Lab5 & 6 实验文档

0. 前言

Lab5 & 6 的主要内容都是任务调度。

Lab5 主要完成任务的数据结构以及上下文切换，在此基础上编写 FCFS 调度算法，最终用静态到达的任务进行测试。

Lab6 在 lab5 的基础之上，实现更多的调度算法。此外，需要实现任务随时间动态到达，也就是说，需要与 lab3 中的时钟中断结合起来。

因两次实验的关联性，完成 lab6 则默认完成了 lab5，有能力的同学可以选择直接完成 lab6。当然，无论你从哪一个实验开始，都希望你能按顺序完整阅读这个文档。

另外，按照设想，本课程实验应是"设计性"的，即希望大家自己动手设计、实现功能，从零开始一步步构建起一个迷你的操作系统。然而因为种种原因，前几次实验都未能达到这一目标。同时，我们也注意到有不少同学对前几次实验框架中的实现方法抱有不同见解。因此，本次实验，只提供一套可运行的 lab4 代码（在 lab4 截止后发布，不强制要求使用，可直接在自己的 lab4 上继续实验），新增的功能模块请同学们自己添加。

1. Lab 5

2020 年 lab5 讲解视频链接：https://www.bilibili.com/video/BV1iK4y1X7M3 （感谢上传视频的往届助教）

1.1 任务的数据结构

本实验中实现的任务可以看作进程的简化模型。在理论课程的学习中，我们知道操作系统通过进程控制块（Process Control Block，PCB）来控制和管理进程，在本实验中，我们需要实现一个类似的数据结构，称为任务控制块（Task Contorl Block），命名为 `myTCB`。

此数据结构中应**至少**包含以下内容：

- 任务 ID（`tid`）
- 状态：就绪、等待等，按需设计
- 栈：可以考虑用一个栈顶指针进行维护，注意栈是反向增长的。栈空间应使用 lab4 实现的 `kmalloc` 分配，用 `kfree` 回收，注意 `kmalloc` 返回的是所分配空间的低地址，而初始栈顶指针应为于栈底（这片空间最高的地址）。栈大小请自行决定。
- 调度相关的参数：按需添加，例如 lab5 中可以只有任务的到达时间，lab6 中则进一步添加优先级等。为了方便 lab6 拓展，可以考虑将参数设置为通用接口。

1.2 任务就绪队列

为管理任务调度，还需实现一个就绪队列，它的元素是 `myTCB`。对于 FCFS，你可以实现一个 FIFO 队列，将任务按照到达时间的顺序插入其中。

1.3 任务的创建和销毁

需要实现任务的创建和销毁两种原语。有两种实现思路：

静态：提前分配好一定数量（可自行配置）的 `myTCB`，存放在数组（任务池）中。创建任务时，直接从任务池中取出一个空闲的 `myTCB`；销毁时则将其重新设置为空闲，释放回任务池

动态：按需分配。创建任务时，使用 `kmalloc` 为 `myTCB` 分配空间；销毁时，使用 `kfree` 释放

无论采取哪种思路，对外封装的接口应是统一的：

- `int createTsk(void (*tskBody)(void))`：创建任务，传入参数为任务的入口函数，返回创建的任务的 `tid`

- `void destroyTsk(int tskIndex)`：销毁任务，传入参数为 `tid`

1.4 任务的启动和结束

需要实现任务的启动和终止两种原语，它们分别是：

- `void tskStart(myTCB *tsk)`：创建好任务后，需要启动任务时，调用此原语。传入参数是任务的 TCB，原语行为是启动任务，将任务状态设置为就绪，然后插入就绪队列。
- `void tskEnd()`：此原语在某个任务执行结束后被调用。其行为是销毁当前任务，并通知操作系统可以进行调度、开始下一个任务。

  此原语可以手动添加在每个任务函数的末尾，也可以选择将其入口地址压入进程栈的最底部，等任务函数结束后通过 `ret` 返回时，自动调用，类似 1.5 中的 `*(*stk)-- = (unsigned long)task;` 这一行代码。

注：老师 ppt 上，在 `tskEnd()` 中才将任务移出就绪队列，但你也可以选择在任务开始运行前就将其移出就绪队列。总而言之，你可以采取自己觉得合理的方式。无论采用什么方法，请与自己编写的就绪队列数据结构相对应。

1.5 上下文切换

上下文切换是任务调度中重要的部分，此处我们提供相关代码：

```
.text
.code32

.global CTX_SW
CTX_SW:
    pushf
    pusha
    movl prevTSK_StackPtrAddr, %eax
    movl %esp, (%eax)
    movl nextTSK_StackPtr, %esp
    popa
    popf
    ret
```

