第八章 I/O操作的实现

4. 第 3 题中用户程序的功能可以用以下 C 语言代码来实现：

```
1  int main()
2  {
3      write(1, "Hello, world.\n", 14);
4      exit(0);
5  }
```

针对上述 C 代码，回答下列问题或完成下列任务。

（1）执行 write()函数时，传递给 write()的实参在 main 栈帧中的存放情况怎样？要求画图说明。

（2）从执行 write()函数开始到调出 write 系统调用服务例程 sys_write()执行的过程中，其函数调用关系是怎样的？要求画图说明。

（3）就程序设计的便捷性和灵活性以及程序执行性能等方面，与第 3 题中的实现方式进行比较，并说明哪个执行时间更短？

参考答案：

（1）用“objdump -d”命令将对应的可执行文件反汇编后，其 main()函数对应的机器级指令序列如下图所示（左图是动态链接生成，右图是静态链接生成）。

```
0804844c <main>:
 804844c:       55                      push   %ebp
 804844d:       89 e5                   mov    %esp,%ebp
 804844f:       83 e4 f0                and    $0xfffffff0,%esp
 8048452:       83 ec 10                sub    $0x10,%esp
 8048455:       c7 44 24 08 0e 00 00    movl   $0xe,0x8(%esp)
 804845c:       00
 804845d:       c7 44 24 04 10 85 04    movl   $0x8048510,0x4(%esp)
 8048464:       08
 8048465:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 804846c:       e8 df fe ff ff          call   8048350 <write@plt>
 8048471:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048478:       e8 b3 fe ff ff          call   8048330 <exit@plt>
 804847d:       90                      nop
 804847e:       90                      nop
 804847f:       90                      nop
```

```
08048254 <main>:
 8048254:       55                      push   %ebp
 8048255:       89 e5                   mov    %esp,%ebp
 8048257:       83 e4 f0                and    $0xfffffff0,%esp
 804825a:       83 ec 10                sub    $0x10,%esp
 804825d:       c7 44 24 08 0e 00 00    movl   $0xe,0x8(%esp)
 8048264:       00
 8048265:       c7 44 24 04 48 a8 0a    movl   $0x80aa848,0x4(%esp)
 804826c:       08
 804826d:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048274:       e8 57 76 00 00          call   804f8d0 <__libc_write>
 8048279:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048280:       e8 cb 08 00 00          call   8048b50 <exit>
```

从上述指令序列可看出，在使用 call 指令调用 write()函数之前，main 函数在自身栈帧中传递了三个参数，首先在地址 R[esp]+8 处存放了 14（0xe），然后在 R[esp]+4 处存放了 0x8048510 (动态链接) 或 0x80aa848 (静态链接)，这个是指向字符串"Hello, world.\n"的指针，最后在 R[esp]处存放了 0x1。根据上述指令可以画出如右图所示的实参在 main 栈帧中的存放情况。

(2) 从执行 write()函数开始到调出 write 系统调用服务例程 sys_write()执行的过程中，首先通过 call 指令进入 write 封装函数执行。上述左图中的指令“call 8048350<write@plt>”（动态链接情况下）或右图中的指令“call 804f8d0<__libc_write>”(静态链接情况下）执行后，便进入封装函数 write 执行。

在进入 write()封装后，执行如下指令（静态链接情况下），可以看出，在执行陷阱指令（软中断指令)“int $0x80”之前，有 4 条 MOV 指令，其中有三条 MOV 指令用来将 main 栈帧中的三个参数：字符串长度 (0xe)、指向字符串"Hello, world.\n"的指针、文件描述符（0x1），分别送 EDX、ECX、EBX。还有一条 MOV 指令用来将调用号（4）送 EAX。

这样，当执行到完“int $0x80”指令后，就从用户态陷入内核态，调出系统调用处理程序 system_call()执行。在 system_call()中，根据系统调用号为 4，再跳转到相应的系统调用服务例程 sys_write()执行，以完成将一个字符串写入文件的功能，其中，字符串的首地址由 ECX 指定，字符串的长度由 EDX 指出，写入文件的文件描述符由 EBX 指出。system_call()执行结束时，从内核返回的参数存放在 EAX 中。

```
0804f8d0 <__libc_write>:
 804f8d0:       65 83 3d 0c 00 00 00    cmpl   $0x0,%gs:0xc
 804f8d7:       00
 804f8d8:       75 21                   jne    804f8fb <__write_nocancel+0x21>

0804f8da <__write_nocancel>:
 804f8da:       53                      push   %ebx
 804f8db:       8b 54 24 10             mov    0x10(%esp),%edx
 804f8df:       8b 4c 24 0c             mov    0xc(%esp),%ecx
 804f8e3:       8b 5c 24 08             mov    0x8(%esp),%ebx
 804f8e7:       b8 04 00 00 00          mov    $0x4,%eax
 804f8ec:       cd 80                   int    $0x80
 804f8ee:       5b                      pop    %ebx
 804f8ef:       3d 01 f0 ff ff          cmp    $0xfffff001,%eax
 804f8f4:       0f 83 f6 1f 00 00       jae    80518f0 <__syscall_error>
 804f8fa:       c3                      ret
 804f8fb:       e8 f0 0d 00 00          call   80506f0 <__libc_enable_asynccancel>
 804f900:       50                      push   %eax
 804f901:       53                      push   %ebx
 804f902:       8b 54 24 14             mov    0x14(%esp),%edx
 804f906:       8b 4c 24 10             mov    0x10(%esp),%ecx
 804f90a:       8b 5c 24 0c             mov    0xc(%esp),%ebx
 804f90e:       b8 04 00 00 00          mov    $0x4,%eax
 804f913:       cd 80                   int    $0x80
 804f915:       5b                      pop    %ebx
```

