# OSlab1

## 组员：

2111451 朱景博
2110729 张弛
2111566 米奕霖

### *练习1: 理解内核启动中的程序入口操作*

> 阅读 kern/init/entry.S内容代码，结合操作系统内核启动流程，说明指令 la sp, bootstacktop 完成了什么操作，目的是什么？ tail kern_init 完成了什么操作，目的是什么？

1. `la sp, bootstacktop`：
   - 这句指令使用伪指令 `la`（Load Address）来加载一个地址到寄存器 `sp` 中。
   - `sp` 是堆栈指针寄存器，用于指示当前堆栈的顶部。
   - `bootstacktop` 是一个全局标签，它代表了内核启动堆栈的顶部地址。
   - 这句指令的目的是将堆栈指针 `sp` 设置为内核启动堆栈的顶部地址，从而初始化堆栈。
2. `tail kern_init`：
   - `tail` 是 MIPS 汇编指令中的一个伪指令，用于无条件跳转到一个目标地址，并且不保留当前函数调用的返回地址。这是一种特殊的跳转，通常用于启动内核的入口点。
   - `kern_init` 是一个全局标签，它代表了内核初始化函数的地址。
   - 执行 `tail kern_init` 会将控制权转移到内核初始化函数 `kern_init`，并且同时保留了调用栈的信息。执行完毕后，程序将不再回到之前的调用点，而是继续执行 `kern_init` 函数。这是因为 `tail` 指令不会保存返回地址，而是直接跳转，这是内核启动过程中的一种常见技巧，以确保启动代码的最后一个步骤是进入内核初始化过程，而不再返回到启动代码。
   - 此指令的目的是跳转到内核初始化函数 `kern_init` 的入口点，以开始执行操作系统内核的初始化过程。

总之，这两句汇编指令的目的是在操作系统内核启动时 **设置堆栈指针并跳转到内核初始化函数**，以便开始操作系统的初始化和执行。

### *练习2: 完善中断处理(需要编程)*

> 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”，在打印完10行后调用sbi.h中的shut_down()函数关机。

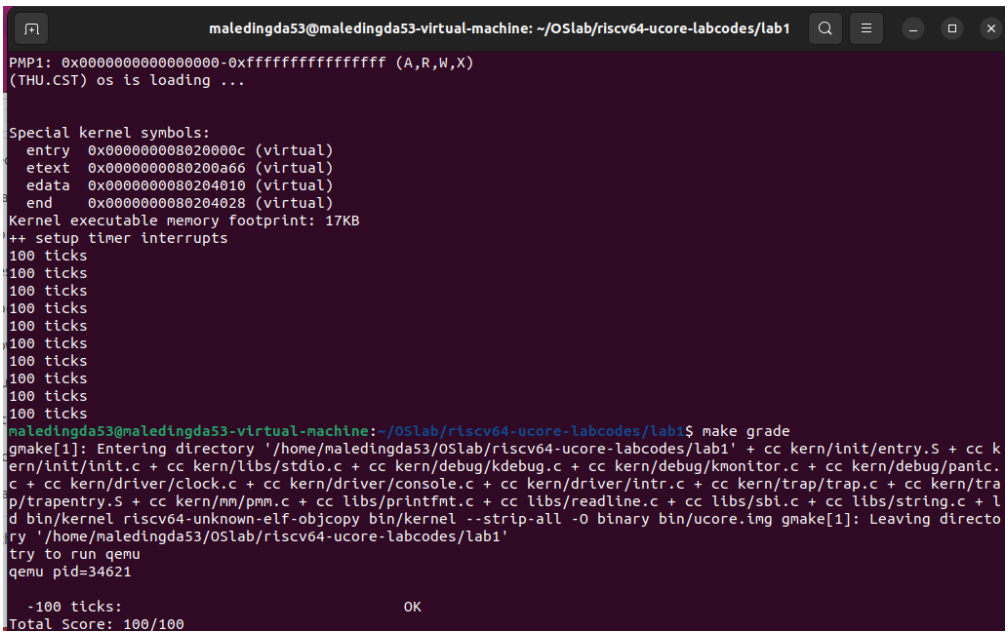
不难得知，`kern/trap/trap.c` 中处理中断的函数为 `interrupt_handler`，要实现上述功能，只需在 `interrupt_handler` 定位到由 `scause` 确定的 `IRQ_S_TIMER` 情况下，即S模式下的计时器中断。

