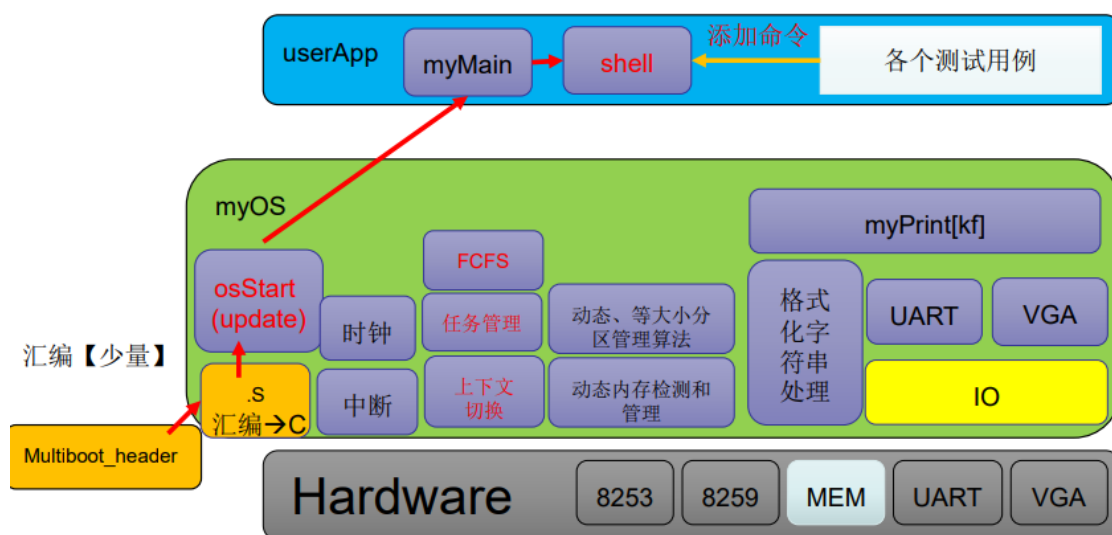


Lab 5 实验报告

I. 程序框图



与上一个实验相比，本次实验主要实现了上下文切换和简单的FCFS算法，多进程的组织是静态实现的：在一个`init`进程中预先写入创建新进程的代码，并为每个创建的新进程提供执行函数；按照创建进程的顺序执行完整的进程。OS处理多进程的流程如下：

1. 操作系统启动流程`osStart()`中调用`TaskManagerInit()`准备进入多任务调度模式。这个过程中实现了任务池、进程就绪队列和空闲TCB链表的初始化，并且创建了`idle`和`init`任务。
2. `TaskManagerInit()`中调用`startMultitask()`，进入多任务调度模式。关键步骤为上下文切换，从OS系统栈切换到`init`进程栈。
3. 执行`init`进程主程序`myMain()`。其中创建了若干个子进程。
4. `myMain()`结束前调用`tskEnd()`终止并销毁进程，它还会调用进程调度函数，进行上下文切换，切换到下一个进程。
5. 后续进程的执行类似3，4步骤。

在这个实现中，虽然 `idle` 进程有对应的主函数和进程栈，但CPU并未执行到主函数 `tskIdleBdy()`，`esp` 寄存器也并未切换到它的栈中；而是通过用户程序直接调用调度函数实现进程的切换。

II. 代码实现

主要理解三个数据结构：任务池，进程就绪队列和空闲TCB链表。

任务池静态实现，大小固定，各个任务的ID即为数组下标；进程就绪队列和空闲TCB链表通过 `nextTCB` 字段在任务池上实现链接。

就绪队列相关函数的实现类似于常规的FIFO队列，主要维护头尾指针。

进程创建函数 `createTsk()` 中，先取出空闲TCB，维护空闲TCB链表；再根据参数 `tskBoody` 初始化TCB；最后将新进程加入就绪队列 `rqFCFS` 中。

`destroyTsk()` 在进程终止后销毁进程，并且将对应TCB加入空闲TCB链表中。

III. 思考题

1.

在上下文切换的现场维护中，`pushf` 和 `popf` 对应，`pusha` 和 `popa` 对应，`call` 和 `ret` 对应，但是为什么 `CTS_SW` 函数中只有 `ret` 而没有 `call` 呢？

因为 `movl prevTSK_StackPtr, %eax movl %esp, (%eax) movl nextTSK_StackPtr, %esp` 不需要组织为一个函数，并且与 `call` 对应的 `ret` 应该位于被调用函数中，而非 `call` 返回之后；此处 `ret` 的作用是从栈中取出 `eip` 字段存入 `eip` 寄存器中。

2.

谈一谈你对 `stack_init` 函数的理解。

`stack_init()` 函数有两个参数：`stk` 是栈顶指针(变量)的地址，初始时栈顶指针指向栈底；`task` 是进程的主函数。

函数第一行 `*(stk)-- = (unsigned long) 0x08;` 先往栈顶指针指向的栈顶(当前也是栈底)写入 `0x08`，然后移动栈顶指针到下一个位置，这就相对于一个入栈的操作。类似地，后续陆续入栈了 `EIP`，`FLAG` 寄存器，`EAX`，`ECX` 等寄存器的初始值，其中 `EIP` 寄存器存储指令地址，因此它应该初始化为函数指针 `task`。

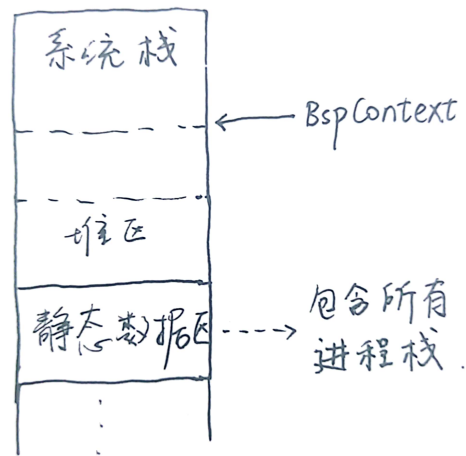
3.

