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. **深入理解虚拟内存管理机制** 通过对 Windows 进程虚拟地址空间的遍历，掌握进程内存布局的结构特征，深入理解虚拟内存区域的状态（提交、保留、空闲）及其访问保护属性（只读、读写、执行等），从而强化对现代操作系统分页存储管理策略的认知。
2. **掌握 Windows 系统级编程技术** 熟悉并掌握 Windows API 中与内存管理相关的核心函数，包括 `GetSystemInfo`、`VirtualQueryEx`、`GlobalMemoryStatusEx` 及 `GetPerformanceInfo`，培养在 Windows 平台下进行系统级程序设计与调试的能力。
3. **分析进程工作集与系统性能** 通过获取进程的工作集（Working Set）信息及系统整体内存性能指标，分析操作系统在物理内存分配、页面置换及内存资源调度方面的行为模式，探究私有内存与共享内存的区别。
4. **验证操作系统理论知识** 将课堂所学的虚拟地址空间、页表映射、内存保护等理论知识与实际操作系统的行为进行对照，验证理论模型在工业级操作系统中的具体实现与应用。

二、 实验内容

本实验在 Windows 操作系统环境下，使用 C++ 语言编写控制台应用程序，调用 Windows API 实现对进程虚拟地址空间的遍历与分析。具体内容包括以下几个方面：

1. **获取系统内存配置与状态信息** 调用 `GetSystemInfo` 函数获取当前系统的硬件层级内存参数，包括系统的内存页大小（Page Size）、用户模式下的最小有效地址和最大有效地址。这些参数为后续的内存地址遍历划定了边界和粒度。同时，利用 `GlobalMemoryStatusEx` 函数获取系统的物理内存总量、当前可用物理内存及内存负载情况，建立对实验环境内存资源的宏观认识。
2. **实时遍历进程虚拟地址空间** 针对指定进程（通过进程 ID 标识），利用 `OpenProcess` 获取具有查询权限的进程句柄，并结合 `VirtualQueryEx` 函数对该进程的虚拟内存空间进行全量扫描。程序从系统允许的最小用户地址开始，循环调用查询接口，逐块获取内存区域（Memory Region）的详细属性，直至遍历至最大用户地址。解析的关键信息包括：
   - **内存区域基址与大小**：确定每一段连续内存块的起始位置和跨度。

- **内存状态**：识别该区域是处于已提交（`MEM_COMMIT`）、保留（`MEM_RESERVE`）还是空闲（`MEM_FREE`）状态。
- **保护属性**：分析该区域的访问权限，如只读（`PAGE_READONLY`）、读写（`PAGE_READWRITE`）或可执行（`PAGE_EXECUTE`）等。
- **物理存储类型**：区分内存块是私有数据（`MEM_PRIVATE`）、映射文件（`MEM_MAPPED`）还是可执行镜像（`MEM_IMAGE`）。

3. **分析进程工作集与系统性能指标** 使用 Windows PSAPI (Process Status API) 库中的 `GetProcessMemoryInfo` 函数，查询目标进程的工作集（Working Set）大小、私有内存使用量（Private Usage）以及页文件使用量，以量化进程对物理内存的实际占用。此外，调用 `GetPerformanceInfo` 函数获取系统范围的性能统计数据，如系统缓存大小、提交限制（Commit Limit）等，从而构建对进程及系统整体内存行为的完整视图。

三、 实验步骤

本实验采用 C++ 语言结合 Windows API 进行开发。实验的具体实现步骤如下：

3.1 开发环境配置与库文件引用

为了调用 Windows 操作系统提供的内存管理接口，程序需引入 `windows.h` 头文件，并包含进程状态库 `psapi.h`。由于部分系统调用（如 `GetProcessMemoryInfo`）位于 PSAPI 库中，需通过编译器指令链接 `psapi.lib` 静态库，以确保链接阶段能够正确解析符号。

```
#pragma comment(lib, "psapi.lib")

#include <windows.h>
#include <psapi.h>
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
#include <cstdint>
```

3.2 辅助功能与状态转换实现

Windows API 返回的内存状态（如 `MEM_COMMIT`）和保护属性（如 `PAGE_READONLY`）均为数值型常量，不易于直观阅读。为此，定义结构体 `RegionInfo` 用于存储提取后的关键信息，并实现 `StateToStr` 与 `ProtectToStr` 辅助函数，将数值型标志位映射为可读性较强的字符串描述，便于后续的格式化输出。

```
struct RegionInfo {
    void* base;
```

```
    SIZE_T size;
    DWORD state;
    DWORD protect;
    DWORD type;
};

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 函数实现逻辑同上，略
```