在这里我们对时钟中断进行了一个简单的判断+输出处理，即每次触发时钟中断的时候，给 ***计数器(ticks)*** 加一，并且设定好下一次时钟中断。当计数器加到100的时候，我们会输出一个 **100ticks** 表示我们触发了100次时钟中断。每次打印后我们也会对 ***打印次数(num)*** 进行判断，当打印次数为10时，调用 `<sbi.h>` 中的关机函数关机。

```
void interrupt_handler(struct trapframe *tf) {
```

```
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        /* blabla 其他case*/
        case IRQ_S_TIMER:
            clock_set_next_event();//设置下一次时钟中断
            if (++ticks % TICK_NUM == 0) {
               print_ticks();
               if (++num == 10)
                {
                  sbi_shutdown();//调用关机函数
                }
            }
            break;
        /* blabla 其他case*/
}
```

完善`trap.c`中的中断处理函数后，在lab1的根目录下执行`make qemu`指令运行整个系统，即可以观察到大约每1秒会输出一次”100 ticks”，输出10行后关机停止，接着执行`make grade`指令则可以更进一步显示出打分成绩。




## *扩展练习 Challenge1：描述与理解中断流程*

> 回答：描述ucore中处理中断异常的流程（从异常的产生开始）

ucore中处理中断异常的流程可以分为以下几个步骤：

1. **中断or异常产生**
2. **根据*stvec*跳转**：
   中断产生后，CPU首先会跳转到 *stvec(中断向量表基址)* ，然后根据中断向量表把不同种类的中断映射到对应的中断处理程序，即根据中断种类进行相应处理程序的跳转。（如果只有一个中断处理程序，则stvec直接指向那个中断处理程序的地址，即指导手册中引以为例的 *Direct模式* ）。在本次实验中，异常产生后，会跳转到寄存器stvec保存的地址执行指令，由于内核初始化时将该寄存器设置为**alltraps，所以会跳转到trapentry.S中的**alltraps标签处执行。

3. **保存上下文**：
   在进入中断处理程序之前，处理器通常会保存当前的 *context* ，包括寄存器状态和程序计数器（PC）。这些信息保存在内核栈上，以便在处理完中断后能够恢复执行。（即执行宏定义的SAVE ALL)
4. **中断处理**：
   处理器跳转到中断处理函数trap后，实际上把中断处理、异常处理的工作分发给了 `interrupt_handler()` ， `exception_handler()` ，这些函数再根据中断或异常的不同类型来处理(中断和异常的区分、不同中断或不同异常种类的区分都依据 *scause* 来进行，源码中对于不同类型的中断，仅是输出信息后返回，实际中“因地制宜”采用不同的处理方式)。
5. **恢复上下文**：
   在处理完中断后，处理器会从内核栈中恢复之前保存的上下文信息，包括寄存器状态和PC。
6. **返回用户态**：
   一旦上下文恢复完成，处理器会将控制权返回到之前被中断的用户进程或代码处(执行 `sret` 指令,从 S 态中断返回到 U 态，即pc←sepc，返回到通过中断进入 S 态之前的地址)，使其继续执行。

> 其中mov a0，sp的目的是什么?

