Lab3 Scheduling 进程调度

操作系统实验指导书 - 2024秋季 [实验概述 (https://elearning.fudan.edu.cn/courses/78523/pages/shi-yan-gai-shu)](https://elearning.fudan.edu.cn/courses/78523/pages/shi-yan-gai-shu) | [常见问题汇总 (https://elearning.fudan.edu.cn/courses/78523/pages/chang-jian-wen-ti-hui-zong)](https://elearning.fudan.edu.cn/courses/78523/pages/chang-jian-wen-ti-hui-zong) | [Lab1 UTIL (https://elearning.fudan.edu.cn/courses/78523/pages/lab1-util-xv6yu-unixying-yong-cheng-xu)](https://elearning.fudan.edu.cn/courses/78523/pages/lab1-util-xv6yu-unixying-yong-cheng-xu) | [Lab2 SYSCALL (https://elearning.fudan.edu.cn/courses/78523/pages/lab2-syscall-xi-tong-diao-yong)](https://elearning.fudan.edu.cn/courses/78523/pages/lab2-syscall-xi-tong-diao-yong) | **Lab3 Scheduling** | [Lab4 Page table (https://elearning.fudan.edu.cn/courses/78523/pages/lab4-page-table-ye-biao)](https://elearning.fudan.edu.cn/courses/78523/pages/lab4-page-table-ye-biao)

一、实验目标

原有的xv6的进程调度实现是一个均衡的调度模式，它会不断地循环找到一个可以运行的进程，切换到这个可以运行的进程，运行的进程结束之后返回控制权给调度器，调度器再次重复上述的这些操作。

这个lab会在原有的Round-Robin机制调度的基础上进行扩展，增加了进程优先级项目，要求大家实现一个考虑进程优先度的调度模式，并且通过实现扩展的系统调用对于不同调度模式下程序的表现行为进行分析。

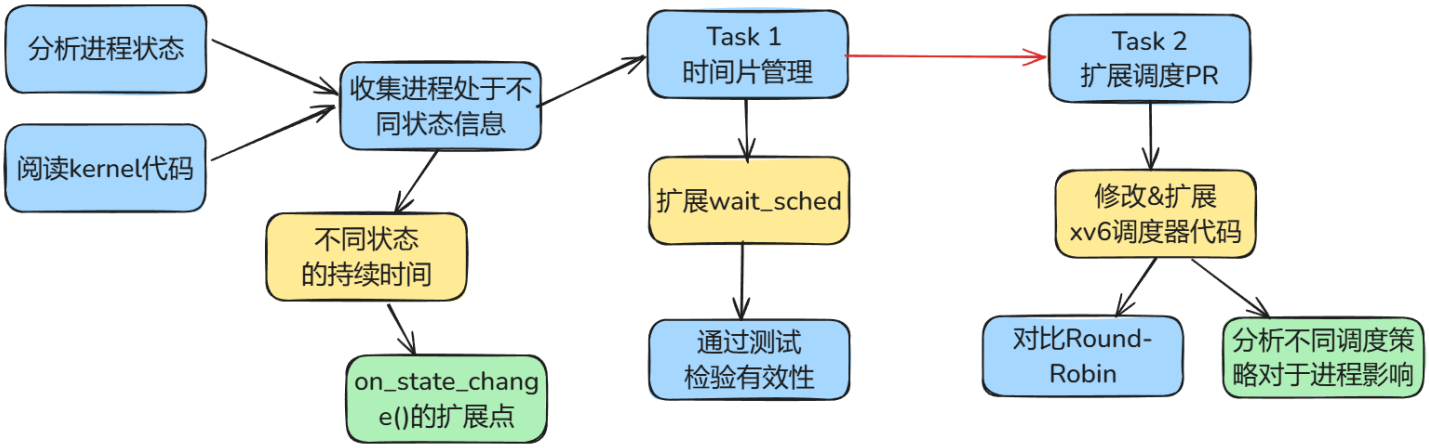
二、实验准备

首先确保你已经拥有远程仓库的最新代码，然后切换到syscall分支后进行开发。

```
$ git fetch
$ git checkout scheduler
$ make clean
```

在切换分支之前，记得通过`git commit -m`保存实验二的内容。 如果你希望舍弃实验二的更改，也可以使用-f选项强制切换分支，例如`git checkout -f scheduler`。

