Память

Виды памяти. Сегментная. Страничная. Сегментно-страничная. Виртуальная.

Описаня виды памяти

Сегментная адресация памяти

Сегментная адресация памяти - схема логической адресации памяти компьютера в архитектуре `x86`. Линейный адрес конкретной ячейки памяти, который в некоторых режимах работы процессора будет совпадать с физическим адресом, делится на две части: сегмент и смещение. Сегментом называется условно выделенная область адресного пространства определённого размера, а смещением - адрес ячейки памяти относительно начала сегмента. Базой сегмента называется линейный адрес (адрес относительно всего объёма памяти), который указывает на начало сегмента в адресном пространстве. В результате получается сегментный (логический) адрес, который соответствует линейному адресу база сегмента +смещение и который выставляется процессором на шину адреса.

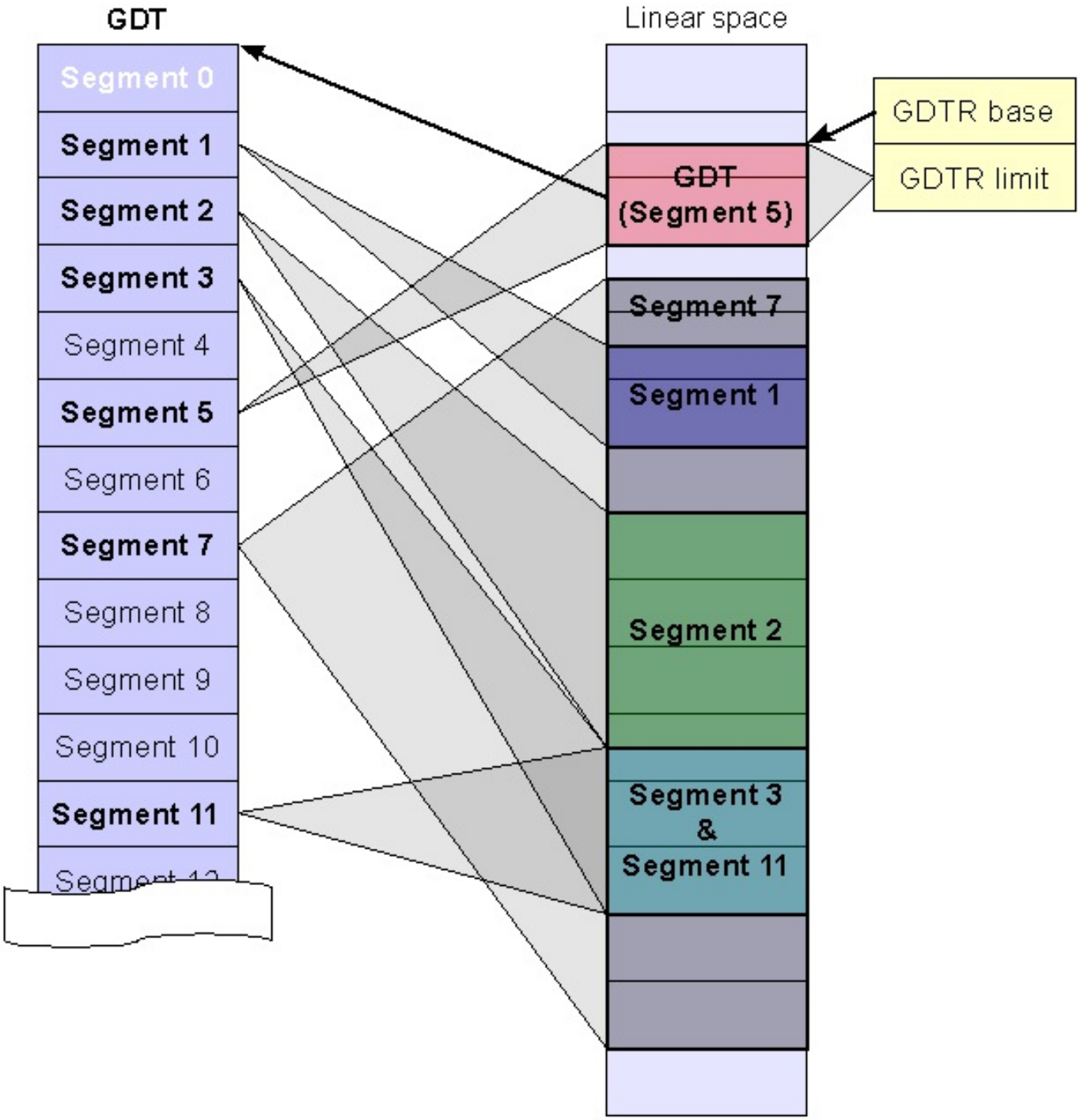
Селектором называется число (в `x86` — 16-битное), однозначно определяющее сегмент. Селектор загружается в сегментные регистры.

