操作系统课后作业

（按章节、对中文版有勘误、补全）

肖德贵

2024 年 2 月修订

目录

说明

文档列出的作业是教材《操作系统导论》(中文版)、“Operating Systems (Three Easy Pieces)”（英文版）的作业，建议阅读英文版。

列出了每一章(如果有的话)的作业题目,根据教学内容,**带*标记的可选做**。

教材中文版有些题目有误，有些章节没有作业，所以，对中文版有误的题目进行了勘误处理,对照英文章节补充了(红色标注的部分)中文版中没有的作业，补充的可能主要是英文描述的。

完成作业涉及到教材配套提供的文档在四个压缩文件中：HW_CPU_Virtualization.zip， HW_MEM_Virtualization.zip， HW_ConCurrency.zip 和 HW_Persistence.zip，假设把这四个文件放到HW目录下并解压缩，则目录组织结构：

```
.
├ HW
| ├ HW_CPU_Virtualization
| | ├ HW-CPU-Intro
| | | ├ process-run.py
| | | └ README-process-run
| | ├ HW-Lottery
| | | ├ lottery.py
| | | └ README-lottery
| | ├ HW-MLFQ
| | | ├ mlfq.py
| | | └ README-mlfq
| | ├ HW-Sched-MultiCPU
| | | ├ multi.py
| | | └ README-multi
| | ├ HW-Scheduler
| | | ├ README-scheduler
| | | └ scheduler.py
| ├ HW_MEM_Virtualization
| | ├ HW-Freespace
| | | ├ malloc.py
| | | └ README-malloc
| | ├ HW-Paging-BeyondPhys-Real
| | | ├ Makefile
| | | ├ mem.c
| | | └ README
| | ├ HW-Paging-LinearTranslate
| | | ├ paging-linear-translate.py
```

```
| | | └ README-paging-linear-translate
| | ├ HW-Paging-MultiLevelTranslate
| | | ├ paging-multilevel-translate.py
| | | └ README-paging-multilevel-translate
| | ├ HW-Paging-Policy
| | | ├ paging-policy.py
| | | └ README-paging-policy
| | ├ HW-Relocation
| | | ├ README-relocation
| | | └ relocation.py
| | ├ HW-Segmentation
| | | ├ README-segmentation
| | | └ segmentation.py
| ├ HW_ConCurrency
| | ├ HW-ThreadsIntro
| | | ├ loop.s
| | | ├ looping-race-nolock.s
| | | ├ README-race
| | | ├ simple-race.s
| | | ├ wait-for-me.s
| | | └ x86.py
| | ├ HW-ThreadsLocks
| | | ├ flag.s
| | | ├ peterson.s
| | | ├ README-locks
| | | ├ test-and-set.s
| | | ├ test-and-test-and-set.s
| | | ├ ticket.s
| | | ├ x86.py
| | | └ yield.s
| | ├ HW-Threads-RealAPI
| | | ├ main-deadlock.c
| | | ├ main-deadlock-global.c
| | | ├ main-race.c
| | | ├ main-signal.c
| | | ├ main-signal-cv.c
| | | ├ Makefile
| | | ├ mythreads.h
| | | └ README
| | ├ HW- Threads-RealCV
| | | ├ main-common.c
| | | ├ main-header.h
| | | ├ main-one-cv-while.c
| | | ├ main-two-cvs-if.c
```

```
│ │ │ ├ main-two-cvs-while.c
│ │ │ ├ main-two-cvs-while-extra-unlock.c
│ │ │ ├ Makefile
│ │ │ ├ mythreads.h
│ │ │ ├ pc-header.h
│ │ │ └ README
│ │ ├ HW-RealDeadlock
│ │ │ ├ main-common.c
│ │ │ ├ main-header.h
│ │ │ ├ Makefile
│ │ │ ├ mythreads.h
│ │ │ ├ README
│ │ │ ├ vector-avoid-hold-and-wait.c
│ │ │ ├ vector-deadlock.c
│ │ │ ├ vector-global-order.c
│ │ │ ├ vector-header.h
│ │ │ ├ vector-nolock.c
│ │ │ └ vector-try-wait.c
│ │ ├ HW-RealSemaphores
│ │ │ ├ barrier.c
│ │ │ ├ fork-join.c
│ │ │ ├ mutex-nostarve.c
│ │ │ ├ reader-writer.c
│ │ │ ├ reader-writer-nostarve.c
│ │ │ ├ README
│ │ │ └ rendezvous.c
└ HW_Persistence
  │ ├ HW-AFS
  │ │ ├ afs.py
  │ │ └ README-afs
  │ ├ HW-Checksums
  │ │ ├ checksum.py
  │ │ └ README
  │ ├ HW-Disk
  │ │ ├ disk.py
  │ │ ├ disk-precise.py
  │ │ └ README-disk
  │ ├ HW-FFS
  │ │ ├ ffs.py
  │ │ ├ in.empty
  │ │ ├ in.example1
  │ │ ├ in.example2
  │ │ ├ in.fragmented
  │ │ ├ in.largefile
```

```
│ │ ├ in.manyfiles
│ │ ├ in.medfile
│ │ └ README
│ ├ HW-Journaling
│ │ ├ fsck.py
│ │ └ README.fsck
│ ├ HW-LFS
│ │ ├ lfs.py
│ │ └ README
│ ├ HW-Raid
│ │ ├ raid.py
│ │ └ README-raid
│ ├ HW-SSD
│ │ ├ README
│ │ └ ssd.py
└ HW-VSFS
    ├ README.vsfs
    └ vsfs.py
```

