国科大操作系统研讨课任务书

Project 2

（第二阶段）

版本：4.0

一、实验说明

在上一个 Project 中，我们实现了进程的 PCB 设计和非抢占式的调度功能。在这个 Project 中，我们将实现抢占式调度和系统调用。这就涉及到操作系统设计中非常重要的例外处理流程设计。例外处理牵扯到操作系统中最重要的用户态与内核态的概念，在实现本 project 的功能时需要非常明确的了解系统的哪一部分功能是运行在用户态的，哪一部分功能是属于内核态的。而对于例外，想要设计好一个功能完备正确的例外处理程序，除了要设计好本身的功能之外，还要了解硬件是在例外发生时做了什么，充当了什么作用。

本次实验的内容如下：

（1）任务一：了解 MIPS 下例外处理的流程，实现操作系统中例外处理、时钟中断处理、抢占式调度、计数函数。

（2）任务二：了解 MIPS 下系统调用处理流程，实现操作系统中系统调用、sleep 系统调用方法。

值得强调的是，本次实验的难度较高。而难度较高的原因是大家对操作系统的一些概念以及设计要点还不清楚。因此，我们推荐大家在动手写代码之前做好整体设计，并在代码调试的过程中多思考，多查阅相关资料，理解硬件的工作原理，以及操作系统中的重要概念。

二、例外处理

在 MIPS 中，将中断和异常统称为**例外**。通俗的点说，它是程序在正常执行过程中的强制转移。产生例外的原因有很多，有一些例外是主动触发的，比如 syscall 调用，有一些例外是被动触发的，比如硬件例外。

在这次实验大家需要掌握和实现 MIPS 下例外处理流程，并且实现时钟中断以及系统调用的例外处理代码。经过本次实验，你的操作系统将具备例外处理能力，实现任务的抢占式调度，以及系统调用处理模块。

2.1 例外处理流程

2.1.1 例外的触发

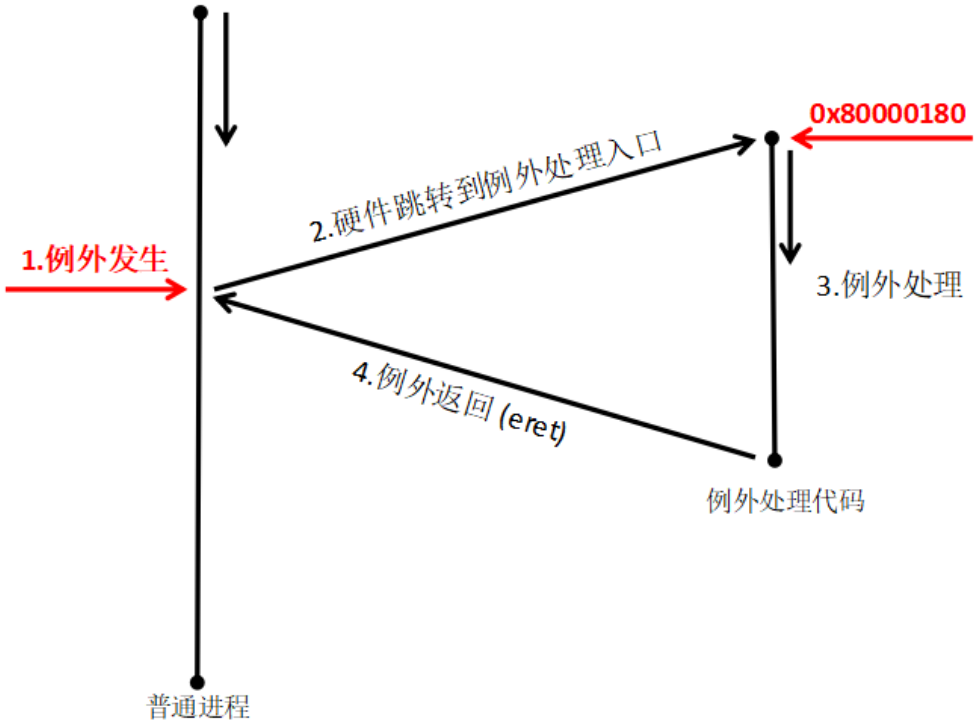
当发生异常时，处理器执行地址会将发生异常的地址放入 CP0_EPC 寄存器，然后自动跳转到一个固定的例外处理入口（这个过程是硬件自动完成的），例外处理入口如下表：