`myTCB` 结构体定义中的 `stack[STACK_SIZE]` 的作用是什么？`BspContextBase[STACK_SIZE]` 的作用又是什么？

- `stack[STACK_SIZE]` 的作用：作为进程执行时的私有栈，存储进程执行过程中的局部数据；存储上下文信息，用于上下文切换。
- `BspContextBase[STACK_SIZE]` 没有任何作用，实际上起作用的是 `BspContext`，它指向系统栈的栈顶。

`startMultitask()` 函数中设置变量 `BspContext` 指向这个栈的栈顶，再以 `BspContext` 的地址为第一个参数调用 `CTX_SW()`；而在 `CTX_SW()` 中，入栈操作对应的是 OS 系统栈中 `startMultitask()` 函数对应的栈帧，因此 `movl %esp, (%eax)` 指令中写入 `BspContext` 的值是该栈帧的顶部地址，也即系统栈的栈顶地址。

如下图：



4.

`prevTSK_StackPtr` 是一级指针还是二级指针？为什么？

`prevTSK_StackPtr` 是二级指针，因为不论在 `startMultitsk()` 中 (`prevTSK_StackPtr = &BspContext;`)，还是在 `context_switch()` 中 (`prevTSK_StackPtr = &(prevTsk->stkTop);`)，它都指向一个指针类型的变量。

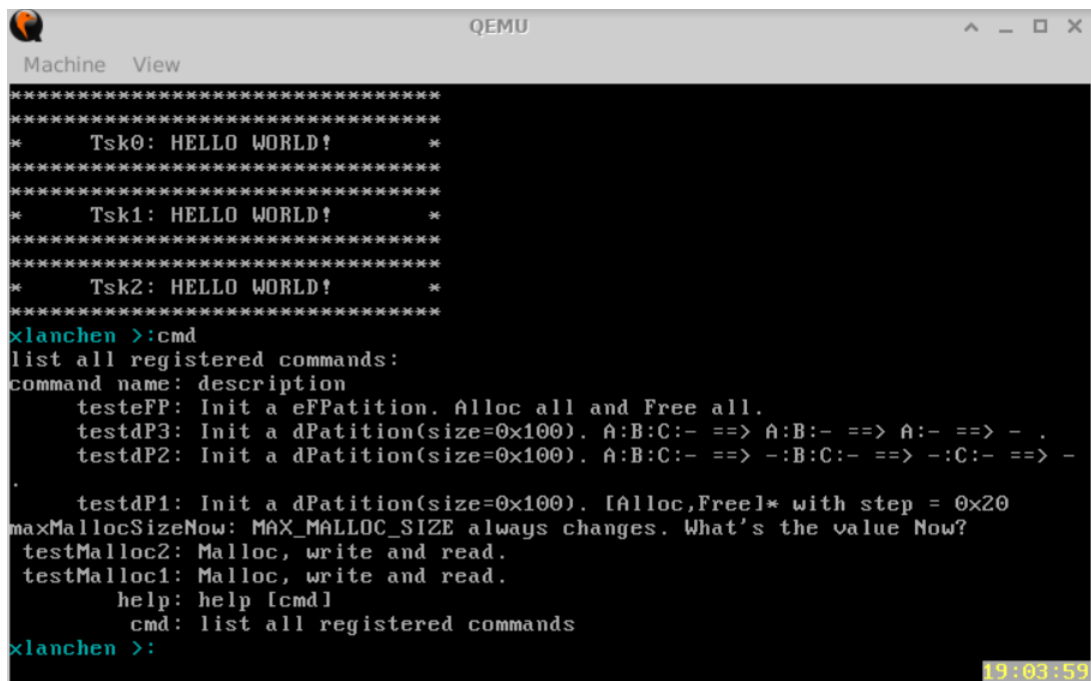
IV. 实验结果

进入项目路径，执行指令 `./source2img.sh` 进行编译、链接并运行，QEMU窗口显式如下结果：



```
Machine View
*****
*      INIT  INIT  !      *
*****
*      Tsk0: HELLO WORLD!  *
*****
*      Tsk1: HELLO WORLD!  *
*****
*      Tsk2: HELLO WORLD!  *
*****
xlanchen >:
19:02:26
```

再输入 `sudo screen /dev/pts/2` 重定向终端，输入 `cmd` 命令，得到如下结果：



```
Machine View
*****
*      Tsk0: HELLO WORLD!  *
*****
*      Tsk1: HELLO WORLD!  *
*****
*      Tsk2: HELLO WORLD!  *
*****
xlanchen >:cmd
list all registered commands:
command name: description
  testeFP: Init a eFPatition. Alloc all and Free all.
  testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
  testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
  .
  testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
  help: help [cmd]
  cmd: list all registered commands
xlanchen >:
19:03:59
```

可以看到进程 `init` 先执行，然后分别是进程 `Tsk0`, `Tsk1`, `Tsk2`，最后执行 `shell` 进程，这符合 FCFS 对进程的调度方式。