在处理中断异常的过程中，编写中断入口点的汇编程序时， `move a0, sp` 的目的是将当前的堆栈指针( `sp` ) 的值(即 **将上下文包装成结构体** )复制到寄存器 `a0` 中，并作为参数传递给接下来要调用的 `trap` 函数。具体阐述如下：

1. **传递参数给 `trap` 函数**： `a0` 寄存器用于传递参数给接下来要调用的 `trap` 函数，这是根据 RISC-V calling convention（RISC-V 的调用约定）来执行的。通常情况下，函数参数会通过寄存器传递给函数，而 `a0` 寄存器是其中的一个通用寄存器，用于传递第一个参数。
2. **调用 `trap` 函数**：接下来的指令 `jal trap` 是一个跳转并链接的指令，它将控制权转移到 `trap` 函数，同时保存了返回地址。 `trap` 函数是中断处理程序，它将处理中断并可能执行一些操作，然后最终需要返回到这个中断入口点。

总之， `move a0, sp` 的目的是将当前堆栈指针的值传递给中断处理程序，以便中断处理程序能够访问当前堆栈上的数据或执行与堆栈相关的操作。这是一种常见的做法，以确保在处理中断时可以访问堆栈上的关键信息。

> SAVE_ALL中寄存器保存在栈中的位置是什么确定的?

在预定义的 `SAVE_ALL` 汇编宏源码 `kern/trap/trapentry.S` 中，我们可以观察到，寄存器保存在栈中的位置是基于 RISC-V 的通用寄存器和特殊寄存器的编号来确定的，具体如下：

1. 前面存放了 32 个通用寄存器 `x0` 到 `x31` ，每个寄存器占据一个 `REGBYTES` 大小的空间，其中 `REGBYTES` 是一个常量，表示一个寄存器占据的字节数。通常情况下， `REGBYTES` 在 RISC-V 中是 8 字节，因此每个寄存器占据 8 字节的空间。
2. 接下来，存放了 4 个 CSR，即 `sscratch` 、 `sstatus` 、 `sepc` 、 `sbadaddr` 、 `scause` 。这些寄存器也占据 `REGBYTES` 大小的空间，依次存放在栈中。

这样，每个寄存器在栈中的存放位置都是按照其编号和 `REGBYTES` 来计算的。例如， `x0` 存放在栈的 0 * `REGBYTES` 处， `x1` 存放在栈的 1 * `REGBYTES` 处，以此类推，而 CSR 也是按照这个规律存放在栈中的。

需要注意的是，由于 ***小端存储*** 的缘故，栈的生长方向是向低地址方向延伸，因此栈顶指针 `sp` 在 `SAVE_ALL` 宏中减去 `36 * REGBYTES` 来分配栈空间，这样可以确保每个寄存器都有足够的空间来存储。

> 对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由。

并非对于任何中断都需要保存所有寄存器。在中断处理中，是否需要保存所有寄存器的内容取决于中断的类型、操作系统设计和具体的处理需求。以下是一些考虑因素：

1. ***Criticality of the Interrupt***: 不同类型的中断可能具有不同的重要性。一些中断可能只需要保存少数寄存器，而其他中断可能需要保存所有寄存器。例如，处理器异常（如页面错误）可能需要保存所有寄存器，以便在异常处理程序中恢复进程状态，而定时器中断可能只需要保存少数寄存器。
2. ***Performance Impact***: 保存和恢复所有寄存器可能会引入一定的性能开销，因为需要大量的数据传输操作。在性能敏感的系统中，可能会选择仅保存必要的寄存器，以降低中断处理的开销。
3. ***Context Switching***: 如果操作系统需要支持多任务或多线程，那么中断处理可能需要考虑上下文切换。在这种情况下，需要保存和恢复更多的寄存器，以确保不同任务之间的上下文切换是正确的。
4. ***Interrupt Handler Design***: 中断处理程序的设计也会影响需要保存的寄存器数量。某些处理程序可能需要更多的寄存器来执行复杂的操作，而其他处理程序可能只需少量寄存器。

总之，是否需要保存所有寄存器取决于具体的中断处理需求和设计，以及性能和上下文切换的考虑。在某些情况下，可能只需要保存少数寄存器，而在其他情况下，可能需要保存所有寄存器来确保正确的中断处理。

## *扩展练习 Challenge2：理解上下文切换机制*

> 回答：在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么?