BEV 位	例外类型	例外向量地址
	Cold Reset/ NMI	0xFFFFFFFF BFC00000
BEV = 0	TLB Refill (EXL=0)	0xFFFFFFFF 80000000
	XTLB Refill (EXL=0)	0xFFFFFFFF 80000000
	Others	0xFFFFFFFF 80000180
BEV = 1	TLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	XTLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	Others	0xFFFFFFFF BFC00380

例外入口地址

启动时(Boot-Time)例外向量(状态寄存器中的 BEV 位=1)地址位于既不通过 Cache 进行存取，也无需地址映射的地址空间。在正常操作期间(BEV 位=0)，普通例外的向量地址位于需要通过 Cache 进行存取的地址空间。

对于我们本次的任务而言，产生一个中断处理器就会跳转到 0x80000180 这个位置，需要一提的是，在发生例外时，跳转到例外处理入口是由硬件做的，但是例外的处理需要软件完成。并且，在发生例外之后，操作系统应该实现从用户态切换进内核态，并在例外处理结束时候返回用户态。例外处理流程如下图所示：



跳转到这里后，需要执行这个地方的例外处理代码，这些代码是需要我们自己写，并在内核初始化的时候将例外处理代码拷贝到这个位置。关于更多例外向量位置的信息请参考《龙芯 2F 处理器用户手册》第六章——**处理器例外**。

2.1.2 龙芯 1c 开发板的 4 级中断例外处理

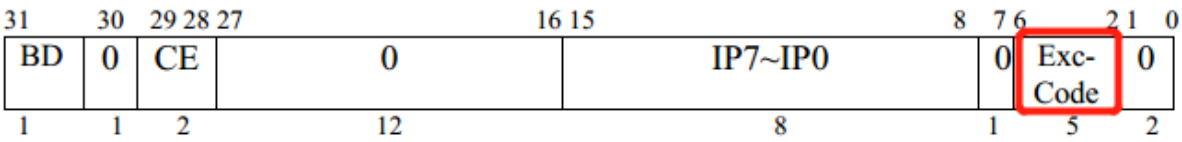
触发例外的情况有很多，中断就是例外的一种，中断又可以分为时钟中断、设备中断等，而设备中断又可以分为键盘中断、串口中断等。那么如何在发生一个例外后，准确的判断例外发生的原因，最后跳转到负责处理该例外的代码去执行呢？

其实在 MIPS 下，以中断处理为例，我们将中断例外的处理分为四级，每一级的处理过程如下：

第一级：各种情况下例外的总入口，即上小节提到的一个固定例外处理的入口，每当 CPU 发现一个例外，都会从执行地址跳转到这个例外向量入口，这也是 MIPS 架构下所有例外的总入口，这第一级的跳转是由硬件完成的，并不需要我们去实现；

第二级：这部分是处理例外的第二阶段，它主要完成对例外种类的确定，然后根据不同种类的例外，跳转到该例外的入口，**这部分代码也是需要我们拷贝到例外向量地址部分的代码（具体见 4.1.1 节）**。

