1. 请以某一种指令系统为例，说明其定义了哪些运行级别，以及这些运行级别之间的区别与联系。

答：以RISC-V和x86为例，它们定义了以下几个运行级别：

* 用户态（User Mode）：这是权限最低的级别，也被称为Ring 3（在x86架构中）或U-mode（在RISC-V中）。在用户态下，程序只能执行非特权指令，不能直接访问硬件或内核区域的内存。大部分用户级应用程序在此模式下运行。
* 内核态（Kernel Mode）：这是权限最高的级别，也被称为Ring 0（在x86架构中）或M-mode（在RISC-V中）。在内核态下，代码可以执行所有CPU指令并直接访问所有硬件资源。操作系统内核和设备驱动程序在此模式下运行。

这两种运行级别的主要区别在于它们的权限级别和能够执行的操作。用户态的程序只能访问有限的内存空间，而内核态的程序则可以访问所有的内存空间。当用户态程序需要执行特权操作（如访问硬件）时，它必须通过系统调用来请求内核在内核态下为其执行这些操作。

RISC-V架构定义了一个额外的运行级别，S-mode（Supervisor Mode），通常用于运行操作系统内核。

1. 请用 C 语言伪代码形式描述一个采用段页式存储管理机制的计算机系统进行虚实地址转换 的过程。（说明：不用描述微结构相关的内容，如 TLB；段描述符或页表中 的各种属性域均 视作有效。）

答：以下假设虚拟地址由段号、页号和页内偏移量组成，物理地址由帧号和帧内偏移量组成

struct VirtualAddress {

int segmentNumber;

int pageNumber;

int offset;

};

struct PhysicalAddress {

int frameNumber;

int offset;

};

// 虚实地址转换函数

PhysicalAddress translate(VirtualAddress va) {

// 从段表中获取段的基地址和页表

SegmentTableEntry ste = segmentTable[va.segmentNumber];

// 从页表中获取帧号

PageTableEntry pte = ste.pageTable[va.pageNumber];

// 构造物理地址

PhysicalAddress pa;

pa.frameNumber = pte.frameNumber;

pa.offset = va.offset;

return pa;

}

1. 请简述桌面电脑 PPT 翻页过程中用户态和核心态的转换过程

答：在桌面电脑上进行PPT翻页时，用户态和核心态的转换过程大致如下：

1. 用户在PPT应用程序（用户态）中按下键盘的翻页键，此时产生一个键盘中断请求。

2. 中断请求被操作系统接收，此时CPU从用户态切换到核心态。

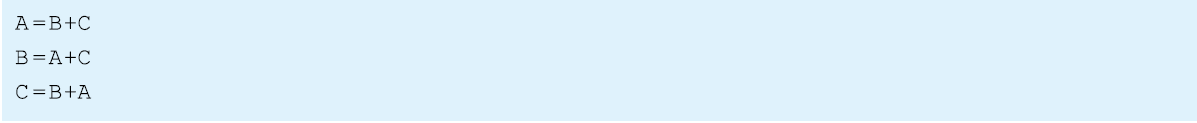
3. 操作系统在核心态下处理这个中断请求，即读取键盘缓冲区的数据，然后将这个翻页请求传递给PPT应用程序。

4. PPT应用程序接收到翻页请求后，调用操作系统提供的API（如图形设备接口），请求操作系统在核心态下进行屏幕刷新操作。

5. 操作系统完成屏幕刷新操作后，CPU从核心态切换回用户态，继续执行PPT应用程序。

这个过程中，用户态和核心态的转换主要发生在处理键盘中断和屏幕刷新操作时。这种转换机制确保了操作系统的稳定性和安全性，因为只有在核心态下，CPU才能执行访问硬件和修改系统资源的特权指令。

4. 给定下列程序片段：



( 1) 写出上述程序片段在四种指令系统类型 （堆钱型、累加器型、寄存器 存储器型、寄存 器 寄存器型） 中的指令序列。

( 2） 假设四种指令系统类型都属于 CISC 型，令指令码宽度为 z 位，寄存器操作数宽度为 y 位，内存地址操作数宽度为 z 位，数据宽度为 w 位。分析指令的总位数和所有内存访问的总位数。