在 `trapentry.S` 中，`csrw sscratch, sp` 和 `csrrw s0, sscratch, x0` 这两条指令用于保存和恢复 `sscratch` 寄存器的值。具体阐述如下：

1. `csrw sscratch, sp`：
   - 这条指令将当前栈指针 `sp` 的值写入 `sscratch` 寄存器中。
   - `sscratch` 寄存器通常被用于保存一个临时值，以便在中断处理过程中进行临时存储。在这里，它被用于保存当前栈指针的值。
   - 目的是将当前栈指针 `sp` 的值保存到 `sscratch` 寄存器，以便在中断处理期间能够随时恢复栈指针的值，以确保中断处理过程不会破坏当前任务的栈。
2. `csrrw s0, sscratch, x0`：
   - 这条指令从 `sscratch` 寄存器中读取其当前值，并将其存储在通用寄存器 `s0` 中，同时将 `sscratch` 寄存器的值设置为 `x0`（零值寄存器）。
   - 目的是将之前保存在 `sscratch` 寄存器中的栈指针的值恢复到通用寄存器 `s0` 中，以便在后续的代码中使用。
   - 设置 `sscratch` 寄存器为零是为了清除其中的值，以确保中断处理完成后，`sscratch` 寄存器不再包含之前的栈指针值，并且如果产生了递归异常，异常向量就会知道它来自于内核。

总之，这两条指令的目的是为了在中断处理过程中安全地保存和恢复栈指针的值，以确保中断处理不会破坏当前任务的栈，并在中断处理完成后正确地恢复上下文。

> save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢?

在 `SAVE_ALL` 汇编宏中，保存了一些 CSR，即`sstatus`、`sepc`、`sbadaddr` 和 `scause`，但在 `RESTORE_ALL` 汇编宏中并没有还原它们。这是因为这些特殊寄存器的值通常在中断处理过程中是不需要恢复的 (即保存它们的值只为了中断处理期间能够访问)，而且恢复它们可能引入不必要的复杂性。

具体阐述如下：

1. `sstatus` **寄存器**：`sstatus` 寄存器包含了当前特权级别（如内核模式或用户模式）和中断使能位等信息。在中断处理期间，通常会切换到内核模式，因此不需要保存和恢复 `sstatus` 寄存器的值，因为在返回到用户代码之前，会执行 `sret` 指令来恢复这些状态。

2. `sepc` **寄存器**：`sepc` 寄存器包含了引发异常或中断时的程序计数器值，它在中断处理过程中是不需要恢复的，因为中断返回时会自动从 `sepc` 寄存器中获取下一条指令的地址。
3. `sbadaddr` **寄存器**：`sbadaddr` 寄存器包含了引发异常或中断时的错误地址，通常不需要在中断处理过程中恢复它，因为在异常或中断处理后，操作系统可以根据需要进行错误处理。
4. `scause` **寄存器**：`scause` 寄存器包含了引发异常或中断的原因代码。在中断处理中，处理程序通常会根据 `scause` 的值来确定中断的类型，并采取适当的措施。不需要在 `RESTORE_ALL` 中还原 `scause` 寄存器，因为它仅用于中断处理的判断。

总之，这些特殊寄存器的值通常在中断处理过程中不需要还原，因为它们的值在中断返回时会自动恢复到先前的状态。保存它们的主要目的是在中断处理期间能够访问它们的值，以便进行错误处理和中断类型判断。不还原它们的目的是简化代码和提高执行效率。

## *扩展练习 Challenge3：完善异常中断*

> 编程完善在触发一条非法指令异常 mret后，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）”与“Exception type:Illegal instruction”，“Exception type: breakpoint”。