使用这些文件时，对于 Python 源代码文件（.py），不同的 Python 版本可能会要对源代码文件进行修改，主要是 printf 函数，Python3.x 的版本 printf 需要加括号()。遇到问题建议多百度。

运行程序时，Linux 环境下，在命令提示符下：./xxxx；Windows 环境下，在命令提示符下：.\xxxx。

Python 程序的运行，依据你的 Python 版本及安装环境，可能以./xxx.py 或者 python ./xxx.py 的方式。如果要运行 C 语言程序、汇编语言程序，需要经过编译、链接、生成可执行程序。

第 4 章 抽象：进程

（完成这些作业题目前，阅读 HW\HW_CPU_Virtualization\HW-CPU-Intro 目录下的文件 process-run.py 和 README-process-run。都可以用文本编辑器打开。）

1．用以下标志运行程序：./process-run.py -l 5:100,5:100。CPU 利用率（CPU 使用时间的百分比）应该是多少？说明理由。然后利用 -c 和-p 标志查看你的答案是否正确。

2．现在用这些标志运行：./process-run.py -l 4:100,1:0。这些标志指定了两个进程：一个进程包含 4 条指令（都使用 CPU），另一个进程只是简单地发出 I/O 并等待它完成。完成这两个进程需要多长时间？利用-c 和-p 检查你的答案是否正确。

3．现在交换进程的顺序运行：./process-run.py -l 1:0,4:100。现在发生了什么？交换顺序是否重要？为什么？同样，用-c 和-p 看看你的答案是否正确。

4．现在探索另一些标志。一个重要的标志是-S，它决定了当进程发出 I/O 时系统如何反应。将标志设置为 SWITCH_ON_END，在进程进行 I/O 操作时，系统将不会切换到另一个进程，而是等待进程完成。当你运行以下两个进程时，会发生什么情况？一个执行 I/O，另一个执行 CPU 工作。（-l 1:0,4:100 -c -S SWITCH_ON_END）

5．现在，运行上述相同的进程，只是把切换行为设置为当一个进程在等待 I/O 时切换到另一个进程（-l 1:0,4:100 -c -S SWITCH_ON_IO）。现在会发生什么？利用-c 和-p 来确认你的答案是否正确。

6．另一个重要的行为是 I/O 完成时要做什么。利用-I IO_RUN_LATER，当 I/O 完成时，发出它的进程不一定马上运行。相反，当时运行的进程一直运行。试试运行./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p，会发生什么？系统资源是否被有效利用？

7．现在运行相同的进程，但使用-I IO_RUN_IMMEDIATE 设置，该设置立即运行发出 I/O 的进程。这种行为有何不同？为什么再次运行一个刚刚完成了 I/O 的进程会是一个好主意？

8．现在运行一些随机生成的进程，例如-s 1 -l 3:50,3:50, -s 2 -l

3:50,3:50, -s 3 -I 3:50,3:50。看看你是否能预测追踪记录会如何变化？当你使用-I IO_RUN_IMMEDIATE 与-I IO_RUN_LATER 时会发生什么？当你使用-S SWITCH_ON_IO 与-S SWITCH_ON_END 时会发生什么？