3.3 系统级内存信息获取

在对特定进程进行分析前，需先获取系统的全局内存参数。通过 `GetSystemInfo` 获得系统页大小及用户模式下的地址空间范围（最小与最大有效地址），这为后续的遍历操作提供了边界条件。同时，利用 `GlobalMemoryStatusEx` 和 `GetPerformanceInfo` 获取物理内存总量、可用量及系统提交限制等宏观性能指标。

```
void PrintSystemInfo() {
    SYSTEM_INFO si{};
    GetSystemInfo(&si);

    PERFORMANCE_INFORMATION perf{};
    perf.cb = sizeof(perf);
    GetPerformanceInfo(&perf, sizeof(perf));

    MEMORYSTATUSEX ms{};
    ms.dwLength = sizeof(ms);
    GlobalMemoryStatusEx(&ms);

    // 输出系统页大小、地址边界及物理内存统计信息
    std::cout << "Page size: " << si.dwPageSize
              << "  Min addr: " << si.lpMinimumApplicationAddress
              << "  Max addr: " << si.lpMaximumApplicationAddress << "\n";
    // ... (后续输出代码)
}
```

3.4 虚拟地址空间遍历算法

这是本实验的核心步骤。程序依据 `GetSystemInfo` 获取的最小应用地址初始化遍历指针，在一个循环结构中不断调用 `VirtualQueryEx` 函数。该函数接受进程句柄与当前基地址作为参数，返回一个 `MEMORY_BASIC_INFORMATION` 结构体，其中包含从当前基址开始的连续内存块的属性。

每次查询成功后，将该内存块的基址、大小、状态及保护属性存入 `RegionInfo` 容器。随后，遍历指针累加当前内存块的大小（`RegionSize`），指向下一块内存区域的起始位置，直至遍历指针超出系统的最大应用地址。

```cpp
std::vector<RegionInfo> EnumerateRegions(HANDLE process) {
    std::vector<RegionInfo> regions;
    SYSTEM_INFO si{};
    GetSystemInfo(&si);
    // 初始化遍历起始地址与结束边界
    auto addr = reinterpret_cast<std::uintptr_t>(si.lpMinimumApplicationAddress);
    auto maxAddr = reinterpret_cast<std::uintptr_t>(si.lpMaximumApplicationAddress);
    MEMORY_BASIC_INFORMATION mbi{};

    // 循环遍历整个虚拟地址空间
    while (addr < maxAddr) {
        if (VirtualQueryEx(process, reinterpret_cast<void*>(addr), &mbi, sizeof(mbi))
== 0)
            break; // 查询失败或遍历结束

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

        // 指针跳转至下一块区域
        addr += mbi.RegionSize;
    }
    return regions;
}
```

3.5 进程工作集查询与主控流程

除了虚拟地址空间的布局，实验还通过 `GetProcessMemoryInfo` 获取目标进程的工作集大小、私有字节数及页文件使用量，以反映进程对物理内存资源的实际占用情况。

在主函数 `main` 中，程序接收用户输入的进程标识符（PID）。若输入为 0，则调用 `GetCurrentProcessId` 获取自身 PID。随后，使用 `OpenProcess` 函数请求 `PROCESS_QUERY_INFORMATION` 和 `PROCESS_VM_READ` 权限以获取目标进程的句柄。在成功获取句柄后，依次调用上述功能模块，输出分析结果，并最终关闭句柄以释放资源。

```cpp
int main() {
    DWORD pid;
    std::cout << "Enter PID (0 for current process): ";
    std::cin >> pid;
    if (pid == 0) pid = GetCurrentProcessId();

    //以此权限打开进程是读取内存信息的前提
    HANDLE process = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
pid);

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

    // 执行各模块功能
    PrintSystemInfo();
    auto regions = EnumerateRegions(process);
    PrintWorkingSet(process);
    PrintRegions(regions);
```

```
20
21      CloseHandle(process);
22      return 0;
23  }
```

四、 实验结果及分析

本章节基于实验程序对 PID 为 34708 的进程及其所在系统的实际采样数据进行分析。

4.1 系统基础环境与资源概况

实验数据表明，当前系统的内存页大小（Page Size）为 4096 字节（4KB），这是 x64 架构下 Windows 系统的标准分页粒度。系统识别的最小用户应用地址为 $0x10000$（64KB），而非 $0x0$。这一机制在操作系统层面保留了低 64KB 的地址空间，旨在捕获空指针引用（Null Pointer Dereference）错误，防止程序因错误访问零地址附近的内存而导致系统不稳定。

最大应用地址显示为 $0x7ffffffeffff$，表明当前系统配置下的用户模式地址空间约为 128TB。这验证了在 64 位 Windows 系统中，虚拟地址空间被划分为巨大的用户分区和内核分区，且用户程序拥有极大的寻址范围。

4.2 进程工作集与物理内存占用

针对 PID 34708 的进程分析显示，其工作集（Working Set）总量为 4MB，其中私有工作集（Private Working Set）仅为 1MB。

这一数据差异揭示了现代操作系统内存共享机制的高效性。工作集总量包含了进程当前引用的所有物理内存页，而私有工作集仅包含该进程独占的数据。两者之间 3MB 的差值主要由共享内存页面构成，推测该进程加载了通用的动态链接库（如 `kernel32.dll`，`ntdll.dll` 等）。操作系统通过将这些常用库的代码段映射到多个进程的虚拟地址空间，并在物理内存中仅保留一份副本，从而显著降低了整体内存消耗。