对于例外种类的确定，是通过 CP0 的 cause 寄存器中 ExcCode 位域来区分不同例外的入口，CP0 CAUSE 寄存器的结构如下：



CP0_CAUSE 寄存器结构

以下为不同 Exccode 对应的例外类型，可以看到，当 ExcCode 为 0 的时候，说明触发的例外类型为中断(INT)：

例外代码	Mnemonic	描述
0	INT	中断
1	MOD	TLB 修改例外
2	TLBL	TLB 例外（读或者取指令）
3	TLBS	TLB 例外（存储）
4	ADEL	地址错误例外（读或者取指令）
5	ADES	地址错误例外（存储）
6	IBE	总线错误例外（取指令）
7	DBE	总线错误例外（数据引用：读或存储）
8	SYS	系统调用例外
9	BP	断点例外
10	RI	保留指令例外
11	CPU	协处理器不可用例外
12	OV	算术溢出例外
13	TR	陷阱例外
14	-	保留
15	FPE	浮点例外
16－22	-	保留
23	WATCH	WATCH 例外
24－30	-	保留
31	-	保留

Exccode 字段对应的例外种类

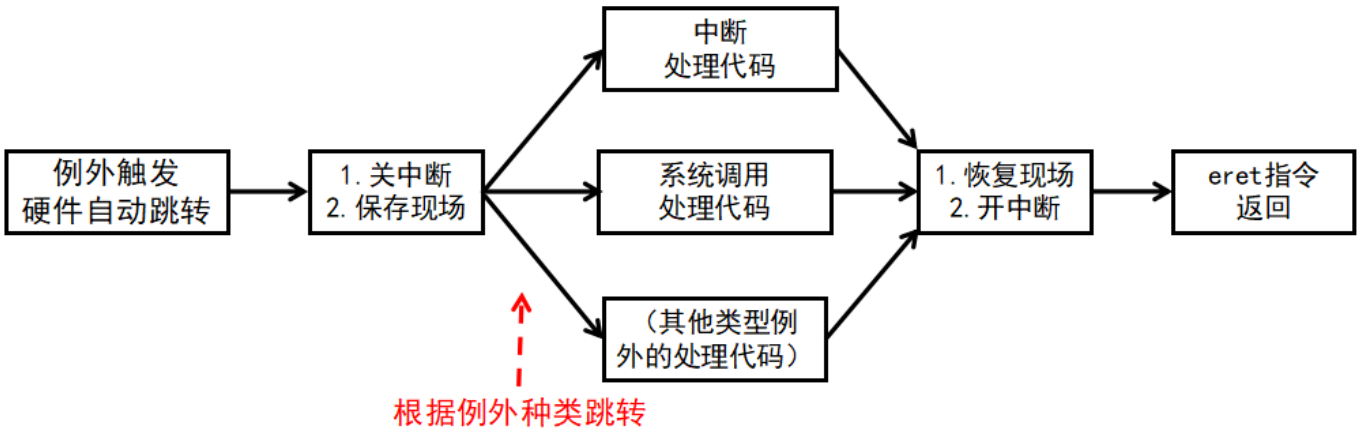
第三级：不同中断的各个入口，根据 IP7~IP0 的 8 位域（CP0_STATUS 寄存器中的）来区分，其中，时钟中断是 IP7=1，IP6~IP0 为 0，**时钟中断只需要判断到第三级，外设中断需要判断到第四级才能确定是哪种外设引起的中断**；

第四级：每个外设中断的入口（龙芯 1c 将所有外设分为五组，对应有 5 组中断寄存器组，查看 5 组寄存器中各个 SR 寄存器中哪个位域是 1，就对应是哪一种外设产生的中断），**具体可以参考《龙芯 1C300 处理器数据手册_v1.3》5.3 中断配置寄存器**。

通过这四级例外的查找，我们最终可以知道到底是哪种情况触发的中断，并对不同的中断进行处理。这四级例外除了第一级的例外处理入口的跳转是由硬件完成的，其余几级都是由软件编程完成的，是本次实验需要同学们完成的。

2.1.3 例外的处理

在处理例外处理前我们需要通过修改 CP0_STATUS 寄存器的值关中断，然后保存发生例外时的现场；对于例外的处理，针对不同的例外需要具体实现，触发例外的原因会以保存在 CP0_CAUSE 寄存器器中，在处理时根据例外触发代码跳转到不同的例外处理位置即可；在处理例外后对保存的现场进行还原，然后开中断，最后进行例外的返回。具体处理流程如下图：