第 5 章 插叙：进程 API

1．编写一个调用 fork()的程序。在调用 fork()之前，让主进程访问一个变量（例如 x）并将其值设置为某个值（例如 100）。子进程中的变量有什么值？当子进程和父进程都改变 x 的值时，变量会发生什么？

2．编写一个打开文件的程序（使用 open()系统调用），然后调用 fork()创建一个新进程。子进程和父进程都可以访问 open()返回的文件描述符吗？当它们并发（即同时）写入文件时，会发生什么？

3．编写另一个程序使用 fork()。子进程应打印“hello”，父进程应打印“goodbye”。你应该尝试确保子进程始终先打印。你能否不在父进程中调用 wait()而做到这一点呢？

4．编写一个调用 fork()的程序，然后调用某种形式的 exec()来运行程序/bin/ls。看看是否可以尝试 exec()的所有变体，包括 execl()、execle()、execlp()、execv()、execvp()和 execvP()。为什么同样的基本调用会有这么多变种？

5．现在编写一个程序，在父进程中使用 wait()，等待子进程完成。wait()返回值是什么？如果你在子进程中使用 wait()会发生什么？

6．对前一个程序稍作修改，这次使用 waitpid()而不是 wait()。什么时候 waitpid()会有用？

7．编写一个创建子进程的程序，然后在子进程中关闭标准输出（STDOUT_FILENO）。如果子进程在关闭描述符后调用 printf()打印输出，会发生什么？

8．编写一个程序创建两个子进程，并使用 pipe()系统调用，将一个子进程的标准输出连接到另一个子进程的标准输入。

第 6 章 机制：受限直接执行

*In this homework, you'll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is gettimeofday(); read the man page for details. What you'll see there is that gettimeofday() returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to gettimeofday() to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result. If gettimeofday() is not precise enough for you, you might look into using the rdtsc instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. The lmbench benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of

communicating like this repeatedly, lmbench can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU; what you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the sched setaffinity() call is what you're looking for. By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

第 7 章 进程调度：介绍

（完成这些作业题目前，阅读 HW\HW_CPU_Virtualization\HW-Scheduler 目录下的文件 scheduler.py 和 README-scheduler。都可以用文本编辑器打开）

1．分别使用 SJF 和 FIFO 调度程序运行长度都为 200 的 3 个作业时，计算响应时间和周转时间。

2．现在做同样的事情，但有不同长度的作业，即 100、200 和 300。

3．现在做同样的事情，但采用 RR 调度程序，时间片为 1。

4．对于什么类型的工作负载，SJF 提供与 FIFO 相同的周转时间？

5．对于什么类型的工作负载和时间片长度，SJF 与 RR 提供相同的响应时间？

6．随着作业长度的增加，SJF 的响应时间会怎样？你能使用模拟程序来展示趋势吗？

7．对于 RR，随着时间片长度的增加，其响应时间会怎样？给定 N 个作业，你能写出一个计算最坏情况下的响应时间方程吗？

第 8 章 调度：多级反馈队列

（完成这些作业题目前，阅读 HW\HW_CPU_Virtualization\HW-MLFQ 目录下的文件 mlfq.py 和 README-mlfq。都可以用文本编辑器打开）

1．只用两个作业和两个队列运行几个随机生成的问题。针对每个作业计算 MLFQ 的执行记录。为了简化计算，限制每项作业的长度并关闭 I/O。

2．如何运行调度程序来重现本章中的每个实例？

3．将如何配置调度程序参数，使得调度程序像轮转调度程序那样工作？

4.精心设计有两个作业的负载和调度程序参数，以便一个作业能利用较早的规则 4a 和 4b（用-S 标志打开）来“愚弄”调度程序，在特定的时间间隔内获得 99%的 CPU。

5．给定一个系统，其最高队列中的时间片长度为 10ms，你需要如何频繁地将工作推回到最高优先级级别（带有-B 标志），以保证一个长时间运行（并可能饥饿）的作业得到至少 5%的 CPU？

6．将刚完成 I/O 的作业添加到队列的哪一端（首或尾）时，会导致调度出现一个问题。-I 标志改变了这个调度模拟器的这方面行为。尝试一些工作负载，看看你是否能看到这个标志的效果。

第 9 章 调度：比例份额

（完成这些作业题目前，阅读 HW\HW_CPU_Virtualization\HW-Lottery 目录下的文件 lottery.py 和 README-lottery。都可以用文本编辑器打开）