(3） 微处理器由32 位时代进入了64 位时代，上述四种指令系统类型哪种更好？

(1)、堆栈型：

PUSH B

PUSH C

ADD

POP A

PUSH A

PUSH C

ADD

POP B

PUSH B

PUSH A

ADD

POP C

累加器型：

LOAD B

ADD C

STORE A

LOAD A

ADD C

STORE B

LOAD B

ADD A

STORE C

寄存器-存储器型：

LOAD R1, B

ADD R1, C

STORE A, R1

LOAD R1, A

ADD R1, C

STORE B, R1

LOAD R1, B

ADD R1, A

STORE C, R1

寄存器-寄存器型：

MOVE R1, B

MOVE R2, C

ADD R1, R1, R2

MOVE A, R1

MOVE R2, A

ADD R1, R1, R2

MOVE B, R1

MOVE R2, B

ADD R1, R1, R2

MOVE C, R1

（2）、对于CISC型指令系统，指令的总位数和所有内存访问的总位数如下：

堆栈型：PUSH/POP: x+z

ADD: x

总长度12x+9z

内存位数9w

访存总位数：12x+9z+9w

累加器型：

指令长度均为x+z

总长度7x+7z

内存位数7w

访存总位数 7x+7z+7w

寄存器-存储器型：

指令长度均为x+y+z

总长度7x+7y+7z

内存位数7w

访存总位数：7x+7y+7w+7z

寄存器-寄存器型：

LOAD/STORE: x+y+z

ADD: x+3y

总长度8x+14y+5z

内存位数5w

访存总位数：8x+14y+5z+5w

（3）、在寄存器-寄存器型指令系统会更好，因为在64位系统中，寄存器的宽度增加，可以存储更多的数据。寄存器-寄存器型指令系统可以充分利用这一优势，它主要在寄存器之间进行操作，而不是频繁地访问内存。这可以提高程序的运行效率。

5. 写出0xDEADBEEF 在大尾端和小尾端下在内存中的排列（由地址0开始）

0xDE AD BE EF

addr 0 1 2 3

Big-endian DE AD BE EF

Little-endian EF BE AD DE

6、

#include <stdio.h>

int main() {

printf("Size of char: %lu byte\n", sizeof(char));

printf("Size of int: %lu bytes\n", sizeof(int));

printf("Size of short: %lu bytes\n", sizeof(short));

printf("Size of long: %lu bytes\n", sizeof(long));

printf("Size of float: %lu bytes\n", sizeof(float));

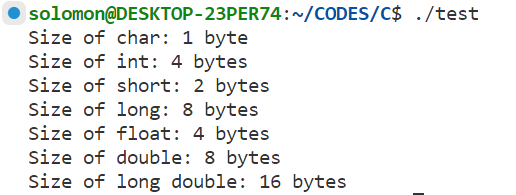
printf("Size of double: %lu bytes\n", sizeof(double));

printf("Size of long double: %lu bytes\n", sizeof(long double));

return 0;

}

(Core AMD Rzen2700x, WSL, Ubuntu 20.04 64bit)下运行结果：



7. 根据 LoongArch 指令集的编码格式计算 2RI16、1RI21 和 126 三种编码格式的直接转移指令各自的跳转范围。

LoongArch 指令集中的所有指令都采用 32 位固定长度。对于直接转移指令，其跳转范围取决于立即数的位数和解释方式。

2RI16：这种格式的指令有 16 位的立即数。由于这是一个转移指令，立即数通常被解释为相对于当前指令的偏移量，且以字（4 字节）为单位。

因此，最大正向跳转范围是 ×4−4=131068字节，最大负向跳转范围是 −×4=−131072字节。

1RI21：这种格式的指令有 21 位的立即数。同样，立即数被解释为相对于当前指令的偏移量，以字为单位。因此，最大正向跳转范围是 ×4−4=4194300字节，最大负向跳转范是 −×4=−4194304字节。

I26：这种格式的指令有 26 位的立即数。立即数被解释为相对于当前指令的偏移量，以字为单位。因此，最大正向跳转范围是 ×4−4=134217724字节，最大负向跳转范围是 −×4=−134217728字节。

指令必须对齐到 4 字节边界，所以偏移量是以 4 字节（一个字）为单位的。这意味着立即数的实际位数比字段中的位数少 2 位。最后，由于跳转偏移是相对于下一条指令的地址计算的，所以在计算最大正向跳转范围时需要减去 4 字节。

8.仅使用对齐访存指令写出如图2.9 所示的不对齐加载 （小尾端）

LW R1, 0

SRL R1, R1, 8

LBU R2, 4

SLL R2,R2, 24

OR R1, R1, R2