 获取的数据显示，该进程的 Working Set (工作集) 为 4MB，但 Private Usage (私有内存) 仅为 1MB。 这意味着该进程实际独占的物理内存较少，约 3MB 的内存属于共享资源。这主要归因于系统 DLL (如 `kernel32.dll`) 的共享映射机制。操作系统在物理内存中仅保留一份动态库副本，即可映射至所有相关进程，从而显著提升了整体系统的内存利用率。

五、 实验收获与体会

本次实验通过代码实现与数据分析，将理论知识与系统实践进行了有效结合，主要收获如下:

1. **深入理解 Windows 权限安全模型** 在编码初期，因 `OpenProcess` 权限掩码设置不当，导致程序无法正确读取目标进程信息。通过解决这一问题，深刻理解了操作系统对进程对象的访问控制机制，以及在系统编程中处理安全描述符的重要性。
2. **实现理论知识的可视化验证** 实验将“虚拟内存”、“分页机制”等抽象概念转化为具体的可观测数据。通过 API 遍历得到的连续地址与离散属性，直观验证了虚拟地址空间的布局特征。特别是观察到巨大的 `Free` 空间，从实践角度验证了 64 位架构下地址空间的广阔性。
3. **提升系统文档查阅与 API 使用能力** 实验过程中涉及多个复杂 API 的调用。通过查阅 MSDN 文档，掌握了 `MEMORY_BASIC_INFORMATION` 结构体中各字段 (如 `MEM_IMAGE` vs `MEM_MAPPED`) 的具体含义，提升了阅读官方技术文档并将其应用于代码开发的能力。
4. **掌握位运算在系统编程中的应用** 在解析内存保护属性时，实际处理了由多个标志位组成的掩码。通过位运算 (`&` 操作) 提取具体属性的过程，体会到了操作系统设计中通过位字段节省空间、提高效率的工程思想。