下面逐行介绍上述代码：

1. `pushf`：旧进程的标志寄存器入栈
2. `pusha`：旧进程的通用寄存器入栈，此条指令和上一条指令一并，起到了保护现场的作用
3. `movl prevTSK_StackPtrAddr, %eax`：`prevTSK_StackPtrAddr` 是存放旧进程栈顶指针值的地址，此行指令将其存入 `eax` 寄存器。注意，栈顶指针值也是一个地址
4. `movl %esp, (%eax)`：`()` 是访存的标志。此行指令是将当前栈顶指针值（也就是旧进程的栈顶指针值）存入地址为 `eax` 的内存中，也就是上一条指令中的 `prevTSK_StackPtrAddr` 这一地址上
5. `movl nextTSK_StackPtr, %esp`：新进程的栈顶指针值存入 `esp` 寄存器，这样就切换到了新进程的栈帧上
6. `popa`：新进程的通用寄存器出栈
7. `popf`：新进程的标志寄存器出栈，此条指令和上一条指令一并，起到了恢复现场的作用
8. `ret`：返回指令，从栈中取出返回地址，存入 `eip` 寄存器（其作用类似于组成原理中所讲的 `PC` 寄存器）。这个返回地址在由 `call` 指令调用 `CTX_SW` 时就已入栈。

注：

- 上述代码中的 `prevTSK_StackPtrAddr` 和 `nextTSK_StackPtr`，可以用下面的方法在 C 程序中赋值：

```
unsigned long **prevTSK_StackPtrAddr;
unsigned long *nextTSK_StackPtr;
void context_switch(unsigned long **prevTskStkAddr, unsigned long *nextTskStk) {
    prevTSK_StackPtrAddr = prevTskStkAddr;
    nextTSK_StackPtr = nextTskStk;
    CTX_SW();
}
```

- `CTX_SW` 切换不同进程的栈帧只修改了栈顶指针 `esp`，未修改栈底指针 `ebp`。追求严谨的同学也可以用类似的方式修改 `ebp`。

下面以一个具体例子说明上下文切换的过程：

任务 a 正在运行，此时，因某个原因，需要切换到任务 b。做好准备后，通过 `call` 调用 `CTX_SW` 这一过程，`call` 指令会将此时 a 的下一条指令地址存入栈中，以便未来切换回 a 时取出，进行返回。进入 `CTX_SW` 后，将任务 a 的标志寄存器和通用寄存器压入 a 的栈帧，并将 a 此时的栈顶指针值保存进 `prevTSK_StackPtrAddr`。随后，通过设置 `esp`，将栈顶指针切换到任务 b 的栈顶 `nextTSK_StackPtr` 上，从 b 的栈中取出此前 b 已经保存好的通用寄存器和标志寄存器，再返回。返回通过 `ret` 指令实现，它会从栈中取出 b 的返回地址，存入 `eip`，以继续 b 的任务函数的运行。

或许你已经发现了一个问题：假如初始创建了进程 a、b，他们都从未进行过上下文切换。先运行 a，然后调用上述代码切换到 b，此时，b 的栈中并没有存储通用寄存器、标志寄存器和返回地址，那么运行 `popa`，`popf`，`ret`，岂不是会出错？

所以，这要求我们在创建一个进程时，就将一些内容压入栈中，这些内容模拟的就是标志寄存器、通用寄存器和返回地址。

例如：

```
void stack_init(unsigned long **stk, void (*task)(void)) {
    *(*stk)-- = (unsigned long)0x08; // CS

    *(*stk)-- = (unsigned long)task; // eip

    // pushf
    *(*stk)-- = (unsigned long)0x0202; // flag registers

    // pusha
    *(*stk)-- = (unsigned long)0xAAAAAAAA; // eax
    *(*stk)-- = (unsigned long)0xCCCCCCCC; // ecx
    *(*stk)-- = (unsigned long)0xDDDDDDDD; // edx
    *(*stk)-- = (unsigned long)0xBBBBBBBB; // ebx
    *(*stk)-- = (unsigned long)0x44444444; // esp
    *(*stk)-- = (unsigned long)0x55555555; // ebp
    *(*stk)-- = (unsigned long)0x66666666; // esi
    **stk     = (unsigned long)0x77777777; // edi
}
```

首行压入的是 `CS` 寄存器值，`CS` 是代码段寄存器。此行代码源于老师的讲解视频，但实际上我们上下文切换后使用的是 `ret` 指令，不会进行 `pop CS`，所以这行可以随意。

接着压入 `eip` 寄存器值，也就是 `ret` 指令所需的返回地址。因为是初始创建，直接压入任务的入口函数地址。

然后模拟 `pushf` 和 `pusha`，压入标志寄存器和通用寄存器。

1.6 调度函数

讲了这么多，究竟如何调度呢？这里提供一种针对于 FCFS 的参考思路：

实现一个函数 `schedule()`，在里面编写具体的调度逻辑。对于 FCFS，就是从就绪队列中获取下一个任务的 TCB，通过上下文切换函数（因为我们已经初始化过进程的栈），开始此任务的运行。