В реальном и защищённом режимах `x86`-процессора функционирование сегментной адресации отличается.

В реальном режиме процессора всё адресное пространство делится на одинаковые сегменты размером 65536 байт. Начало каждого последующего сегмента (так называемая База сегмента) смещена относительно базы предыдущего на минимальный размер сегмента, то есть на 16 байт (т. н. параграф). Таким образом, сегменты могут частично перекрывать друг друга. (Например, байт 17 сегмента 2 - это также и байт `1 = 17 - 16` сегмента 3, и байт `33 = 17 + 16` сегмента 1.)

Селектор 16-разрядный и задаёт номер сегмента. Учитывая, что сегменты следуют друг за другом с постоянным интервалом в `2^4` = 16 байт, очень легко выяснить линейный адрес сегмента, умножая его на 16.

В защищённом режиме процессора адресное пространство задачи делится на сегменты различных размеров с различными базами. Для определения базы и размера сегментов служат дескрипторы сегментов, хранящиеся в дескрипторных таблицах (`GDT` и `LDT`).



Здесь сегменты 3 и 11 указывают на одну и ту же область и являются псевдонимами (`Alias`). Сегмент 7 охватывает сегменты 1, 2, 3 и 11. Сегмент 5 указывает на `GDT`, позволяя её изменять (это никак не относится к `GDT` - её настоящий дескриптор хранится в регистре `GDTR` (показан жёлтым)). Адресация через локальную таблицу дескрипторов (`LDT`) происходит аналогично.

Селектор также 16-разрядный, но делится на три части: `RPL` (биты 0 - 1), `TI` (бит 2) и номер дескриптора (биты 3 - 15).

- `RPL` - Сегментная защита памяти;
- `TI` определяет дескрипторную таблицу (`GDT` или `LDT` при 0 или 1 соответственно), из которой выбирается дескриптор;
- номер дескриптора - порядковый номер в дескрипторной таблице. Так как размер дескриптора равен восьми байтам, а номер дескриптора начинается с третьего бита, то можно адресовать дескриптор (если надо), просто обнулив `RPL` и `TI`.

Страничная память

Страничная память - способ организации виртуальной памяти, при котором единицей отображения виртуальных адресов на физические является регион постоянного размера (т. н. страница). Типичный размер страницы — 4096 байт, для некоторых архитектур — до 128 КБ.

Поддержка такого режима присутствует в большинстве 32-битных и 64-битных процессоров. Такой режим является классическим для почти всех современных ОС, в том числе `Windows` и семейства `UNIX`. Широкое использование такого режима началось с процессора `VAX` и `ОС VMS` с конца 1970-х годов (по некоторым сведениям, первая реализация). В семействе `x86` поддержка появилась с поколения `386`, оно же первое 32-битное поколение.

Оперативная память делится на страницы: области памяти фиксированной длины (например, 4096 байт), которые являются

минимальной единицей выделяемой памяти (то есть даже запрос на 1 байт от приложения приведёт к выделению ему страницы памяти). Исполняемый процессором пользовательский поток обращается к памяти с помощью адреса виртуальной памяти, который делится на номер страницы и смещение внутри страницы. Процессор преобразует номер виртуальной страницы в адрес соответствующей ей физической страницы при помощи буфера ассоциативной трансляции (`TLB`). Если ему не удалось это сделать, то требуется дозаполнение буфера путём обращения к таблице страниц (так называемый `Page Walk`), что может сделать либо сам процессор, либо операционная система (в зависимости от архитектуры). Если страница была выгружена из оперативной памяти, то операционная система подкачивает страницу с жёсткого диска в ходе обработки события `Page fault`. При запросе на выделение памяти операционная система может “сбросить” на жёсткий диск страницы, к которым давно не было обращений. Критические данные (например, код запущенных и работающих программ, код и память ядра системы) обычно находятся в оперативной памяти (исключения существуют, однако они не касаются тех частей, которые отвечают за обработку аппаратных прерываний, работу с таблицей страниц и использование файла подкачки).

Огромным достоинством страничной виртуальной памяти по сравнению с сегментной является отсутствие “ближних” и “дальних” указателей.

Наличие таких концепций в программировании уменьшает применимость арифметики указателей и приводит к огромным проблемам с переносимостью кода с/на такие архитектуры. Так,

например, значительная часть ПО с открытым кодом изначально разрабатывалась для бессегментных 32-битных платформ со страничной памятью и не может быть перенесена на сегментные архитектуры без серьёзной переработки.