(3) 显然，对于第 3 题给出的实现方式，其程序设计的便捷性和灵活性都不如本题给出的实现方式，第 3 题采用汇编程序设计方式，只要参数不同，就需要重新编写不同的指令；而本题采用高级语言程序设计方式，只要在 write 函数中改变不同的实参，就可以完成不同的功能。

不过，第 3 题实现方式下的程序执行性能比本题实现方式好，因为用汇编实现时，省去了高级语言程序中大量的函数调用，因而它的执行时间更短。

9. 假定某计算机的 CPU 主频为 500MHz，所连接的某个外设的最大数据传输率为 20KBps，该外设接口中有一个 16 位的数据缓存器，相应的中断服务程序的执行时间为 500 个时钟周期，则是否可以用中断方式进行该外设的输入输出？假定该外设的最大数据传输率改为 2MBps，则是否可以用中断方式进行该外设的输入输出？

参考答案：

（1）因为该外设接口中有一个 16 位数据缓存器，所以，若用中断方式进行输入/输出的话，可以每 16 位数据进行一次中断请求，因此，中断请求的时间间隔为 $10^6 \times 2B/20kB=100\mu s$。

对应的中断服务程序的执行时间为$(10^6/500M)\times 500=1\mu s$，因为中断响应过程就是执行一条隐指令的过程，所用时间相对于中断处理时间（即执行中断服务程序的时间）而言，几乎可以忽略不计，因而整个中断响应并处理的时间大约 1μs 多一点，远远小于中断请求的间隔时间。因此，可以用中断方式进行该外设的输入输出。若用中断方式进行该设备的输入/输出，则该设备持续工作期间，CPU 用于该设备进行输入/输出的时间占整个 CPU 时间的百分比大约为 1/100=1%（也可以通过考察 1 秒钟内 500M 个时钟周期中有多少时钟周期用于中断来计算百分比，其计算公式为$(10^6/100\times 500)/500M=1\%$）。

（2）若外设的最大传输率为 2MBps，则中断请求的时间间隔为 $10^6 \times 2B/2MB=1\mu s$。而整个中断响应并处理的时间大约 1μs 多一点，中断请求的间隔时间小于中断响应和处理时间，也即中断处理还未结束就会有该外设新的中断到来，所以不可以用中断方式进行该外设的输入输出。

10. 若某计算机有 5 级中断，中断响应优先级为 1>2>3>4>5，而中断处理优先级为 1>4>5>2>3。要求：

(1) 设计各级中断处理程序的中断屏蔽位(假设 1 为屏蔽，0 为开放)；

(2) 若在运行主程序时，同时出现第 2、4 级中断请求，而在处理第 2 级中断过程中，又同时出现 1、3、5 级中断请求，试画出此程序运行过程示意图。

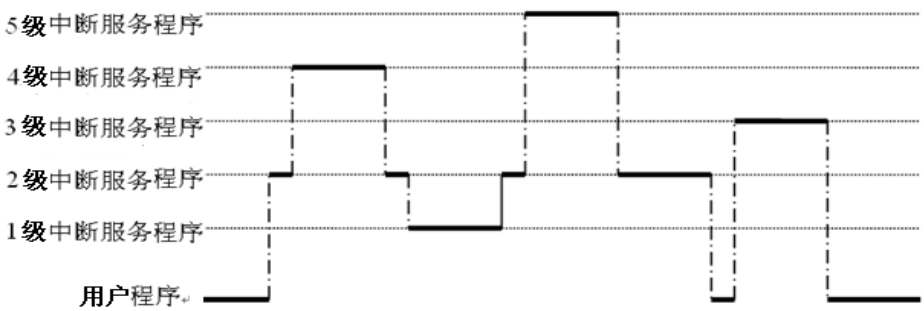
参考答案：

（1）由题意可知，1 级中断的处理优先级最高，说明 1 级中断对其他所有中断都屏蔽，其屏蔽字为全 1；3 级中断的处理优先级最低，所以除了 3 级中断本身之外，对其他中断全都开放，其屏蔽字为 00100。以此类推，得到所有各级中断的中断服务程序中设置的中断屏蔽字如下表所示。