`schedule()` 函数的调用时机可以是每个任务结束时，即 `tskEnd()`。

1.7 idle 任务和 init 任务

idle 任务是闲置任务，当就绪队列中没有任务时，切换到 idle。idle 的行为是：一个死循环，不断调用 `schedule`，检查是否有新任务到来。如果有，会切换到新任务（当然，对于静态任务的 lab5，并不会有新任务到来。这项功能主要针对于 lab6）

init 任务是初始任务，其任务函数体为 `initTskBdy()`。本实验中，使用用户程序的入口 `myMain()` 对接 `initTskBdy()`

1.8 任务管理器初始化和进入多任务运行

在 `osStrat()` 函数中，添加任务管理器的初始化。初始化可以包括：

- 进程池初始化
- 初始化就绪队列
- 创建 idle 任务
- 创建 init 任务并插入就绪队列
- 进入多任务运行

进入多任务运行的参考方法：

创建一个初始的、形式上的上下文环境（栈），然后获取就绪队列首个任务（通常为 `init`），利用上下文切换函数切换到它。

例如：

```
unsigned long BspContextBase[STACK_SIZE];
unsigned long *BspContext;
void startMultitask(void) {
    BspContext = BspContextBase + STACK_SIZE - 1;
    firstTsk = nextFCFSTsk();
    context_switch(&BspContext, firstTsk->stkTop);
}
```

1.9 运行和测试

上面提到，init 任务对接的是用户程序入口 `myMain()`，测试用例请自行在用户程序里编写。具体来说，请创建若干个任务，它们具有不同的到达时间参数，它们的任务函数可以是输出自己的任务信息。注意，shell 也必须封装成一个任务。然后，逐个启动这些任务（插入就绪队列），观察它们的运行顺序是否符合 FCFS。

在你的报告中，请简单介绍测试用例信息，以及它们是如何与运行结果对应的。助教检查时，会阅读你的介绍，对比实际运行结果，判断是否符合要求。

1.10 目录组织参考

只给出新增模块的目录组织；仅供参考，不要求与此保持一致：

`myOS/i386/CTX_SW.S`：上下文切换的汇编代码

`myOS/include/task.h` 和 `myOS/task/task.c`：任务的数据结构，各个任务原语，上下文切换，任务管理器初始化，开始多任务运行，等

`myOS/include/taskQueueFIFO.h` 和 `myOS/task/taskQueueFIFO.c`：任务的 FIFO 队列

`myOS/include/scheduler.h` 和 `myOS/scheduler/scheduler.c`：调度算法

创建新的目录、文件后，请不要忘记添加或修改 Makefile，其内容请参考其他子目录的 Makefile

lab6 直接在 `myOS/task`、`myOS/scheduler`、`myOS/include` 中进行拓展即可

2. Lab 6

2020 年 lab6 讲解视频链接：https://www.bilibili.com/video/BV1Mb4y1d7ZP （感谢上传视频的往届助教）

2.1 TCB 调度参数扩展

本次实验需要实现多种调度算法，因此需要对 `myTCB` 的调度参数进行一定扩展。例如添加优先级、执行时间（用于 SJF，认为已知，创建时设置），等等。

任务参数要求使用通用接口：

数据结构命名为 `tskPara`，内部成员包含多种调度参数，例如：

```
typedef struct tskPara {
    usigined int priority;
    usigined int arrTime;
    usigined int exeTime;
} tskPara;
```

任务参数的操作接口：

```
void initTskPara(tskPara *buffer);
void setTskPara(unsigned int option, unsigned int value, tskPara *buffer);
void getTskPara(unsigned int option, unsigned int *para, tskPara *buffer)
```

分别是初始化、设置某项参数、获取某项参数值。`option` 指定了操作的参数项，它的设计应与 `tskPara` 的内部成员对应，例如：

```
#define PRIORITY 1
#define EXETIME  2
#define ARRTIME  3
```

2.2 调度相关接口与调度器

本次需要编写多个调度算法，但要求在最顶层封装统一的接口。

每一种调度需要提供一个调度器数据结构，具体可以参考老师 ppt 第 10 、第 12 页（不需要完全一致）。

调度器数据结构中，有一个名为 `type` 的成员，它用于标识相应的调度算法的类型，你可以通过宏来定义各个调度算法的 `type` 值，例如：

```
#define FCFS 1
#define RR   2
```