例外处理流程

2.1.4 例外的返回

例外的返回是通过 eret 指令进行返回的。当一个例外发生时，硬件会自动的将发生例外的地址保存到 CP0_EPC 寄存器，之后跳到例外处理入口。当例外处理结束后，使用 eret 指令就可以返回 CP0_EPC 所指向的地址，也就是发生异常前运行到地址。

2.2 CP0 控制寄存器

MIPS 架构里除了 32 个通用寄存器以外还设计了 32 个 CP0 寄存器，用于处理例外，它们的功能各不相同，以下几个为本次实验会用到的 CP0 寄存器，如果同学们想了解更多，请详细参考《龙芯 2F 处理器用户手册》第五章——CP0 控制寄存器，里面对每个 CP0 寄存器有详细的说明。

2.3 时钟中断

在之前的任务里我们已经实现了任务的非抢占式调度，但是你可能已经看出了问题，那就是在我们的任务运行时，需要不断的使用 do_scheduler 方法去交出控制权，但其实在一个操作系统中，决定交不交出控制权的不是任务本身，而是操作系统。因此我们需要使用时钟中断去打断正在运行的任务，并在时钟中断的例外处理部分进行任务的切换，从而实现基于时间片的抢占式调度。

注：时钟中断的触发涉及 CP0_COUNT、CP0_COMPARE 寄存器，CP0_COUNT 寄存器的值每个时钟周期会自动增加，当 CP0_COUNT 和 CP0_COMPARE 寄存器的值相等时会触发一个时钟中断。

2.4 任务 3：时钟中断、基于优先级的抢占式调度

2.4.1 实验要求

（1）掌握 MIPS 下中断处理流程，实现中断处理逻辑。

（2）实现时钟中断处理逻辑，并基于时钟中断实现轮询式抢占式中断。

（3）在简单的调度方法上改进，实现**基于优先级**的抢占式调度。

（4）运行给定的测试任务，能正确输出结果。

2.4.2 文件说明

请基于 project2 第一阶段的代码继续进行实验。

2.4.3 实验步骤

（1）完善 main.c 中 init_exception 代码，主要完成异常处理相关的初始化内容：将例外处理代码拷贝到例外处理入口、初始化例外向量表，初始化 CP0_STATUS、CP0_COUNT、CP0_COMPARE 等异常处理相关寄存器。

（2）完善 entry.S 中 exception_handler_entry，主要完成例外处理入口相关内容：关中断、保存现场、根据 CP0_CAUSE 寄存器的例外触发状态跳转到中断处理函数（handle_int）。

（3）完善 entry.S 中 handle_int 代码，主要完成的内容为：跳转到**中断向量处理函数**（interrupt_helper 方法）、恢复现场、开中断。

（4）在 irq.c 中完善中断处理的相关代码中断向量处理（interrupt_helper 方法）、时钟中断处理函数（irq_timer 方法），具体内容请同学们自己思考如何实现。

（5）考虑调度算法，完善 sched.c 中 scheduler 方法，使其基于优先级和等待时间进行调度（优先级越高，越先进行调度；等待时间越长，越先进行调度）。

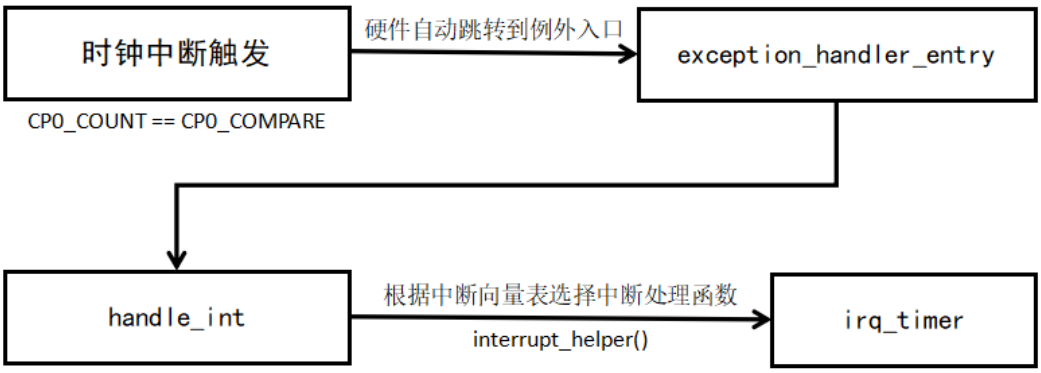
（6）修改 test.c 中 sched1_tasks 数组中的三个任务（去除 do_scheduler 方法的调用），要求打印出和任务 1 一样的正确结果。