中断处理	中断屏蔽字

程序	第 1 级	第 2 级	第 3 级	第 4 级	第 5 级
第 1 级	1	1	1	1	1
第 2 级	0	1	1	0	0
第 3 级	0	0	1	0	0
第 4 级	0	1	1	1	1
第 5 级	0	1	1	0	1

（2）在运行用户程序时，同时出现 2、4 级，因为用户程序对所有中断都开放，所以，在中断响应优先级排队电路中，有 2、4 两级中断进行排队判优，根据中断响应优先级 2>4，因此先响应 2 级中断。在 CPU 执行 2 级中断服务程序过程中，首先保护现场、保护旧屏蔽字、设置新的屏蔽字 01100，然后，在具体中断处理前先开中断。一旦开中断，则马上响应 4 级中断，因为 2 级中断屏蔽字中对 4 级中断的屏蔽位是 0，也即对 4 级中断是开放的。在执行 4 级中断结束后，回到 2 级中断服务程序执行；在具体处理 2 级中断过程中，同时发生了 1、3、5 级中断，因为 2 级中断对 1、5 级中断开放，对 3 级中断屏蔽，所以只有 1 和 5 两级中断进行排队判优，根据中断响应优先级 1>5，所以先响应 1 级中断。因为 1 级中断处理优先级最高，所以在其处理过程中不会响应任何新的中断请求，直到 1 级中断处理结束，然后返回 2 级中断；因为 2 级中断对 5 级中断开放，所以在 2 级中服务程序中执行一条指令后，又转去执行 5 级中断服务程序，执行完后回到 2 级中断，在 2 级中断服务程序执行过程中，虽然 3 级中断有请求，但是，因为 2 级中断对 3 级中断不开放，所以，3 级中断一直得不到响应。直到 2 级中断处理完回到用户程序，才能响应并处理 3 级中断。CPU 运行过程如下图所示。



11. 假定某计算机字长 16 位，没有 cache，运算器一次定点加法时间等于 100ns，配置的磁盘旋转速度为每分钟 3000 转，每个磁道上记录两个数据块，每一块有 8000 个字节，两个数据块之间间隙的越过时间为 2ms，主存周期为 500ns，存储器总线宽度为 16 位，总线带宽为 4MBps，假定磁盘采用 DMA 方式进行 I/O，CPU 时钟周期等于主存周期。回答下列问题。

(1) 磁盘读写数据时的最大数据传输率是多少?

(2) 当磁盘按最大数据传输率与主机交换数据时，主存周期空闲百分比是多少?

(3) 直接寻址的“存储器-存储器”SS 型加法指令在无磁盘 I/O 操作打扰时的执行时间为多少? 当磁盘 I/O 操作与一连串这种 SS 型加法指令执行同时进行时，则这种 SS 型加法指令的最快和最慢执行时间各是多少?

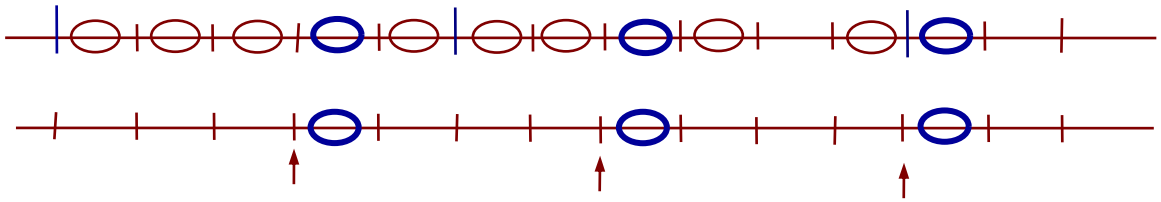
参考答案：

(1) 磁盘旋转一周所需时间为 $60\times10^3/3000=20$ms，单个数据块的传输时间为 (20ms/2)-2ms=8ms，所以最大数据传输率为 8000B/8ms=1MBps。平均数据传输率为 2×8000B/20ms=0.8MBps。

(2) 磁盘最大数据传输率为 1MBps，存储器总线宽度为 16bit=2B，故每隔 $10^9\times2B/1MB=2000$ns 产生一个 DMA 请求，即每 2000ns/500ns=4 个主存周期中有一个被 DMA 挪用，此时，CPU 没有访问主存，因此，4 个主存周期中有 3 个空闲，故主存频带空闲百分比是 75%，如下图所示。图中箭头处开始的一个主存周期被 DMA 挪用。