实际使用时，用以下接口设置调度算法：

```
void setSysScheduler(unsigned int what);
```

参数 `what` 就是你所选择的调度算法的 `type`。此函数会将相应调度器里的调度接口与顶层的统一调度接口进行对接。

老师 ppt 上还给出了 `getSysScheduler`、`getSysSchedulerPara` 等函数，这些函数不强制要求，可以选择性地按需实现。

2.3 结合时钟中断的任务到达与执行

本次实验相比于 lab5 最大的不同，是任务的到达和执行都是动态的，与 lab3 中的时钟中断 tick 相关。具体效果可以参考助教上传的演示视频。

- 任务随时钟动态到达

  每个任务创建时，必须指定一个到达时间，然后将其插入一个到达队列中（到达时间最小任务永远位于队首）。每次 tick，检查到达队列队首任务的到达时间，如果系统当前时间已经等于或大于到达时间，则开始此任务（使用 `tskStart()`）。

  如何在每次 tick 检查到达队列？请回忆一下 lab3 提到的 hook 机制，其实质是在 tick 相关模块中维护一个 hook 函数列表，可以动态向此列表中注册新函数（对外提供动态注册接口），每次 tick 会调用此列表中的所有函数。lab3 中，更新墙钟正是一个 tick 的 hook 函数。在本实验中，你同样可以将检查到达队列的函数作为一个 hook 函数，注册上去。

- 任务执行

  每个任务需要指定一个执行时间，表示任务函数会执行多久。

  任务函数的实际执行时间需要与此变量指定的执行时间（近似）相等，为实现这一点，一种思路是在任务函数中进行相应时间的等待。其他实现方法也是允许的。

2.4 调度算法要求

本次要求实现任意三种调度算法，**其中至少有一种抢占式调度算法**，例如 RR、抢占式优先级调度等。另外两种可以随意选择。

如果你选择的调度算法与时钟中断相关，例如 RR，同样可以使用 hook 机制向 tick 注册相关函数。

建议：实现一个具有通用比较函数接口的优先队列，以应对不同调度算法。

2.5 运行和测试

实际运行时选择哪种调度算法，可以在用户程序中指定（参考老师 ppt 14 页），也可以在系统启动之前通过手动输入来指定（参考助教上传的视频）。

关于测试，类似 lab5，你同样需要在用户程序中编写每个调度算法对应的测试用例，并在报告中加以介绍。助教检查时，会阅读你的介绍，对比实际运行结果，判断是否达到算法的预期效果。

3. 实验评分准则

只要完成 lab6 的任意一种调度算法，则 lab5 默认为满分

实验要求与 ppt 有出入时，以本文档为准

- Lab5：满分 100

    - 完成 myTCB 数据结构：5 分
    - 实现就绪队列的数据结构：15 分
    - 完成任务的创建、销毁原语：5 分
    - 完成上下文切换和调度函数：30 分
    - 完成任务的启动、结束原语：5 分
    - 完成任务管理器的初始化，可以正常进入多任务运行状态：10 分
    - 编写测试用例（含 shell 的封装），最终运行结果与测试用例相符合：10 分
    - 实验报告：20 分
- Lab6：满分 100
    - 实现任务随时钟动态化到达：15 分
    - 调度器和任务参数采用统一接口：5 分
    - 完成一种抢占式调度算法：30 分
    - 完成另外两种任意调度算法：40 分，每种 20 分
    - 实验报告：10 分

4. 实验报告要求

4.1 画出系统结构框图 (30%)

形式上，参考老师 ppt “软件架构和功能”，**请自己动手画图**（推荐使用 draw.io 画图）

4.2 介绍主要功能模块及其实现方法 (30%)

结合你的源代码，说明你是如何实现预期功能的

4.3 源代码组织说明（目录组织、Makefile 组织） (10%)

- 目录组织：以树形结构展示你的源码目录
- Makefile 组织：给出所有 Makefile 路径变量（如 OS_OBJS）和 目标文件 .o 的组织方式，例如：

```
.
├── MULTI_BOOT_HEADER
│     └── output/multibootheader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
    │    ├── ...
    │    ├── ...
    │    ├── ...
    └── USER_APP_OBJS
         └── ...
```

4.4 编译和运行方法说明 （可选）

如果你的实验需要使用特殊的编译与运行方法，请在此说明

4.5 测试用例与运行结果说明 (30%)

简单介绍你编写的测试用例，以及它是如何与运行结果进行对应的

4.6 遇到的问题和解决方案说明 （可选）

5. 提交要求

lab5 与 lab6 有各自的提交入口，请分开提交

提交内容应为 **实验报告** + **源代码**，整个文件夹目录结构如下：

```
.
├── doc
|   └── report.pdf（实验报告）
└── src
    ├── ...（源代码）
    └── source2img.sh（运行脚本）
```

请将此文件夹打包为压缩包（格式不限），命名为 `学号_姓名_lab5` / `学号_姓名_lab6`，在 bb 系统作业区提交

提交 DDL：暂定于期末考试前一天 23:59，后续如有调整会及时通知