2.4.4 注意事项

（1）由于 MIPS 下例外入口地址是固定的，因此我们要将我们自己实现的例外入口函数（exception_handler_entry）拷贝到该指定位置，具体实现请同学们自行思考。

（2）关于如何正确的开始一个任务的第一次调度，在抢占调度下是和非抢占调度是不同的，希望大家仔细思考如何在抢占调度模式下对一个任务发起第一次调度。

（3）时钟中断方法调用栈如下，请认真思考每个函数应该对应哪一级的例外处理：



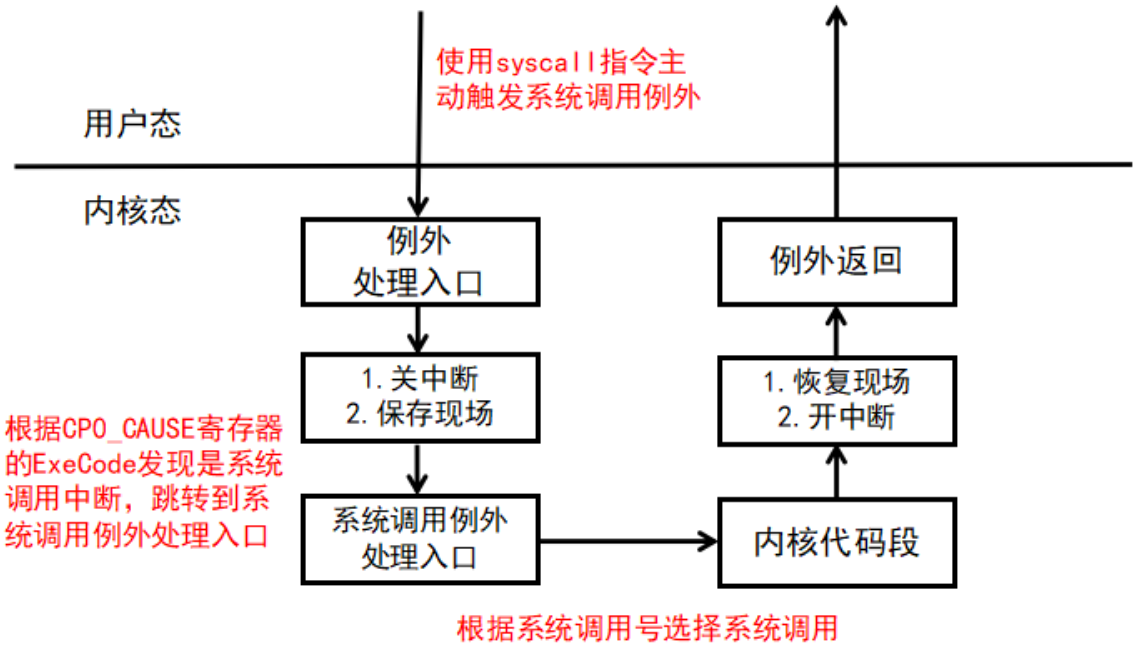
例外处理对应的函数栈

其中 exception_handler_entry 的代码就是例外处理中的第二级，handle_int 和 interrupt_helper 的代码就是例外处理中的第三级，irq_timer 代码就是处理时钟中断的具体代码。因为时钟中断处理不需要考虑第四级例外处理，因此只处理三级即可。