Кроме того, сегментные архитектуры имеют тяжелейшую проблему `SS != DS`, широко известную в начале 1990-х годов в программировании под 16-битные версии `Windows`. Эта проблема приводит к затруднениям в реализации динамических библиотек, ибо они имеют свой собственный `DS`, и `SS` текущего процесса, что приводит к невозможности использования “ближних” указателей в них. Также наличие своего собственного `DS` в библиотеках требует устанавливающих правильное значение `DS` заплаток (`MakeProcInstance`) для обратных вызовов из библиотеки в вызвавшее приложение.

Отображаемые в память файлы

Основная статья: Отображение файла на память
Обработчик отказа страницы в ядре способен прочитать данную страницу из файла.

Это приводит к возможности лёгкой реализации отображенных в память файлов. Концептуально это то же, что выделение памяти и чтение в неё отрезка файла, с той разницей, что чтение осуществляется неявно “по требованию”, выраженному отказом страницы при попытке обращения к ней.

Вторым преимуществом такого подхода является - в случае отображения “только для чтения” - разделение одной и той же физической памяти между всеми процессами, отображающими данный файл (выделенная же память своя у каждого процесса).

Третьим преимуществом является возможность “забывания” (`discard`) некоторых отображенных страниц без выгрузки их в область подкачки, обязательной для выделенной памяти. В случае повторной потребности в странице она может быть быстро загружена из файла снова.

Четвёртым преимуществом является неиспользование дискового кэша в этом режиме, что означает экономию на копировании данных из кэша в запрошенный регион. Преимущества дискового кэша, оптимизирующего операции небольшого размера, а также повторное чтение одних и тех же данных, полностью исчезают при чтениях целых страниц и тем более их групп, недостаток же в виде обязательного лишнего копирования - сохраняется.

Отображаемые в память файлы используется в ОС `Windows`, а также ОС семейства `UNIX`, для загрузки исполняемых модулей и динамических библиотек. Они же используются утилитой `GNU grep` для чтения входящего файла, а также для загрузки шрифтов в ряде графических подсистем.

Пример для `Windows API` (CreateFileMapping)

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

#define BUF_SIZE 65536

TCHAR szName[] = TEXT("LARGEPAGE");
typedef int (*GETLARGEPAGEMINIMUM)(void);

void DisplayError(TCHAR* pszAPI, DWORD dwError) {
    LPVOID lpvMessageBuffer;

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM |
                  FORMAT_MESSAGE_IGNORE_INSERTS,
                  NULL, dwError,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPTSTR)&lpvMessageBuffer, 0, NULL);

    _tprintf(TEXT("ERROR: API        = %s\n"), pszAPI);
    _tprintf(TEXT("       error code = %d\n"), dwError);
    _tprintf(TEXT("       message    = %s\n"), lpvMessageBuff
er);

    LocalFree(lpvMessageBuffer);
    ExitProcess(GetLastError());
}
```

```
void Privilege(TCHAR* pszPrivilege, BOOL bEnable) {
    HANDLE           hToken;
    TOKEN_PRIVILEGES tp;
    BOOL             status;
    DWORD            error;

    // open process token
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_P
RIVILEGES | TOKEN_QUERY, &hToken))
        DisplayError(TEXT("OpenProcessToken"), GetLastError()
);

    // get the luid
    if (!LookupPrivilegeValue(NULL, pszPrivilege, &tp.Privile
ges[0].Luid))
        DisplayError(TEXT("LookupPrivilegeValue"), GetLastErr
or());

    tp.PrivilegeCount = 1;

    // enable or disable privilege
    if (bEnable)
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    else
        tp.Privileges[0].Attributes = 0;

    // enable or disable privilege
```

```
    status = AdjustTokenPrivileges(hToken, FALSE, &tp, 0, (PT
OKEN_PRIVILEGES)NULL, 0);

    // It is possible for AdjustTokenPrivileges to return TRU
E and still not succeed.
    // So always check for the last error value.
    error = GetLastError();
    if (!status || (error != ERROR_SUCCESS))
        DisplayError(TEXT("AdjustTokenPrivileges"), GetLastEr
ror());

    // close the handle
    if (!CloseHandle(hToken))
        DisplayError(TEXT("CloseHandle"), GetLastError());
}

void _tmain(void) {
    HANDLE hMapFile;
    LPCTSTR pBuf;
    DWORD size;
    GETLARGEPAGEMINIMUM pGetLargePageMinimum;
    HINSTANCE  hDll;

    // call succeeds only on Windows Server 2003 SP1 or later
    hDll = LoadLibrary(TEXT("kernel32.dll"));
    if (hDll == NULL)
        DisplayError(TEXT("LoadLibrary"), GetLastError());
```