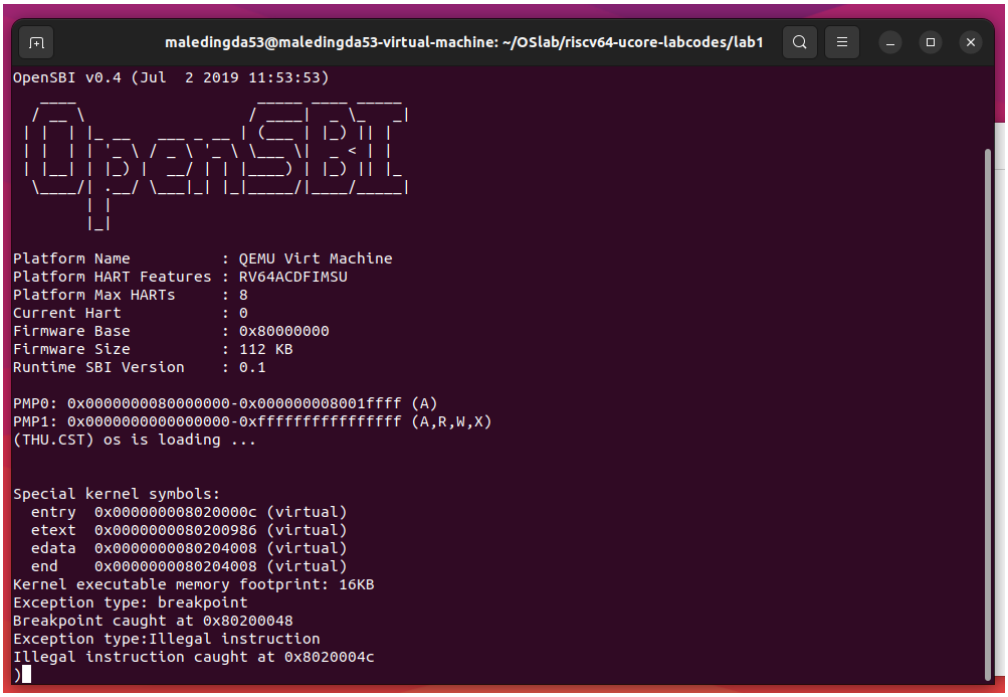
首先与练习2类似地定位到 `kern/trap/trap.c` 中的处理异常函数 `exception_handler`，锁定要修改的两种异常处理情况：**非法指令异常** 和 **断点异常**，即 `CAUSE_ILLEGAL_INSTRUCTION` 和 `CAUSE_BREAKPOINT` 对应的两种 case，由于此处的处理只是简单的输出异常信息和对异常指令的无脑跳过(更新 `tf->epc`)，故两种异常处理的完善逻辑一致，都只需要先输出异常类型，以及发生异常指令的地址( `pc<-epc` )，并且将 `tf->epc` 加 4 跳转到原本待执行的下一条指令即完成了异常的处理。

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        /* blabla 其他case*/
        case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB1 CHALLENGE3   YOUR CODE :  */
            /*(1)输出指令异常类型（ Illegal instruction）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
            */
            cprintf("Exception type:Illegal instruction\n");
            cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
            tf->epc += 4;
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB1 CHALLLENGE3   YOUR CODE :  */
            /*(1)输出指令异常类型（ breakpoint）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
            */
            cprintf("Exception type: breakpoint\n");
            cprintf("Breakpoint caught at 0x%08x\n", tf->epc);
            tf->epc += 4;
        /* blabla 其他case*/
    }
}
```

完善好两种异常的处理函数后，还需要考虑如何分别触发它们。首先定位到源码中的`kern\init\init.c`，容易观察到默认的是初始化 console 、输出 Message 后直接调用`clock_init()`初始化时钟中断，从而引发链式反应，不断的触发中断直到调用`sbi_shutdown()`关闭退出，此处为了触发两种异常，只需要将初始化触发中断的函数注释掉后，通过内联汇编的方法先后触发异常即可，这里分别采用的是`ebreak`和`.word 0xFFFFFFFF`指令来分别触发断点异常和非法指令异常。

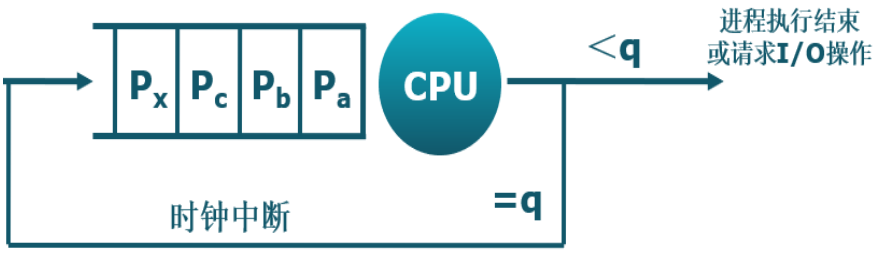
```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    cons_init();  // init the console