三、实验内容



任务一：获取进程运行状态

实验目标

在该任务中，你需要在xv6加入具有获取进程运行功能的 `wait_sched` 系统调用。

它可以统计xv6中，用户通过fork系统调用生成的子进程处于不同状态时间的信息，这也为后续对比进程被不同调度器调度的时候产生的不同影响提供了数据源。

具体要求：添加一个名为 `wait_sched` 的系统调用

```
// get the running time, sleeping time, runnable time when the child process returns
int wait_sched(int *runable_time, int *running_time, int *sleep_time)
```

1. **输入**包括：获取进程处于RUNNABLE，RUNNING，SLEEPING等状态下的时间，单位为xv6内部触发中断的间隔数ticks，ticks是xv6触发时间中断自增的一个变量，可以相对反映出一个进程的运行时长。
2. **输出**包括：状态码 -1 表示出错，成功则返回pid
3. **功能**：在完成wait的基本逻辑之外，额外返回父进程fork出来的子进程的运行状态，父进程在子进程结束的时刻通过wait_sched可以获取到OS调度进程（调度的对象是我们fork出的子进程）的相关信息。

关于wait_sched这一系统调用，在proc.c中已经为大家写好了proc.c的函数接口，proc.h中写好了所需要增加的成员变量，如下面所示：

```
// Per-process state
struct proc {
  ...
  // Newly added for Lab3
  uint64 created_time;         // creation time
  uint64 finish_time;          // finish time
  uint64 running_time;         // running time
  uint64 runable_time;         // runnable time
  uint64 sleep_time;           // sleeping time
  uint64 start;                // start time of a process state
  uint64 end;                  // end time of a process state
```

```
};
```

具体要求：实现`on_state_change`在进程状态发生变化时收集运行时长

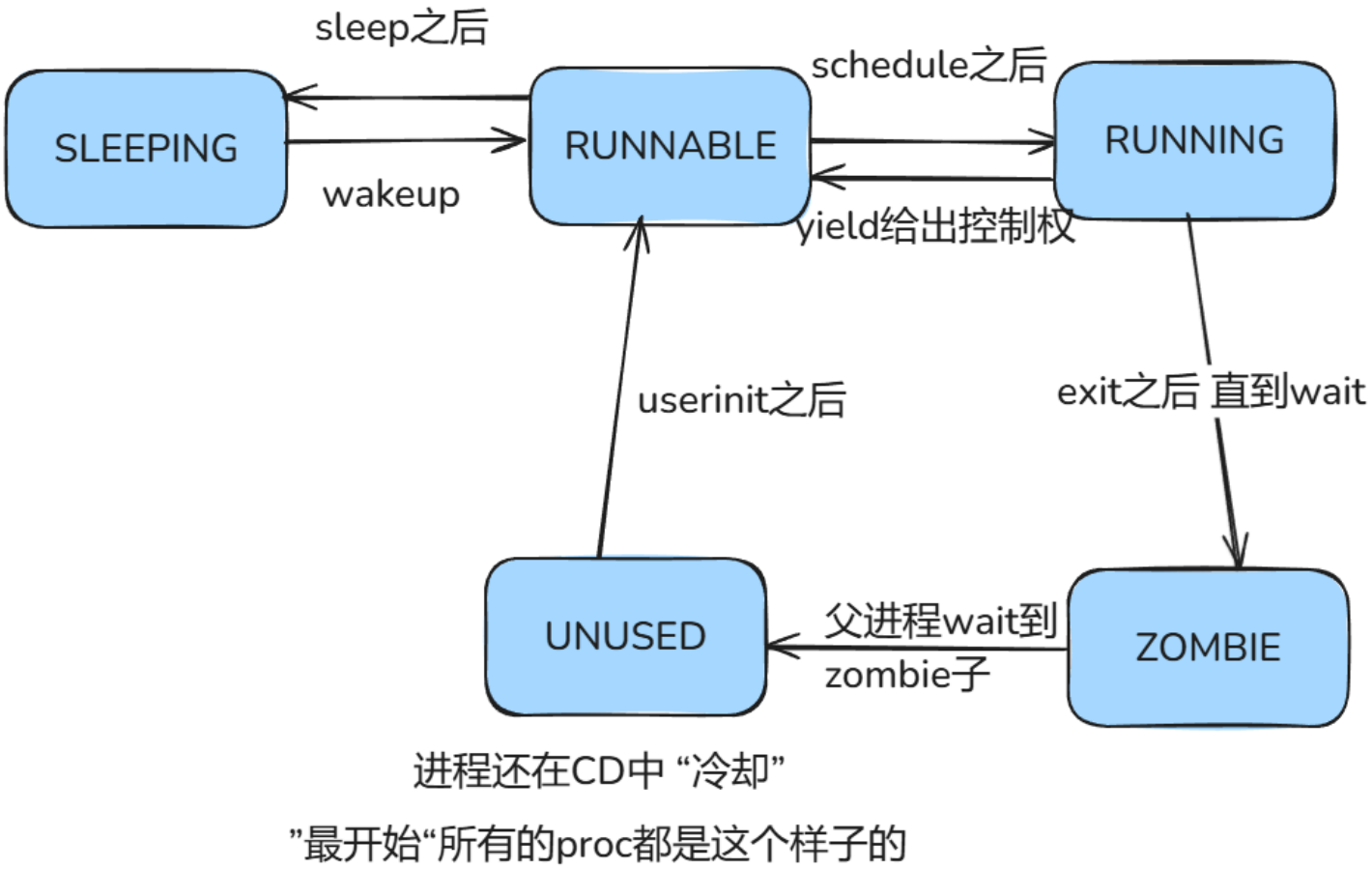
除此之外，为了能够让上面的系统调用能够收集到数据，以及降低代码实现的难度，我们已提供 on_state_change 这一函数接口，用于在进程状态发生变化的时刻，实时的将进程处于某一状态的总时间长度（单位为ticks）进行统计。

```
// function interface
int on_state_change(int cur_state, int nxt_state, struct proc *p);
```