```
    pGetLargePageMinimum = (GETLARGEPAGEMINIMUM)GetProcAddres
s(hDll,
        "GetLargePageMinimum");
    if (pGetLargePageMinimum == NULL)
        DisplayError(TEXT("GetProcAddress"), GetLastError());

    size = (*pGetLargePageMinimum)();

    FreeLibrary(hDll);

    _tprintf(TEXT("Page Size: %u\n"), size);

    Privilege(TEXT("SeLockMemoryPrivilege"), TRUE);

    hMapFile = CreateFileMapping(
         INVALID_HANDLE_VALUE,    // use paging file
         NULL,                    // default security
         PAGE_READWRITE | SEC_COMMIT | SEC_LARGE_PAGES,
         0,                       // max. object size
         size,                    // buffer size
         szName);                 // name of mapping object

    if (hMapFile == NULL)
        DisplayError(TEXT("CreateFileMapping"), GetLastError(
));
    else
        _tprintf(TEXT("File mapping object successfulyl creat
ed.\n"));
```

```
    Privilege(TEXT("SeLockMemoryPrivilege"), FALSE);

    pBuf = (LPTSTR) MapViewOfFile(hMapFile,   // handle to ma
p object
          FILE_MAP_ALL_ACCESS, // read/write permission
          0,
          0,
          BUF_SIZE);

    if (pBuf == NULL)
        DisplayError(TEXT("MapViewOfFile"), GetLastError());
    else
        _tprintf(TEXT("View of file successfully mapped.\n"))
;

    // do nothing, clean up an exit
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
}
```

Плоская модель памяти

Метод организации адресного пространства оперативной памяти вычислительных устройств. В плоской модели код и данные используют одно и то же адресное пространство. Для 16-битных процессоров плоская модель памяти позволяет адресовать 64 кБ оперативной памяти; для 32-битных процессоров 4 ГБ, для 64-битных

— до 16 эксабайт (для `amd64` размер ограничен 256 ТБ).

Управление памятью все ещё реализуется на основе плоской модели, в целях содействия функциональности операционной системы, защиты ресурсов, многозадачности или увеличения объёма памяти за пределы ограничений, налагаемых физическим адресным пространством процессора.

Преимущества управления памятью с плоской моделью:

1. В одном из многозадачных встроенных приложений, где управление памятью не нужно и не желательно, модель обеспечивает простейший интерфейс для программирования, с прямым доступом ко всем местам в памяти и минимальной сложностью конструкции программы.
2. При многозадачности и распределении ресурсов плоская модель по-прежнему обеспечивает максимальную гибкость для реализации этого типа управления памятью.

Виртуальная память

Виртуальная память - метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между основной памятью и вторичным хранилищем (например, жёстким диском). Для выполняющейся программы данный метод полностью прозрачен и не требует дополнительных усилий со стороны программиста, однако реализация этого метода требует как аппаратной поддержки, так и

поддержки со стороны операционной системы.

В системе с виртуальной памятью используемые программами адреса, называемые виртуальными адресами, транслируются в физические адреса в памяти компьютера. Трансляцию виртуальных адресов в физические выполняет аппаратное обеспечение, называемое блоком управления памятью. Для программы основная память выглядит как доступное и непрерывное адресное пространство либо как набор непрерывных сегментов, вне зависимости от наличия у компьютера соответствующего объёма оперативной памяти. Управление виртуальными адресными пространствами, соотнесение физической и виртуальной памяти, а также перемещение фрагментов памяти между основным и вторичным хранилищами выполняет операционная система.

Применение виртуальной памяти позволяет:

1. освободить программиста от необходимости вручную управлять загрузкой частей программы в память и согласовывать использование памяти с другими программами
2. предоставлять программам больше памяти, чем физически установлено в системе
3. в многозадачных системах изолировать выполняющиеся программы друг от друга путём назначения им непересекающихся адресных пространств

В настоящее время виртуальная память аппаратно поддерживается в большинстве современных процессоров. В то же время в

микроконтроллерах и в системах специального назначения, где требуется либо очень быстрая работа, либо есть ограничения на длительность отклика (системы реального времени), виртуальная память используется относительно редко. Также в таких системах реже встречается многозадачность и сложные иерархии памяти.

Типы памяти. Статическая, динамическая, стек.

Статическая

```
int variable = 10;
```

Динамическая

```
int *variable = calloc(sizeof(struct Person), 1);
```

Стековая

```
{
  int valiable = 0;
}
```

Получение доступа к сегментам исполняемого модуля.

Пример для `Microsoft Compiler`

```
#pragma code_seg(".my_data1")
void func2() {}                    // stored in my_data1

#pragma data_seg(".my_data1")
int d = 0;
```

Работа с памятью в Си

Указатели

Представление памяти.

Опасности. Виды атак. Утечка памяти.