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);

    print_kerninfo();

    // grade_backtrace();

    idt_init();  // init interrupt descriptor table

    asm volatile("ebreak"); // trigger a breakpoint exception

    asm volatile(".word 0xFFFFFFFF");// trigger an Illegal instruction exception

    // rdtime in mbare mode crashes
    //clock_init();  // init clock interrupt

    //intr_enable();  // enable irq interrupt

    while (1)
        ;
}
```

最终在 **OpenSBI** 中的效果如下，成功触发了两种异常并作了相关信息的打印。




## *知识点复盘*

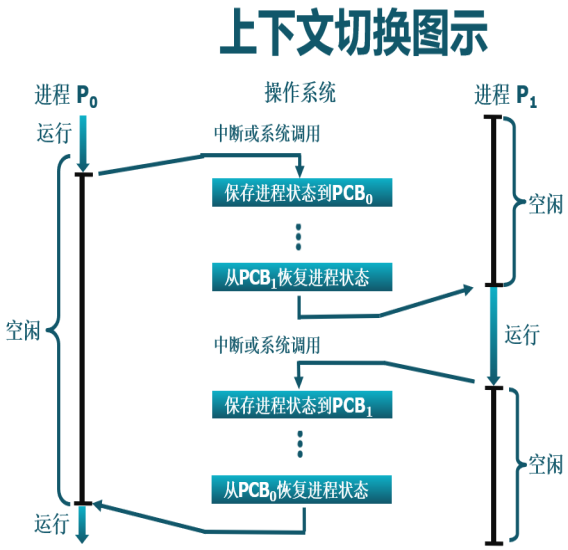
### 1. *OS* 何时会触发时钟中断呢?




- **定时器中断**：操作系统通常会设置一个硬件定时器，该定时器以固定的时间间隔触发中断。这个中断被称为时钟中断。时钟中断的频率通常是操作系统的时钟周期，通常以毫秒为单位。当定时器中断发生时，操作系统会执行与时钟中断相关的处理程序。
- **多任务调度**：操作系统使用时钟中断来实现多任务调度。时钟中断会定期触发，操作系统在每次时钟中断时可以检查当前运行的任务，并根据调度算法决定是否切换到另一个任务。这样，操作系统能够实现多任务并发执行。
- **计时器和延时**：操作系统可以使用时钟中断来测量时间间隔或实现延时操作。例如，如果一个应用程序需要等待一定的时间后才能执行下一步操作，操作系统可以使用定时器来实现这个延时。
- **性能监测和统计**：时钟中断还可以用于操作系统性能监测和统计。操作系统可以使用时钟中断来定期收集系统性能数据，如CPU利用率、内存使用情况等。
- **时间管理**：时钟中断还用于操作系统的时间管理。它帮助操作系统跟踪时间、记录时间戳以及执行时间相关的操作。

总之，时钟中断是操作系统中的一个关键机制，用于管理多任务、测量时间间隔、实现延时、进行性能监测和时间管理。操作系统通过配置硬件定时器来触发时钟中断，以便在不同的情况下执行相应的处理程序和任务调度。这有助于操作系统有效地控制计算机系统的资源和响应用户和应用程序的需求。

## 2. 操作系统中的上下文切换




1. **触发上下文切换的情况**：
   - **时钟中断**：时钟中断是最常见的触发上下文切换的情况。操作系统会定期生成时钟中断，以便轮流分配CPU时间给不同的任务。
   - **系统调用**：当任务执行系统调用（如I/O操作）时，操作系统可能会执行上下文切换，以便处理系统调用。
   - **任务阻塞**：如果一个任务需要等待某些事件（例如等待I/O完成），操作系统会将该任务置于阻塞状态并切换到另一个任务。