1. **输入**包括：参数cur_state，表示当前的进程处于的状态，参数nxt_state，表示当前进程即将进入的下一状态，进程p的指针，表示跟踪的进程是哪个。
2. **输出**包括：没有输出，只是用来帮助实现wait_sched统计的一个辅助函数。
3. **功能**：负责在xv6进程状态发生变化的时刻，统计进程处于该状态的总时长，更新进程处于该状态的时间长度到进程的信息中，供wait_sched的系统调用使用，用户从而可以最终通过使用wait_sched得知进程实际的耗时情况。

xv6的进程状态总共有以下几种：**UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE**

然后这些状态之间发生变化的点，在下面的图例中进行了大致的概括：



提示：

提示1

关于`on_state_change`需要在哪些进程状态”可能发生变化“的位置添加，已经在proc.c代码中为大家添加了标记`TODO`，下面是一个总结：

需要添加`on_state_change`的位置：

- 修改allocproc
- 修改userinit
- 修改fork
- 修改exit
- 修改scheduler
- 修改 yield
- 修改 sleep
- 修改 wakeup
- 修改 kill

记得也要看一下上面的状态切换图，可以很直观的理解xv6进程状态切换的具体逻辑，实现起来更加清楚和简单。

提示2

wait_sched的大部分内容和原来的xv6 wait的系统调用类似，可以在wait的基础上复制过来，修改我们新增的关于进程状态时间的字段逻辑即可。

具体添加wait_sched系统调用的流程：

kernel侧

- defs.h 指明你的系统调用具体实现的函数
- syscall.c 增加系统调用的接口名称
- syscall.h 添加系统调用对应的数码编号
- sysproc.c 按照在syscall.c中声明的系统调用格式，具体实现系统调用，内部则通过defs.h的声明的底层实现

user侧：

- usys.pl：添加entry项
- user.h 添加给用户空间程序使用的系统调用的函数签名

Makefile

- 添加调用wait_sched系统调用的用户程序 stat

运行结果

实验提供了一个stat（见 `user/stat.c`）用户级应用程序，stat程序接受一个int的输入参数n，表示统计子进程的数量。

该程序提供一个需要很大计算时间长度的任务big_calculate_task，用于让fork出的子进程运行，从而让父进程通过上面实现的wait_sched系统调用在子进程运行结束被回收的时刻收集到子进程的运行状态信息，从而反映xv6默认的调度方式（提问：是怎样的调度策略？）对于进程运行的影响。

注意：大家**不要修改** `user/stat.c` 应用程序，它只是用于测试新增的wait_sched系统调用。

在你添加完wait_sched系统调用之后，运行用户程序stat，输入 `stat 5`，运行正确的情况下，你可以看到以下输出：

```
$ stat 5
PID: 4 | Runnable Time: 36 ticks | Running Time: 49 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 39 ticks | Running Time: 61 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 37 ticks | Running Time: 63 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 44 ticks | Running Time: 60 ticks | Sleep Time: 12 ticks
```

```
PID: 8 | Runnable Time: 44 ticks | Running Time: 60 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 110 ticks
```

我们先不着急动手，先看看结果长什么样。
简单的解释一下：stat n在接受了你指定的副本数量n之后，会统计相应数量的进程的进程号，进程处于可运行状态的等待运行时间，真正运行的时间，休眠状态时长，以及最终计算的平均轮转时间（对上述三者的时长求和并取平均），注意关于时间的单位都是xv6时钟中断的间隔ticks).