2.5 系统调用

之前的任务里，都是直接调用了内核的代码，但其实作为用户进程的任务是不允许直接访问内核代码段的，需要通过内核给出的访问接口去访问内核代码段，调用内核的功能。这个访问接口我们通常称之为**系统调用**。

系统调用也是例外的一种，只不过这种中断是用户主动触发的，我们触发系统调用中断的方式是在使用 syscall 汇编指令，当触发系统调用中断时和处理其他例外一样，处理器会自动跳入例外处理入口，保存用户态现场，然后进入到内核的系统调用处理代码段，当调用完内核代码后返回用户态现场，这和中断处理流程大致一样，具体的流程如下图：



系统调用处理流程

在以后的测试任务中，我们的任务都是用户进程，我们的实现代码在内核态，因此最后我们还需要对内核的代码实现一步系统调用的封装，提供给用户进程使用。

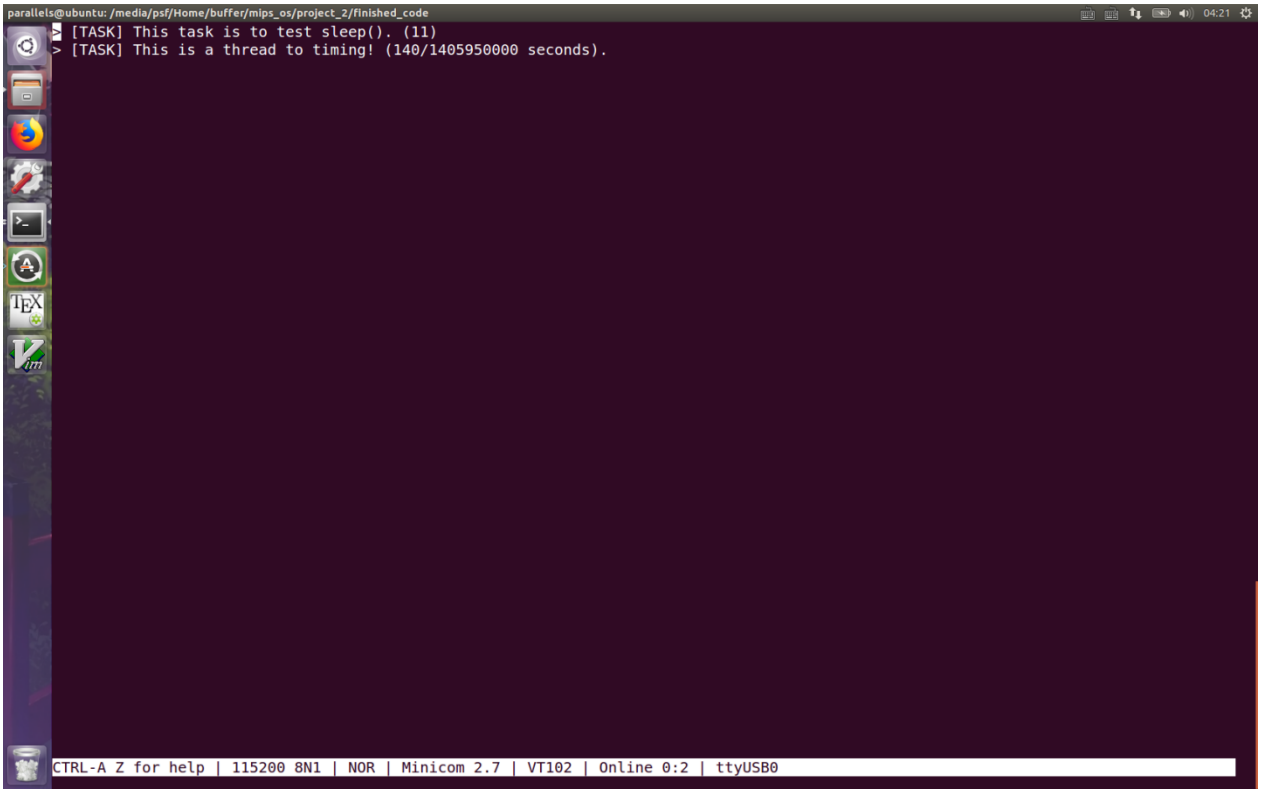
2.6 任务 4：系统调用

2.6.1 实验要求

（1）掌握 MIPS 下系统调用处理流程，实现系统调用处理逻辑。

（2）实现系统调用 sleep 方法。

（3）使用给出的测试任务，打印出正确结果。



2.6.2 文件说明

请基于 project2 任务三的代码继续进行实验。

2.6.3 实验步骤

（1）完善 main.c 中系统调用相关初始化（init_syscall）。

（2）完善 syscall.S 中的 invoke_syscall 方法，需要完善的内容为调用 syscall 指令发起一次系统调用。

（3）完善 entry.S 中 exception_handler_entry，需要完成的内容为：根据 CP0_CAUSE 寄存器的内容跳转到指定系统调用处理方法（handle_syscall）。