(3) 无 I/O 打扰时，执行一条直接寻址的 SS 型加法指令的过程如下图所示，包括取指令、取源操作数 1、取目操作数（源操作数 2）、执行、写结果，因此执行时间为 5×500ns=2.5 μs。此时，每个指令周期所包含的 5 个时钟周期中，只有执行阶段不访问主存，所以主存频带空闲百分比是 20%。

当磁盘 I/O 操作与一连串这种 SS 型加法指令同时进行时，可能因为 CPU 和 DMA 同时访存而使指令的执行时间被延长。每次 DMA 请求要求挪用一个主存周期来访问主存，同时，CPU 执行指令时也要求访问主存，当两者发生冲突时，DMA 优先级高，CPU 的访存请求被延迟。因为每隔 2000ns 产生一个 DMA 请求，因此每 4 个主存周期必定有一个被 DMA 所挪用。此时，主存周期的占用情况如下图所示。



由上图可知，最好的情况是在 SS 型加法指令执行过程中没有访存冲突（如上图中最开始的一个指令周期），此时最快，指令执行时间为 2.5μs；最坏的情况是有一次访存冲突（如上图中第二个指令周期），此时最慢，指令执行时间为 2.5μs +500ns=3μs。

13. 假设一个主频为 1GHz 的处理器需要从某个成块传送的 I/O 设备读取 1000 字节的数据到主存缓冲区中，该 I/O 设备一旦启动即按 50kBps 的数据传输率向主机传送 1000 字节数据，每个字节的读取、处理并存入内存缓冲区需要 1000 个时钟周期，则以下几种方式下，在 1000 字节的读取过程中，CPU 花在该设备的 I/O 操作上的时间分别为多少？占整个处理器时间的百分比分别是多少？

(1) 采用定时查询方式，每次处理一个字节，一次状态查询至少需要 60 个时钟周期；

(2) 采用独占查询方式，每次处理一个字节，一次状态查询至少需要 60 个时钟周期；

(3) 采用中断 I/O 方式，外设每准备好一个字节发送一次中断请求。每次中断响应需要 2 个时钟周期，中断服务程序的执行需要 1200 个时钟周期；

(4) 采用周期挪用 DMA 方式，每挪用一次主存周期处理一个字节，一次 DMA 传送完成 1000 字节数据的 I/O，DMA 初始化和后处理的时间为 2000 个时钟周期，CPU 和 DMA 没有访存冲突。

(5) 如果设备的速度提高到 5MBps，则上述 4 种方式中，哪些是不可行的？为什么？对于可行的方式，计算出 CPU 花在该设备 I/O 操作上的时间占整个处理器时间的百分比？

参考答案

主频为 1GHz，所以，时钟周期为 1/1GHz=1ns。因为每个字节的读取、处理并存入内存缓冲区需要 1000 个时钟周期，所以，对于像程序查询和中断等用软件实现输入/输出的方式，CPU 为每个字节传送所用的时间至少为 1000×1=1000ns=1μs。在 50kBps 的数据传输率下，设备每隔 10^6×1B/50kB=20μs=20000ns 准备好一个字节，因而读取 1000 字节的时间为 1000×20μs=20ms。

(1) 定时查询方式下的 I/O 过程如图 9.9 所示。可以设置每隔 20000ns 查询一次，这样使得查询程序的开销达到最小，即第一次读取状态时就可能会发现就绪，然后用 1000 个时钟周期进行相应处理，因此，对于每个字节的传送，CPU 所用时钟周期数

为 60+1000=1060。因此，在 1000 个字节的读取过程中，CPU 用在该设备的 I/O 操作上的时间至少为 1000×1060×1ns=1.060ms，占整个 CPU 时间的百分比至少为 1.060/20=5.3%。