注意：在成功运行之后，将实验结果截图，并把数据整理成表格，将**实际运行的截图**和**数据表格**一同放在实验报告的**任务一部分**（作为任务一的完成成果）

表格格式大概长这样：

PID	Runnable Time (ticks)	Running Time (ticks)	Sleep Time (ticks)
A	***		
B			
C			
D			
E			

平均轮转时间: *** ticks

任务二：优先级调度（priority scheduling）

实验目标

在该任务中，你需要扩展xv6的进程调度实现，**修改xv6的进程调度逻辑（主要修改kernel/proc.c的scheduler()函数）**，使得xv6支持指定进程优先级的**静态优先级调度模式**PR。

同时，为了让用户可以为进程指定优先级以使用我们提供的优先级调度策略，我们还需要实现系统调用功能 `set_priority`，让用户可以通过**设置进程优先级**改变进程的运行模式。

具体要求：修改scheduler的实现，扩展优先级调度部分

```
scheduler的代码如下面的结构组织:
#if defined RR

// Round-Robin-Schedule Here
```

```
#elif defined PR

// PRiority-Schedule Logics Here

#endif
```

- if - elif - endif 通过宏定义的方式选择具体启用哪一种调度策略
- 控制逻辑在 proc.h中通过添加 `#define RR / #define PR` 即可选定相应的调度策略进行测试.
- 优先级调度的逻辑：我们为每个进程指定一个优先级（默认优先级情况下的优先级=2），寻找RUNNABLE状态下优先级最高的进程（优先级范围为0-3的整数，0为最高优先级，3为最低优先级），找到这个优先级最高的进程（记作max_p），参考xv6默认调度的实现将max_p的进程相关上下文进行设置并且切换过去，完成进程调度。

在`proc.h`，我们为进程添加了表示优先级的成员变量priority，以及切换不同调度模式的MACRO定义（不需要额外修改）

```
// Per-process state
struct proc {
  ...
  // Newly added for Lab3
  ...

  int priority;                // Process priority, within [0,1,2,3], 0 is the highest and 3 is
the lowest
};

// TODO: using RR for Round-Robin, using PR for Priority-Scheduling
// #define RR
#define PR
```

具体要求：实现set_priority系统调用

`set_priority`的接口也在proc.c中定义好了，如下所示：

```
// set priority [0-3] to a given process [pid]
// -1 means error, 0 means success
int set_priority(int priority, int pid)
```

我们设置优先级范围为0-3的整数，0为最高优先级，3为最低优先级

set_priority的具体实现：

1. **输入**包括：priority需要指定的优先级大小，以及进程号pid。
2. **输出**包括：返回一个状态码， -1 表示出错，成功则返回0。
3. **功能**：如果指定的优先级大小符合范围和类型的要求，则将指定的priority数值赋值给进程号=pid的目标进程，返回赋值结果成功则返回0，否则失败返回-1。

提示：

1. 我们实现的优先级是最简单的 **静态优先级调度**：静态优先级调度指的是进程最初有一个优先级，运行过程中可以通过系统调用改变这个优先级，但是不会随着等待时间或运行时间的增加而自动改变。

2. 调度器修改为优先级调度的过程，逻辑上说其实就是遍历可以被调度的进程，寻找里面最高优先级的进程（只需要考虑优先级），找到它调度它，不断重复这个过程，但是需要小心获取进程状态的时候，它们的锁状态和数据状态不能出现**冲突/死锁**的问题。

3. 关于添加系统调用的流程，可以大量的参考任务一部分的修改（defs.h修改很多，proc修改很多，syscall修改很多），最终测试之前不要忘记 在用户侧的usys.pl中添加entry项 `entry("set_priority");` ，以及user.h中加上你的系统调用 `int set_priority(int, int);`