（4）完善 entry.S 中 handle_syscall 方法、syscall.c 的 system_call_helper 方法，需要完成的内容为根据系统调用号选择要跳转的系统调用函数进行跳转。

（5）实现 sched.c 中 do_sleep 方法以及 check_sleeping 方法，并实现其系统调用 sleep 方法。

（6）完善我们的用户态打印函数库 printf。

（7）实现我们之前完成的 lock 的系统调用。

（8）运行给定的测试任务（test.c 中 timer_tasks 数组中的三个任务、test.c 中 sched2_tasks 数组中的三个任务），要求打印出正确结果。

2.6.4 注意事项

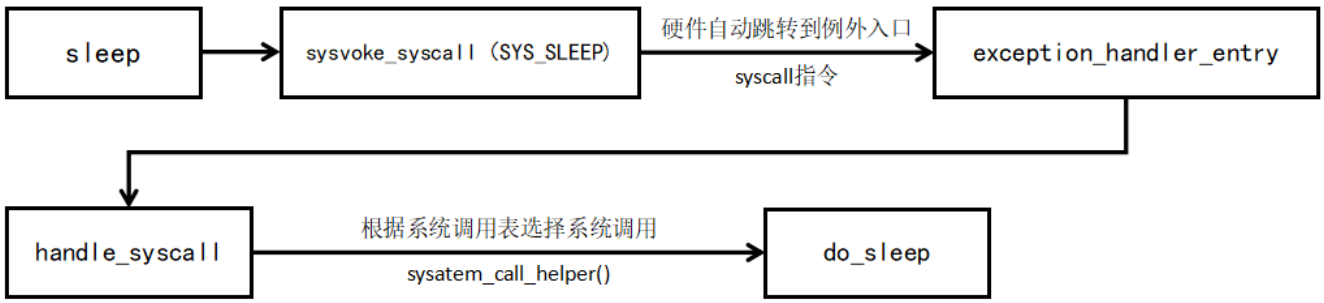
（1）sleep 方法的功能为将调用该方法的进程挂起到全局阻塞队列，当睡眠时间达到后再由调度器从阻塞队列（blocked queue）将其加入到就绪队列（ready queue）中继续运行。

（2）请认真思考系统调用模块的可拓展性，使得自己的设计便于拓展。

（3）了解 MIPS 下 syscall 指令的作用，该指令会触发系统调用例外。

（4）在我们之前的实验中我们一直是使用 printk 作为输出函数，在具备了系统调用模块后我们可以使用用户级的打印函数 pirntf，但是使用 printf 的前提是完成系统调用 sys_write（其实就是将 screen_write 封装为系统调用，在 printf 函数里调用），请参考第五节打印函数的内容了解 printf 函数。