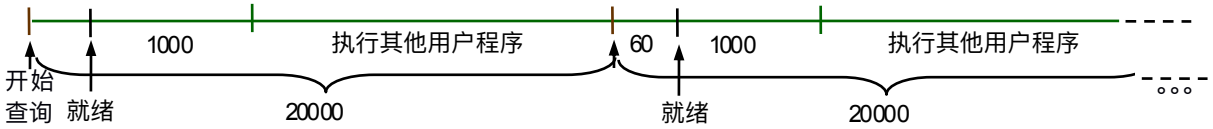
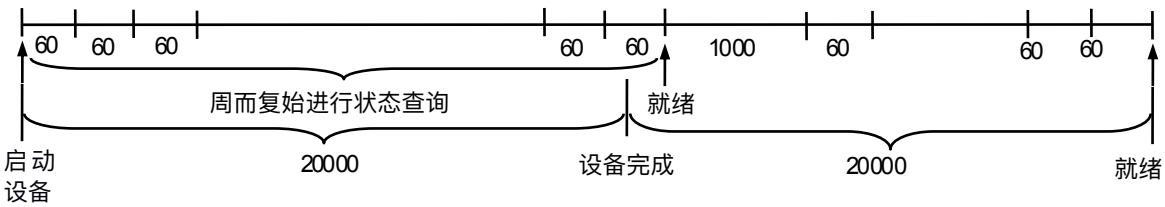
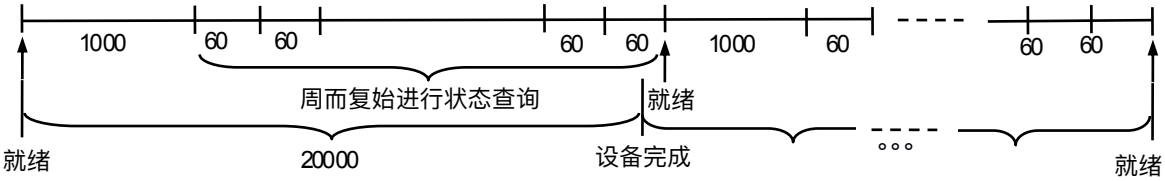


图 9.9　定时查询方式下的 I/O 过程

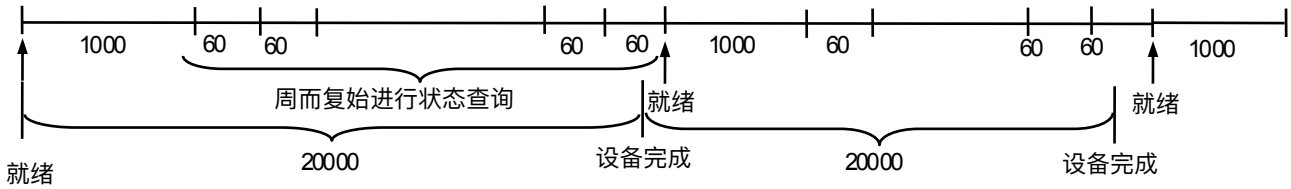
（2）独占查询方式下的 I/O 过程如图 9.10 所示。启动设备后，CPU 就开始查询，因为 333×60+20=20000，所以第一个字节传送在第 334 次读取状态查询时检测到就绪，随后用 1000 个时钟周期进行相应的处理，然后继续第二个字节的状态查询，因为 40+1000+316×60=20000，所以，第二个字节的传送在第 316 次读取状态查询时检测到就绪，第一个和第二个字节的传送过程如图 9.10(a)所示。每次检测到就绪后，就进行相应的处理，然后周而复始地进行查询，因为 20000-1000/60=316.7，所以，第 317 次状态查询时发现就绪。因为 1000+60×317-20000=20，所以，每 3 个字节可多 60 个时钟周期，正好进行一次状态查询，因此，在剩下的 998 个字节的读取过程中，前 996 个字节的传送正好用了 996×20000 个时钟周期，如图 9.10（b）所示。最后两个字节的传送过程如图 9.10（c）所示，因为 2×(1000+60×317-20000) = 40，此外，最后一个字节的处理还有 1000 个时钟周期，所以最后两个字节总的时间为 2×20000+40+1000=41040 个时钟周期。综上所述，CPU 用在该设备的 I/O 操作上的总时间为（1000×20000+1040×1ns=20.00104ms≈20ms。即在 1000 字节的整个传输过程中，CPU 一直为该设备服务，所用时间占整个 CPU 时间的 100%。



(a) 前两个字节的查询过程



(b) 中间996个字节的查询过程



(c) 最后两个字节的查询过程

图 9.10 独占查询方式下的 I/O 过程

(3) 中断方式下的 I/O 过程如图 9.11 所示。中断方式下，外设每准备好一个字节请求一次中断，每次中断 CPU 所用时钟周期数为 2+1200=1202，因此 CPU 用在该设备的 I/O 操作上的时间为 1000×1202×1ns=1.202ms，占整个 CPU 时间的百分比至少为 1.202/20=6.01%。

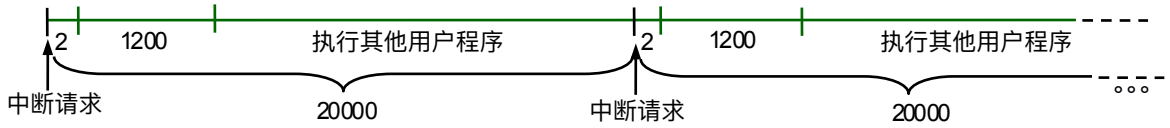


图 9.11 中断方式下的 I/O 过程

(4) DMA 方式下，由于 CPU 和 DMA 没有访存冲突，所以不需考虑由于 DMA 而影响到 CPU 执行其他程序。因此，传送 1000 个字节 CPU 所用的时钟周期数就是 2000，在 1000 个字节的读取过程中，CPU 用在该设备的 I/O 操作上的时间为 2000×1ns=2μs，占整个 CPU 时间的百分比为 2/(1000×20)=0.01%。