1．计算 3 个作业在随机种子为 1、2 和 3 时的模拟解。

2．现在运行两个特定的作业：每个长度为 10，但是一个（作业 0）只有一张彩票，另一个（作业 1）有 100 张彩票（-l 10：1,10：100）。彩票数量如此不平衡时会发生什么？在作业 1 完成之前，作业 0 是否会运行？运行频率（how often）？一般来说，这种彩票不平衡对彩票调度的行为有什么影响？

3．如果运行两个长度为 100、都有 100 张彩票的作业（-l100：100,100：100),调度程序有多不公平？运行一些不同的随机种子来确定(概率上的)答案，以一项作业比另一项作业早完成多少来确定不公平性。

4．随着时间片规模（-q）变大，你对上一个问题的答案如何改变？

5．你可以制作类似本章中的图表吗？还有什么值得探讨的？用步长调度程序，图表看起来如何？

第 10 章 多处理器调度（进阶）

（完成这些作业题目前，阅读 HW\HW_CPU_Virtualization\ HW-Sched-MultiCPU 目录下的文件 multi.py 和 README。都可以用文本编辑器打开）

1. To start things off, let's learn how to use the simulator to study how to build an effective multi-processor scheduler. The first simulation will run just one job, which has a run-time of 30, and a working-set size of 200. Run this job (called job 'a' here) on one simulated CPU as follows: ./multi.py -n 1 -L a:30:200. How long will it take to complete? Turn on the -c flag to see a final answer, and the -t flag to see a tick-by-tick trace of the job and how it is scheduled.

2. Now increase the cache size so as to make the job's working set (size=200) fit into the cache (which, by default, is size=100); for example, run ./multi.py -n 1 -L a:30:200 -M 300. Can you predict how fast the job will run once it fits in cache? (hint: remember the key parameter of the warm rate, which is set by the -r flag) Check your answer by running with the solve flag (-c) enabled.

3. One cool thing about multi.py is that you can see more detail about what is going on with different tracing flags. Run the same simulation as above, but this time with time left tracing enabled (-T). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?

4. Now add one more bit of tracing, to show the status of each CPU cache for each job, with the -C flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a 'w' (if the cache is warm for that job). At what point does the cache become warm for job 'a' in this simple example? What happens as you change the warmup time parameter (-w) to lower or higher values than

the default?

5. At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn't this a multi-processor CPU scheduling chapter? Oh yeah! So let's start working with multiple jobs. Specifically, let's run the following three jobs on a two-CPU system (i.e., type ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50) Can you predict how long this will take, given a round-robin centralized scheduler? Use -c to see if you were right, and then dive down into details with -t to see a step-by-step and then -C to see whether caches got warmed effectively for these jobs. What do you notice?

6. Now we'll apply some explicit controls to study cache affinity, as described in the chapter. To do this, you'll need the -A flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. In this case, let's use it to place jobs 'b' and 'c' on CPU 1, while restricting 'a' to CPU 0. This magic is accomplished by typing this ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1 ; don't forget to turn on various tracing options to see what is really happening! Can you predict how fast this version will run? Why does it do better? Will other combinations of 'a', 'b', and 'c' onto the two processors run faster or slower?

7. One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, a situation entitled super-linear speedup. To experiment with this, use the job description here (-L a:100:100,b:100:100,c:100:100) with a small cache (-M 50) to create three jobs. Run this on systems with 1, 2, and 3 CPUs (-n 1, -n 2, -n

3). Now, do the same, but with a larger per-CPU cache of size 100. What do you notice about performance as the number of CPUs scales? Use -c to confirm your guesses, and other tracing flags to dive even deeper.

8. One other aspect of the simulator worth studying is the per-CPU scheduling option, the -p flag. Run with two CPUs again, and this three job configuration (-L a:100:100,b:100:50,c:100:50). How does this option do, as opposed to the hand-controlled affinity limits you put in place above? How does performance change as you alter the 'peek interval' (-P) to lower or higher values? How does this per-CPU approach work as the number of CPUs scales?

9. Finally, feel free to just generate random workloads and see if you can predict their performance on different numbers of processors, cache sizes, and scheduling options. If you do this, you'll soon be a multi-processor scheduling master, which is a pretty awesome thing to be. Good luck!

