MMU

MMU 是 Memory Management Unit 的缩写，中文名是内存管理单元，它是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路，同时也负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权。

一、MMU 的历史

许多年以前，当人们还在使用 DOS 或是更古老的操作系统的时候，计算机的内存还非常小，一般都是以 K 为单位进行计算，相应的，当时的程序规模也不大，所以内存容量虽然小，但还是可以容纳当时的程序。但随着图形界面的兴起还用用户需求的不断增大，应用程序的规模也随之膨胀起来，终于一个难题出现在程序员的面前，那就是应用程序太大以至于内存容纳不下该程序，通常解决的办法是把程序分割成许多称为覆盖块（overlay）的片段。覆盖块 0 首先运行，结束时他将调用另一个覆盖块。虽然覆盖块的交换是由 OS 完成的，但是必须先由程序员把程序先进行分割，这是一个费时费力的工作，而且相当枯燥。人们必须找到更好的办法从根本上解决这个问题。不久人们找到了一个办法，这就是虚拟存储器(virtual memory).虚拟存储器的基本思想是程序，数据，堆栈的总的大小可以超过物理存储器的大小，操作系统把当前使用的部分保留在内存中，而把其他未被使用的部分保存在磁盘上。比如对一个 16MB 的程序和一个内存只有 4MB 的机器，操作系统通过选择，可以决定各个时刻将哪 4M 的内容保留在内存中，并在需要时在内存和磁盘间交换程序片段，这样就可以把这个 16M 的程序运行在一个只具有 4M 内存机器上了。而这个 16M 的程序在运行前不必由程序员进行分割。

二、MMU 的相关概念——地址范围、虚拟地址映射为物理地址 以及 分页机制

任何时候，计算机上都存在一个程序能够产生的地址集合，我们称之为地址范围。这个范围的大小由 CPU 的位数决定，例如一个 32 位的 CPU，它的地址范围是 0~0xFFFFFFFF (4G),而对于一个 64 位的 CPU，它的地址范围为 0~0xFFFFFFFFFFFFFFFF (64T).这个范围就是我们的程序能够产生的地址范围，我们把这个地址范围称为虚拟地址空间，该空间中的某一个地址我们称之为虚拟地址。与虚拟地址空间和虚拟地址相对应的则是物理地址空间和物理地址，大多数时候我们的系统所具备的物理地址空间只是虚拟地址空间的一个子集。这里举一个最简单的例子直观地说明这两者，对于一台内存为 256M 的 32bit x86 主机来说，它的虚拟地址空间范围是 0~0xFFFFFFFF（4G）,而物理地址空间范围是 0x000000000~0x0FFFFFFF（256M）。

在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写；而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存地址总线上，而是送到存储器管理单元 MMU，把虚拟地址映射为物理地址。

大多数使用虚拟存储器的系统都使用一种称为分页（paging）机制。虚拟地址空间划分成称为页（page）的单位,而相应的物理地址空间也被进行划分，单位是页桢(frame).页和页桢的大小必须相同。在这个例子中我们有一台可以生成 32 位地址的机

器，它的虚拟地址范围从 0~0xFFFFFFFF（4G）,而这台机器只有 256M 的物理地址，因此他可以运行 4G 的程序，但该程序不能一次性调入内存运行。这台机器必须有一个达到可以存放 4G 程序的外部存储器（例如磁盘或是 FLASH）,以保证程序片段在需要时可以被调用。在这个例子中，页的大小为 4K,页桢大小与页相同——这点是必须保证的，因为内存和外围存储器之间的传输总是以页为单位的。对应 4G 的虚拟地址和 256M 的物理存储器，他们分别包含了 1M 个页和 64K 个页桢。

三、MMU 的功能

1、将虚拟地址映射为物理地址

现代的多用户多进程操作系统，需要 MMU，才能达到每个用户进程都拥有自己独立的地址空间的目标。使用 MMU，操作系统划分出一段地址区域，在这块地址区域中，每个进程看到的内容都不一定一样。例如 MICROSOFT WINDOWS 操作系统将地址范围 4M-2G 划分为用户地址空间，进程 A 在地址 0X400000（4M）映射了可执行文件，进程 B 同样在地址 0X400000（4M）映射了可执行文件，如果 A 进程读地址 0X400000，读到的是 A 的可执行文件映射到 RAM 的内容，而进程 B 读取地址 0X400000 时，则读到的是 B 的可执行文件映射到 RAM 的内容。

这就是 MMU 在当中进行地址转换所起的作用。

2、提供硬件机制的内存访问授权

多年以来，微处理器一直带有片上存储器管理单元(MMU)，MMU 能使单个软件线程工作于硬件保护地址空间。但是在许多商用实时操作系统中，即使系统中含有这些硬件也没采用 MMU。