(5) 若设备数据传输率为 5MBps，则外设传输 1000 字节所用时间为 10^6×1000B/(5×10^6) B =200μs。

对于定时查询和独占查询方式，传送 1000 字节 CPU 所用时间至少为

1000×(60+1000)×1ns=1060μs；对于中断方式，传送1000字节CPU所用时间为1000×(2+1200)×1ns=1202μs。上述三种方式下，CPU所用的时间都比设备所用时间长得多，也即设备的传输比CPU的处理快得多，因而发生数据丢失。因此，这三种方式都不能用于该设备的I/O操作。对于DMA方式，传送1000字节CPU所用时间为2000×1ns=2μs，占整个CPU时间的百分比为2/200=1%。说明可以使用DMA方式，不过由于外设传输速度加快，使得CPU频繁进行DMA预处理和后处理，因而CPU的开销从0.01%上升到了1%。

第六章 层次存储结构

20. 假定一个虚拟存储系统的虚拟地址为40位，物理地址为36位，页大小为16KB。若页表中有有效位、访问权限位、修改位、使用位，共占4位，磁盘地址不记录在页表中，则该存储系统中每个进程的页表大小为多少？如果按计算出来的实际大小构建页表，则会出现什么问题?

参考答案：

因为每页大小有 16KB，所以虚拟页数为 2^{40}B/16KB=$2^{(40-14)}$=2^{26} 页。
物理页面和虚拟页面大小相等，所以物理页号的位数为 36–14=22 位。
页表项位数为：有效位+保护位+修改位+使用位+物理页号位数=4+22=26 位。
为简化页表访问，每项大小取 32 位。因此，每个进程的页表大小为：2^{26}×32b=256MB。
如果按实际计算出的页表大小构建页表，则页表过大而导致页表无法一次装入内存。

21. 假定一个计算机系统中有一个TLB和一个L1 Data Cache。该系统按字节编址，虚拟地址16位，物理地址12位，页大小为128B；TLB采用4路组相联方式，共有16个页表项；L1 Data Cache采用直接映射方式，块大小为4B，共16行。在系统运行到某一时刻时，TLB、页表和L1 Data Cache中的部分内容如下：

组号	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	13	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	63	0D	1	0A	34	1	72	–	0

(a) TLB（4路组相联)：4组、16个页表项

虚页号	页框号	有效位
00	08	1
01	03	1

行索引	标记	有效位	字节3	字节2	字节1	字节0
0	19	1	12	56	C9	AC
1	–	0	–	–	–	–

02	14	1
03	02	1
04	–	0
05	16	1
06	–	0
07	07	1
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	19	1
0D	–	0
0E	11	1
0F	0D	1

(b) 部分页表：(开始 16 项)

2	1B	1	03	45	12	CD
3	–	0	–	–	–	–
4	32	1	23	34	C2	2A
5	0D	1	46	67	23	3D
6	–	0	–	–	–	–
7	16	1	12	54	65	DC
8	24	1	23	62	12	3A
9	–	0	–	–	–	–
A	2D	1	43	62	23	C3
B	–	0	–	–	–	–
C	12	1	76	83	21	35
D	16	1	A3	F4	23	11
E	33	1	2D	4A	45	55
F	–	0	–	–	–	–

(c) L1 Data Cache：直接映射，共 16 行，块大小为 4B

请问（假定图中数据都为十六进制形式）：

(1) 虚拟地址中哪几位表示虚拟页号？哪几位表示页内偏移量？虚拟页号中哪几位表示 TLB 标记？哪几位表示 TLB 组索引？

(2) 物理地址中哪几位表示物理页号？哪几位表示页内偏移量？

(3) 主存物理地址如何划分成标记字段、行索引字段和块内地址字段？

(4) CPU 从虚拟地址 067AH 中取出的值为多少？说明 CPU 读取地址 067AH 中内容的过程。

(1) 16 位虚拟地址中低 7 位为页内偏移量，高 9 位为虚页号；虚页号中高 7 位为 TLB 标记，低 2 位为 TLB 组索引。

(2) 12 位物理地址中低 7 位为页内偏移量，高 5 位为物理页号。

(3) 12 位物理（主存）地址中，低 2 位为块内地址，中间 4 位为 cache 行索引，高 6 位为标记。

(4) 地址 067AH=0000 0110 0111 1010B，所以，虚页号为 0000011 00B，映射到 TLB 的第 00 组，将 0000011B=03H 与 TLB 第 0 组的四个标记比较，虽然和其中一个相等，但对应的有效位为 0，其余都不等，所以 TLB 缺失，需要访问主存中的慢表。直接查看 0000011 00B =00CH 处的页表项，有效位为 1，取出物理页号 19H=11001B，和页内偏移 111 1010B 拼接成物理地址：11001 111 1010B。根据中间 4 位 1110 直接找到 cache 第 14 行(即：第 E 行)，有效位为 1，且标记为 33H=110011B，正好等于物理地址高 6 位，故命中。根据物理地址最低两位 10，取出字节 2 中的内容 4AH=01001010B。