（5）对于系统调用的处理不像中断处理那么繁琐，但处理的流程大致相似，请同学们仔细思考二者的区别，代码框架内正确的系统调用方法调用栈如下：



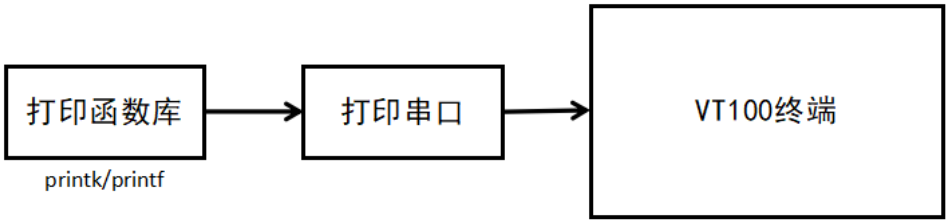
系统调用处理函数栈

三、 打印函数

关于该实验的打印驱动我们在实验的 start code 里实现了（位于 drivers 文件夹 screen.c 中），并分别给出了内核级的打印方法 **printk** 以及用户级的打印方法 **printf**（位于 libs 文件夹下）。为了方便同学们对该库的理解，同学们可以通过这章了解打印机制，并思考为何这么做。

3.1 VT100 控制码

对于开发板的打印，我们只能使用串口 IO，因此在输出的时候是往串口寄存器写字符，然后串口通过 VT100 虚拟终端最终呈现到屏幕的，如下图：



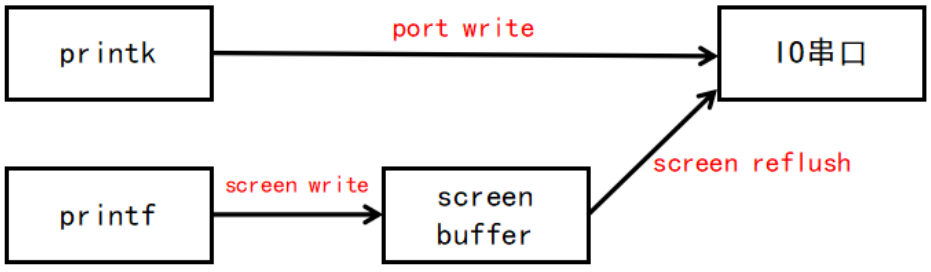
VT100 是一个终端类型定义，VT100 控制码是用来在终端扩展显示的代码，我们只需要往串口输入一些特定的字符串，就可以完成 VT100 终端的打印控制，比如光标移动，字体变色等功能，以下为部分控制码：

编号	控制码	描述	编号	控制码	描述
1	\033[0m	关闭所有属性	6	\033[y;xH	设置光标位置到（x,y）
2	\033[nA	光标上移 n 行	7	\033[2J	清屏
3	\033[nB	光标下移 n 行	8	\033[?25l	隐藏光标
4	\033[nC	光标左移 n 行	9	\033[?25h	显示光标
5	\033[nD	光标右移 n 行			

3.2 屏幕模拟驱动

由于我们开发板的打印只能通过串口一个字符一个字符的进行输出，没有显存这么一说，因此我们的做法就是在内存模拟一块显存，然后每次往模拟的显存里写数据（screen_write 方法），每次在**时钟中断处理函数（irq_timer）**里去一次性的将模拟显存里的数据刷新到串口里（screen_reflush 方法）。这么做的好处就是可以在处理一些进程对屏幕的占用时更加清楚，不会造成由于多任务模式下屏幕打印混乱的情况。

当然，这种做法只是对已经具备了中断的情况而言，因此屏幕模拟驱动只适用于用户及的 printf 方法，而对于内核级的 printk 方法，我们的做法依旧是每次还是直接往串口写数据。如下图：



3.3 打印优化

由于每个时钟周期都需要进行 screen buffer 的刷新，一般而言，buffer 的大小为 80 * 35 大小，因此每次需要刷新上千的字符到串口，这无疑是非常耗时的，因此我们可以只修改从这上一次刷新到这一次刷新期间修改过位置的字符，这样就可以大大的提升我们的打印速度了。关于优化的具体实现可以参考 driver/screen.c 中 screen_reflush 方法。