   - **任务主动让出CPU**：有些情况下，任务可以显式地请求操作系统将CPU分配给其他任务，这称为任务主动让出CPU。
2. **上下文切换的步骤**：
   - 保存当前任务的上下文信息，包括寄存器状态、程序计数器等。
   - 在任务切换过程中执行调度算法，选择下一个要执行的任务。
   - 加载所选任务的上下文信息，包括寄存器状态、程序计数器等。
   - 继续执行所选任务的代码。
3. **开销**：上下文切换是有开销的，因为它需要保存和加载大量的上下文信息。为了减少开销，操作系统会尽量减小上下文切换的频率。
4. **协程和用户级线程**：与传统的基于进程或线程的上下文切换不同，协程和用户级线程可以在用户空间中进行上下文切换，而无需涉及操作系统内核，从而减小了切换的开销。
5. **内核级上下文切换和用户级上下文切换**：内核级上下文切换是在操作系统内核的支持下进行的，而用户级上下文切换是在用户空间的应用程序内进行的。内核级上下文切换更强大，但通常开销更大。

上下文切换是操作系统实现多任务和并发的核心机制之一，它需要高效地管理任务的上下文信息，确保任务之间的无缝切换。操作系统的性能和响应能力受到上下文切换的影响，因此上下文切换的优化对于提高系统性能非常重要。

## 3.操作系统中常见的中断和异常类型

1. **外部硬件中断**：这包括来自外部设备（如磁盘驱动器、键盘、鼠标、网络适配器等）的中断。操作系统需要处理这些中断以响应设备的输入或输出请求。
2. **异常**：异常是由于程序执行期间出现的错误或非正常情况而触发的事件。常见的异常类型包括：

    - **除零异常**：当程序尝试除以零时触发。
    - **页故障**：当程序访问未映射的内存或者发生页面交换时触发。
    - **非法指令**：当程序尝试执行非法指令时触发。
    - **越界异常**：当程序访问数组越界或者栈溢出时触发。
    - **浮点异常**：当浮点运算出现问题时触发。
3. **系统调用**：当应用程序请求操作系统提供的服务（如文件操作、进程管理、网络通信等）时，操作系统会通过系统调用中断来响应这些请求。
4. **软件中断**：软件中断是由程序内部生成的中断。它可以用于实现进程间通信、调试和特定功能的触发。
5. **输入/输出完成中断**：操作系统需要监视设备的I/O操作并处理I/O完成中断，以通知应用程序其I/O请求的状态。
6. **时钟周期中断**：除了时钟中断之外，还可以存在其他周期性中断，用于执行特定的任务或服务。
7. **硬件故障中断**：操作系统需要处理硬件故障，如内存错误、CPU错误等，以确保系统的稳定性和可靠性。
8. **电源管理中断**：操作系统需要处理与电源管理相关的中断，以管理设备的电源状态，以及在低功耗模式和唤醒模式之间切换。
9. **通信中断**：对于多处理器系统或多核心系统，可能需要处理各个核心之间的通信中断，以便协调和同步它们的操作。

不同的操作系统和硬件平台可能支持不同类型的中断和异常，但上述列出的是常见的中断和异常类型，它们在操作系统的正常运行中扮演着重要的角色。操作系统需要有效地管理这些中断和异常，以确保系统的稳定性、可靠性和性能。