23. 假设在IA-32/Linux平台上运行一个C语言源程序P对应的用户进程，P中有一条循环语句S如下：

for (i=0; i<N; i++) sum+=a[i];

已知变量*sum*和数组*a*都是long型，链接后确定*a*的首地址为0x804d000。假设编译器将*a*的首地址分配在EDX中，数组的下标变量*i*分配在ECX中，*sum*分配在EAX中，赋值语句“sum+=a[i];”仅用一条指令I实现。已知IA-32/Linux平台采用图6.41所示的两级页表分页虚拟存储管理方式，页大小为4KB，系统启动后控制寄存器CR0中的控制位NW和CD均为0。假定系统中没有其他用户进程，回答下列问题或完成下列任务。

(1) 指令I对应的汇编形式（AT&T格式）是什么？指令I中存储器操作数的寻址方式是哪种?

(2) 假定指令I的地址为0x8049c08，执行指令I时，CS段寄存器对应的描述符cache中存放的是表6.2中所示的用户代码段信息且CPL=3，DS段寄存器对应的描述符cache中存放的是表6.2中所示的用户数据段信息，则当*i*=50时，取指令操作过程中MMU得到的指令的线性地址是多少？取数操作过程中MMU得到的操作数的线性地址是多少?

段	基地址	G	限界	S	TYPE	DPL	D	P
用户代码段	0x0000 0000	1	0xFFFFF	1	10	3	1	1
用户数据段	0x0000 0000	1	0xFFFFF	1	2	3	1	1
内核代码段	0x0000 0000	1	0xFFFFF	1	10	0	1	1
内核数据段	0x0000 0000	1	0xFFFFF	1	2	0	1	1

(3) 假定常数N在EBX中，手工写出循环语句S对应的指令序列，与GCC生成的目标代码进行比较。

(4) 在执行到程序P时，控制寄存器CR0中的控制位PE和PG各是什么?

(5) 指令I在第一次执行过程中，有没有可能发生缺页异常？为什么？如果发生缺页异常的话，则页故障线性地址是什么？该地址会保存在哪个控制寄存器中?

(6) 指令I所在页的虚页号是什么？指令I的线性地址中，页目录索引、页表索引和页内偏移量分别是什么？第一次执行指令I时，指令I所在页对应页表项中，字段P、R/W、U/S、A和D的内容各是什么?

(7) 指令I在第一次执行过程中，有没有可能发生TLB缺失？为什么？若指令TLB共有16个表项，采用4路组相联方式，则虚拟页号中哪几位为TLB标记？哪几位表示TLB组索引？若第一次执行到指令I时，指令TLB中的部分内容如下表（TLB表项中部分字段缺失，内容以十六进制表示），则指令I所存放的主存地址是什么?

组号	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位

0	03010	00A10	1	02101	0D001	1	00000	00000	0	00000	00000	0
1	02011	02D02	1	02012	028B0	1	04001	02012	1	00000	00000	0
2	02001	08902	1	02120	09200	0	02010	0340A	0	02301	0320A	1
3	01002	08770	1	00000	00000	0	0A001	02010	1	00000	00000	0

(8) 若指令cache的数据区容量为8KB，主存块大小为32B，采用2路组相联映射方式，则指令I在第一次执行时的取指令过程中，会不会发生cache缺失？指令I所在的主存块应映射到指令cache的哪一组中？

(9) 当N=2000时，数组a占用几个页面？每个页的虚页号是什么？数组元素a[1200]在哪个页中？

参考答案：

（1）因为IA-32中的long型和int型一样，都是32位带符号整数，因此，指令I对应的汇编形式（AT&T格式）是“addl (%edx, %ecx, 4), %eax”。其中存储器操作数的寻址方式是“基址+比例变址+偏移量”，这里的偏移量为0，比例因子是4，因为每个数组元素（long型）的大小为4个字节。

（2）取指令操作过程中，MMU根据CS段寄存器对应的描述符cache中的信息（因为执行的是用户程序，所以用Linux初始化时设定的用户代码段信息）进行逻辑地址到线性地址的转换。MMU根据CS对应段描述符中的DPL，确定DPL的特权级别不高于CPL才能继续进行地址转换，这里DPL=CPL=3，没有发生存储保护错。因此，MMU将CS对应段描述符中的基地址与指令逻辑地址（指令逻辑地址为指令在代码段的段内偏移量）相加，得到线性地址为0x0+0x8049c08=0x8049c08。