当应用程序的所有线程共享同一存储器空间时，任何一个线程将有意或无意地破坏其它线程的代码、数据或堆栈。异常线程甚至可能破坏内核代码或内部数据结构。例如线程中的指针错误就能轻易使整个系统崩溃，或至少导致系统工作异常。

就安全性和可靠性而言，基于进程的实时操作系统(RTOS)的性能更为优越。为生成具有单独地址空间的进程，RTOS 只需要生成一些基于 RAM 的数据结构并使 MMU 加强对这些数据结构的保护。基本思路是在每个关联转换中“接入”一组新的逻辑地址。MMU 利用当前映射，将在指令调用或数据读写过程中使用的逻辑地址映射为存储器物理地址。MMU 还标记对非法逻辑地址进行的访问，这些非法逻辑地址并没有映射到任何物理地址。

这些进程虽然增加了利用查询表访问存储器所固有的系统开销，但其实现的效益很高。在进程边界处，疏忽或错误操作将不会出现，用户接口线程中的缺陷并不会导致其它更关键线程的代码或数据遭到破坏。目前在可靠性和安全性要求很高的复杂嵌入式系统中，仍然存在采无存储器保护的操作系统的情况，这实在有些不可思议。

采用 MMU 还有利于选择性地将页面映射或解映射到逻辑地址空间。物理存储器页面映射至逻辑空间，以保持当前进程的代码，其余页面则用于数据映射。类似地，物理存储器页面通过映射可保持进程的线程堆栈。RTOS 可以在每个线程堆栈解映射之后，很容易地保留逻辑地址所对应的页面内容。这样，如果任何线程分配的堆栈发生溢出，将产生硬件存储器保护故障，内核将挂起该线程，而不使其破坏位于该地址

空间中的其它重要存储器区，如另一线程堆栈。这不仅在线程之间，还在同一地址空间之间增加了存储器保护。

存储器保护(包括这类堆栈溢出检测)在应用程序开发中通常非常有效。采用了存储器保护，程序错误将产生异常并能被立即检测，它由源代码进行跟踪。如果没有存储器保护，程序错误将导致一些细微的难以跟踪的故障。实际上，由于在扁平存储器模型中，RAM 通常位于物理地址的零页面，因此甚至 NULL 指针引用的解除都无法检测到。

四、MMU 和 CPU

1、X86 系列的 MMU

INTEL 出品的 80386CPU 或者更新的 CPU 中都集成有 MMU. 可以提供 32BIT 共 4G 的地址空间.

X86 MMU 提供的寻址模式有 4K/2M/4M 的 PAGE 模式(根据不同的 CPU，提供不同的能力), 此处提供的是目前大部分操作系统使用的 4K 分页机制的描述，并且不提供 ACCESS CHECK 的部分。

涉及的寄存器

a) GDT

b) LDT

c) CR0

d) CR3

e) SEGMENT REGISTER

虚拟地址到物理地址的转换步骤

a) SEGMENT REGISTER 作为 GDT 或者 LDT 的 INDEX, 取出对应的 GDT/LDT ENTRY.

注意: SEGMENT 是无法取消的, 即使是 FLAT 模式下也是如此. 说 FLAT 模式下不使用 SEGMENT REGISTER 是错误的. 任意的 RAM 寻址指令中均有 DEFAULT 的 SEGMENT 假定. 除非使用 SEGMENT OVERRIDE PREFIX 来改变当前寻址指令的 SEGMENT, 否则使用的就是 DEFAULT SEGMENT.

ENTRY 格式

```
typedef struct
{
UINT16 limit_0_15;
UINT16 base_0_15;
UINT8 base_16_23;
UINT8 accessed : 1;
UINT8 readable : 1;
UINT8 conforming : 1;
UINT8 code_data : 1;
UINT8 app_system : 1;
```

```
UINT8 dpl : 2;
UINT8 present : 1;
UINT8 limit_16_19 : 4;
UINT8 unused : 1;
UINT8 always_0 : 1;
UINT8 seg_16_32 : 1;
UINT8 granularity : 1;
UINT8 base_24_31;
} CODE_SEG_DESCRIPTOR,*PCODE_SEG_DESCRIPTOR;
typedef struct
{
UINT16 limit_0_15;
UINT16 base_0_15;
UINT8 base_16_23;
UINT8 accessed : 1;
UINT8 writeable : 1;
UINT8 expanddown : 1;
UINT8 code_data : 1;
UINT8 app_system : 1;
UINT8 dpl : 2;
UINT8 present : 1;
UINT8 limit_16_19 : 4;
UINT8 unused : 1;
UINT8 always_0 : 1;
UINT8 seg_16_32 : 1;
UINT8 granularity : 1;
UINT8 base_24_31;
} DATA_SEG_DESCRIPTOR,*PDATA_SEG_DESCRIPTOR;
```