4.3 虚拟地址空间布局特征

通过对 `VirtualQueryEx` 返回的区域列表进行解析，该进程的虚拟地址空间呈现出明显的离散化与分段特征：

1. **地址空间的稀疏性** 输出结果中存在大量状态为 `Free` 的内存区域，且尺寸巨大。例如，地址 $0x2028ba20000$ 处存在一个大小约为 135GB 的空闲块。这证实了在 64 位地址空间中，已提交（`Committed`）和已保留（`Reserved`）的内存仅占极小比例，绝大部分地址空间处于未分配状态，为大内存应用（如数据库、科学计算）预留了充足的扩展能力。

2. **内存保护属性与代码隔离** 内存区域的保护属性（Protect）严格遵循了数据与代码分离的原则：
   - **可执行区域**：如地址 $0x7fff7e2f91000$ 处的 16KB 区域，属性为 `R-X`（可读、可执行）。此区域通常对应进程的主代码段（`.text` 段），操作系统禁止对其进行写操作以防止代码篡改。
   - **读写数据区域**：如地址 $0x7fff7e2f99000$ 处的 4KB 区域，属性为 `RW-`（可读写、不可执行）。这通常对应全局变量或堆栈数据，禁止执行权限可有效防御缓冲区溢出攻击（DEP 机制）。
   - **写时复制机制**：部分区域如 $0x7fff7e2f95000$ 标记为 `RWC`（Read-Write-Copy）。这表明该页面可能属于数据段的初始映射，当进程试图修改此页面时，操作系统将触发缺页中断并分配新的物理页，从而实现进程间的私有化修改。
3. **系统共享页面的映射** 在地址 $0x7fffe0000$ 处观察到一个属性为 `R--`（只读）的 `Committed` 页面。这是 Windows 系统中典型的 `KUSER_SHARED_DATA` 结构映射地址。操作系统内核利用该固定地址向所有用户进程暴露系统时间、版本号等只读数据，使得用户进程无需陷入内核态即可获取常用系统信息，从而提高了系统调用的执行效率。

4.4 实验结论

本实验成功利用 Windows API 遍历了目标进程的虚拟地址空间。实验结果直观地验证了分页存储管理中的关键概念：虚拟地址到物理地址的非连续映射、内存区域的权限控制机制（如 NX 位支持）、以及写时复制与共享库技术在节省物理内存方面的实际应用。实验数据与操作系统内存管理理论模型高度一致。

五、 实验收获与体会

通过本次遍历进程地址空间的实验，从代码实现到数据分析的过程不仅强化了对操作系统内存管理理论的理解，也在系统级编程方面获得了实质性的技术积累。

1. **理论与实践的深度融合** 实验将课堂上抽象的“虚拟内存”、“分页机制”及“段页式管理”概念具体化为可观测的系统行为。通过直接观察 `VirtualQueryEx` 返回的内存块状态，深刻体会了操作系统如何通过页表映射机制，向用户进程提供一个连续且私有的虚拟地址空间假象，而底层物理内存实际是离散且共享的。特别是观察到大量 `Free` 状态的内存区域，直观验证了 64 位架构下虚拟地址空间的稀疏性特征，理解了为何现代操作系统能够支持远超物理内存容量的寻址范围。
2. **系统级编程能力的提升** 在技术层面，掌握了 Windows 内存管理核心 API 的调用范式。实验过程中解决了权限控制问题，理解了在调用 `OpenProcess` 时必须明确指定 `PROCESS_VM_READ` 和 `PROCESS_QUERY_INFORMATION` 权限掩码，否则操作系统将基于安全策略拒绝访问。此外，通过解析

`MEMORY_BASIC_INFORMATION` 结构体中的位字段（Bit Field），提升了处理底层系统数据结构的能力，学会了如何将晦涩的十六进制标志位转化为具有可读性的状态描述。

3. **对内存保护与系统安全的认知** 通过分析内存区域的保护属性（如 `PAGE_EXECUTE_READ` 与 `PAGE_READWRITE` 的严格区分），深入理解了操作系统的数据执行保护（DEP）机制。代码段与数据段的权限隔离是防止缓冲区溢出攻击的基石。同时，在实验结果中观测到的 `PAGE_WRITECOPY`（写时复制）属性，揭示了操作系统在多进程运行环境下，通过延迟物理内存分配来优化资源利用率的核心策略。

4. **共享内存机制的实证** 实验数据中“私有工作集”与“总工作集”的显著差异，提供了共享库（Shared Libraries）机制存在的直接证据。这一发现证实了操作系统通过在物理内存中仅保留一份常用 DLL（如 `kernel32.dll`）副本，并将其映射到不同进程虚拟地址空间的技术路径，有效解释了多任务环境下系统内存资源的高效复用原理。