取数操作中，MMU先根据DS对应段描述符获得DPL，确定DPL的特权级别不高于CPL才能继续进行地址转换，否则发生存储保护错。显然，这里DPL=CPL=3，没有发生存储保护错，于是，MMU将DS对应段描述符中的基地址与操作数有效地址相加得到线性地址。因为操作数“(%edx, %ecx, 4)”的寻址方式为“基址加比例变址加偏移量”，故有效地址EA=R[edx]+R[ecx]×4+0=0x804d000+50×4=0x804d0c8。因此，操作数的线性地址为0x0+0x804d0c8=0x804d0c8。

（3）假定常数N在EBX中，则以下指令序列可以实现循环语句S的功能：

```
    movl    $0, %ecx
. LOOP
    cmpl    %ebx, %ecx
    jge     .EXIT
    addl (%edx, %ecx, 4), %eax
    incl    %ecx
    jmp     .LOOP
.EXIT
```

（4）在执行到程序P时，已经是保护模式并采用分页虚拟管理方式，因此，控制寄存器CR0中的控制位PE=1（表示保护模式），PG=1（表示启用分页）。PPT 162

（5）指令I在第一次执行过程中，取指令时不会发生缺页异常，因为，指令I不在一个页面起始处，在执行指令I前面的指令而发生缺页时，会将指令I一起调入内存。但是，取操作数a[0]时可能发生缺页，因为a[0]的地址为0x804d000，它正好位于一个页面的起始处（页大小为4KB，因此，页起始地址的后12位为0），它所在的页面很有可能以前未被访问过，若是这样的话，执行指令I的过程中，当访问操作数时会发生缺页异常。此时，页故障线性地址是0x804d000，该地址保存在控制寄存器CR2中。

（6）指令I的线性地址为0x8049c08，其中低12位（1100 0000 1000）为页内偏移量，高20位（0000 1000 0000 0100 1001）为虚页号。虚页号中高10位（0000 1000 00）为页目录索引，低10位（00 0100 1001）为页表索引。第一次执行指令I时，指令I所在页对应页表项中，P=1（所在页已在主存），R/W=0（所在页只能读和执行，不能写），U/S=1（所在页允许用户进程访问），A=1（所在页已被访问过），D=0（所在页为代码页，不会被修改）。PPT 164

（7）指令I在第一次执行过程中，取指令时不会发生TLB缺失，因为，指令I不在一个页面起始处，在执行指令I前面的指令而发生TLB缺失时，已经将所在页的页表项装入TLB。但是，取操作数a[0]时有可能发生TLB缺失，因为a[0]所在页可能是第一次被访问，因而对应页表项可能不在TLB中。

指令TLB共有16个表项，采用4路组相联方式，因此，20位虚拟页号中高18位为TLB标记，低两位为TLB组索引。指令I的线性地址为0x8049c08，虚页号为0000 1000 0000 0100 1001，其中，TLB组索引为01，因此，MMU将TLB标记0000 1000 0000 0100 10（02012H）与第1组所有标记相比，能找到一个相等且有效位为1的页表项在TLB中，因而TLB命中，取出对应页表项中的页框号028B0H，得到主存地址为028B0C08H=0x28b0c08。

（8）指令cache的数据区容量为8KB，主存块大小为32B，因此，指令cache共有8KB/32B=256行。因为采用2路组相联映射方式，因此，cache组数为256/2=128，主存地址划分为：高20位为标记、中间7位为组索引、最低5位为块内地址。因为页大小为4KB，所以线性地址中低12位的页内地址与主存地址低12位相同，在MMU进行逻辑地址到物理地址转换的同时，CPU可以利用这低12位进行cache访问。

指令I的线性地址为0x8049c08，其中低12位（1100 0000 1000）为块内偏移量，因此，组索引为1100000，块内地址为01000。显然，指令I不在一个主存块的起始位置，因而，即使在第一次执行指令I时，在取指令阶段也不会发生cache缺失（在取它前面的指令时已经顺带把它装入了指令cache）。

指令I所在的主存块应映射到指令cache的第1100000组中，即cache组号为96。

（9）当N=2000时，数组a占用空间大小为4×2000=8000字节，因为链接后*a*的首地址为0x804d000，而且对应段基地址为0，所以数组*a*的起始线性地址为0x804d000，显然这是一个页面的起始地址，因而数组*a*所占的页面个数为8000B/4KB=1.953125，即约占两个页面。虚页号分别是0000 1000 0000 0100 1101和0000 1000 0000 0100 1110。因为4×1200=4800>4096，所以数组元素a[1200]在后面一个页面中。