运行结果

实验提供了一个 `priostat` 用户级应用程序（见 `user/priostat.c` ），类似stat用户程序，priostat也接受一个参数n表示观测的进程数量。

只不过这里通过增加的set_priority系统调用为进程设置了不同的优先级，然后启用了上面实现的优先级调度模式，让xv6的调度运行模式变成优先级调度，最后也是让父进程通过上面实现的wait_sched系统调用在子进程运行结束被回收的时刻收集到子进程的运行状态信息。

完成任务后，你需要首先在**proc.h中启用PR的宏定义**，然后make qemu，在xv6中运行priostat程序，输入 `priostat 5` ，通过测试会显示如下内容：

```
$ priostat 5
Set priority 0 to PID 4
Set priority 1 to PID 5
Set priority 2 to PID 6
Set priority 3 to PID 7
Set priority 0 to PID 8
PID: 4 | Runnable Time: 9 ticks | Running Time: 48 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 7 ticks | Running Time: 52 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 25 ticks | Running Time: 57 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 60 ticks | Running Time: 59 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 95 ticks | Running Time: 51 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 104 ticks
```

我们先不着急动手，先看看结果长什么样。
简单的解释一下： `priostat n` 在接受了你指定的副本数量n之后，会统计相应数量的进程的进程号，进程的优先级，进程处于可运行状态的等待运行时间，真正运行的时间，休眠状态时长，以及最终计算的平均轮转时间（对上述三者的时长求和并取平均），注意关于时间的单位都是xv6时钟中断的间隔ticks).

注意：在成功运行之后，将实验结果截图，并把数据整理成表格，将**实际运行的截图**和**数据表格**一同

放在实验报告的**任务二部分**（作为任务二的完成成果）

表格的格式：（注意0为最高优先级，3为最低优先级）

PID	Priority	Runnable Time (ticks)	Running Time (ticks)	Sleep Time (ticks)
A	0	***		
B	3			
C	1			
D	2			
E	1			

平均轮转时间: ***ticks

任务三： 不同调度模式的对比

这一部分请写在实验报告内

1. 对比xv6中，通过Round-Robin和优先级调度对于进程运行的影响（结合上面的运行情况，**对比不同状态的实际，平均轮转时间等等方面...**）
2. 简单描述一下Round-Robin和 优先级调度的特点，分别适应什么场景?
3. 任务一和任务二的代码实现思路（可以只写思路和关键代码） 例如：`proc.c`的某个功能`wait_sched`，逻辑上在做什么？调用了哪些？实现上哪里出了bug?

关于Lab3的打分测试

当完成上述的两个写代码的任务后，请在xv6目录下，**新建time.txt文件**，在该文件中写入你做完这个实验所花费的时间（估算一下就行，单位是小时），只需要写一个整数即可。
本学期后面的lab会根据目前lab完成时间和难度进行（可能的）相应的修正 ***（出这个lab3大概我花了10个小时）***
注意：**本lab没有make grade打分测试的这一步**，但是需要在报告的任务三部分详细**写明自己完成lab的过程和实现的思路**。

四、实验结果提交

将实验报告和实验代码打包为压缩文件提交到eLearning平台。

4.1 实验报告

参照课程文件 (https://elearning.fudan.edu.cn/courses/78523/files/folder/%E5%AE%9E%E9%AA%8C%E7%9B%B8%E5%85%B3%E6%96%87%E6%A1%A3) 中的《OS实验报告模板》 (https://elearning.fudan.edu.cn/files/4915949/)，按如下要求书写实验报告，力争规范、简洁。

1. 对于每个实验，详细描述实验过程，对于你认为的关键步骤附上必要的截图。
2. 有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。
3. 鼓励实验报告中包括但不局限于以下内容：实验过程中碰到了什么问题？如何解决这些问题？实验后还存在哪些疑问或者有什么感想？
4. 如果实验附有练习，请在每个练习之后作答，这是实验报告评分的重要部分。

4.2 实验代码

不需要提交完整的代码包，**只需要提交**`commit.patch`**文件即可**，操作步骤如下：

- 在完成实验之后，将当前分支上的所有更改进行提交（`commit`，具体方法参考git使用教程 (https://os-labs.pages.dev/lab1/part4/#3-git)）。
- 在仓库的目录下使用`make diff`命令导出更改文件`commit.patch`。

4.3 提交平台

请将**生成的**`commit.patch`**文件**与**实验报告**一起打包提交到eLearning平台Lab3