共有 4 种 ENTRY 格式，此处提供的是 CODE SEGMENT 和 DATA SEGMENT 的 ENTRY 格式. FLAT 模式下的 ENTRY 在 base_0_15, base_16_23 处为 0，而 limit_0_15, limit_16_19 处为 0xfffff. granularity 处为 1. 表名 SEGMENT 地址空间是从 0 到 0XFFFFFFFF 的 4G 的地址空间.

b) 从 SEGMENT 处取出 BASE ADDRESS 和 LIMIT. 将要访问的 ADDRESS 首先进行 ACCESS CHECK，是否超出 SEGMENT 的限制.

c) 将要访问的 ADDRESS+BASE ADDRESS，形成需要 32BIT 访问的虚拟地址. 该地址被解释成如下格式:

```
typedef struct
{
```

UINT32 offset :12;
UINT32 page_index :10;
UINT32 pdbr_index :10;
} VA,*LPVA;

d) pdbr_index 作为 CR3 的 INDEX，获得到一个如下定义的数据结构

typedef struct
{
UINT8 present :1;
UINT8 writable :1;
UINT8 supervisor :1;
UINT8 writethrough:1;
UINT8 cachedisable:1;
UINT8 accessed :1;
UINT8 reserved1 :1;
UINT8 pagesize :1;
UINT8 ignoreed :1;
UINT8 avl :3;
UINT8 ptadr_12_15 :4;
UINT16 ptadr_16_31;
}PDE,*LPPDE;

e) 从中取出 PAGE TABLE 的地址．并且使用 page_index 作为 INDEX，得到如下数据结构

typedef struct
{
UINT8 present :1;
UINT8 writable :1;
UINT8 supervisor :1;
UINT8 writethrough:1;
UINT8 cachedisable:1;
UINT8 accessed :1;
UINT8 dirty :1;
UINT8 pta :1;
UINT8 global :1;
UINT8 avl :3;
UINT8 ptadr_12_15 :4;
UINT16 ptadr_16_31;
}PTE,*LPPTE;

f) 从 PTE 中获得 PAGE 的真正物理地址的 BASE ADDRESS. 此 BASE ADDRESS 表名了物理地址的.高 20 位. 加上虚拟地址的 offset 就是物理地址所在了.

2、ARM 系列的 MMU

ARM 出品的 CPU，MMU 作为一个协处理器存在。根据不同的系列有不同搭配。需要查询 DATASHEET 才可知道是否有 MMU。如果有的话，一定是编号为 15 的协处理器。可以提供 32BIT 共 4G 的地址空间。

ARM MMU 提供的分页机制有 1K/4K/64K 3 种模式. 本文介绍的是目前操作系统通常使用的 4K 模式。

涉及的寄存器, 全部位于协处理器 15.

ARM 没有 SEGMENT 的寄存器，是真正的 FLAT 模式的 CPU。给定一个 ADDRESS，该地址可以被理解为如下数据结构:

```
typedef struct
{
UINT32 offset :12;
UINT32 page_index :8;
UINT32 pdbr_index :12;
} VA,*LPVA;
```

从 MMU 寄存器 2 中取出 BIT14-31，pdbr_index 就是这个表的索引，每个入口为 4BYTE 大小，结构为

```
typedef struct
{
UINT32 type :2; //always set to 01b
UINT32 writebackcacheable:1;
UINT32 writethroughcacheable:1;
UINT32 ignore :1; //set to 1b always
UINT32 domain :4;
UINT32 reserved :1; //set 0
UINT32 base_addr:22;
} PDE,*LPPDE;
```

获得的 PDE 地址，获得如下结构的 ARRAY，用 page_index 作为索引，取出内容。

```
typedef struct
{
UINT32 type :2; //always set to 11b
UINT32 ignore :3; //set to 100b always
UINT32 domain :4;
UINT32 reserved :3; //set 0
UINT32 base_addr:20;
```

} PTE,*LPPTE;

从 PTE 中获得的基地址和上 offset，组成了物理地址.

PDE/PTE 中其他的 BIT，用于访问控制。这边讲述的是一切正常，物理地址被正常组合出来的状况。

ARM/X86 MMU 使用上的差异

1、X86 始终是有 SEGMENT 的概念存在. 而 ARM 则没有此概念(没有 SEGMENT REGISTER.).

2、ARM 有个 DOMAIN 的概念. 用于访问授权. 这是 X86 所没有的概念. 当通用 OS 尝试同时适用于此 2 者的 CPU 上, 一般会抛弃 DOMAIN 的使用.