第 13 章 抽象：地址空间

（写代码，使用工具 free、pmap）

1. The first Linux tool you should check out is the very simple tool free. First, type man free and read its entire manual page; it's short, don't worry!

2. Now, run free, perhaps using some of the arguments that might be useful (e.g., -m, to display memory totals in megabytes). How much memory is in your system? How much is free? Do these numbers match your intuition?

3. Next, create a little program that uses a certain amount of memory, called memory-user.c. This program should take one command-line argument: the number of megabytes of memory it will use. When run, it should allocate an array, and constantly stream through the array, touching each entry. The program should do this indefinitely, or, perhaps, for a certain amount of time also specified at the command line.

4. Now, while running your memory-user program, also (in a different terminal window, but on the same machine) run the free tool. How do the memory usage totals change when your program is running? How about when you kill the memory-user program? Do the numbers match your expectations? Try this for different amounts of memory usage. What happens when you use really large amounts of memory?

5. Let's try one more tool, known as pmap. Spend some time, and read the pmap manual page in detail.

6. To use pmap, you have to know the process ID of the process you're interested in. Thus, first run ps auxw to see a list of all processes; then, pick an interesting one, such as a browser. You can also use your memory-user program in this case (indeed, you can even have that program call getpid() and print out its PID for your

convenience).

7. Now run pmap on some of these processes, using various flags (like -X) to reveal many details about the process. What do you see? How many different entities make up a modern address space, as opposed to our simple conception of code/stack/heap?

8. Finally, let's run pmap on your your memory-user program, with different amounts of used memory. What do you see here? Does the output from pmap match your expectations?

第 14 章 插叙：内存操作 API

（写代码，使用工具 gdb、valgrind）

1. 首先，编写一个名为 null.c 的简单程序，它创建一个指向整数的指针，将其设置为 NULL，然后尝试对其进行释放内存操作（dereference it）。把它编译成一个名为 null 的可执行文件。当你运行这个程序时会发生什么？

2. 接下来，编译该程序，其中包含符号信息（使用-g 标志）。这样做可以将更多信息放入可执行文件中，使调试器可以访问有关变量名称等的更多有用信息。通过输入 gdb null，在调试器下运行该程序，然后，一旦 gdb 运行，输入 run，gdb 显示什么信息？

3. 最后，对这个程序使用 valgrind 工具。我们将使用属于 valgrind 的 memcheck 工具来分析发生的情况。输入以下命令来运行程序：valgrind --leak-check=yes null。当你运行它时会发生什么？你能解释工具的输出吗？

4. 编写一个简单程序使用 malloc()来分配了内存，但在退出之前忘记释放它。这个程序运行时会发生什么？你可以用 gdb 来发现它的任何问题吗？用 valgrind 呢（再次使用--leak-check=yes 标志）？

5. 编写一个程序，使用 malloc 创建一个名为 data、大小为 100 的整数数组。然后，将 data[100]设置为 0。当你运行这个程序时会发生什么？当你使用 valgrind 运行这个程序时会发生什么？程序是否正确？

6. 创建一个程序分配整数数组（如上所述），释放它们，然后尝试打印数组中某个元素的值。程序会运行吗？当你使用 valgrind 运行时会发生什么？

7. 现在传递一个有趣的值来释放（例如，在上面分配的数组中间的一个指针）。 会发生什么？你是否需要工具来找到这种类型的问题？

8. 尝试一些内存分配的其他接口。例如，创建一个简单的向量似的数据结构，以及使用 realloc()来管理向量的相关函数。使用数组来存储向量元素，当用户在向量中添加条目时，请使用 realloc()为向量分配更多空间。这样的向量表现如何？它与链表相比如何？使用 valgrind 来帮助你发现错误。

9. 花更多时间阅读有关使用 gdb 和 valgrind 的信息。了解你的工具至关重要，花时间学习如何成为 UNIX 和 C 环境中的调试器专家。

第 15 章 机制：地址转换

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Relocation 目录下的文件README-relocation和relocation.py。都可以用文本编辑器打开）

1．用种子 1、2 和 3 运行，并计算进程生成的每个虚拟地址是处于界限内还是界限外?如果在界限内，请计算地址转换。

2．使用以下标志运行：-s 0 -n 10。为了确保所有生成的虚拟地址都处于界限内，要将-l（界限寄存器）设置为什么值？

3．使用以下标志运行：-s 1 -n 10 -l 100。可以设置基址的最大值是多少，以便地址空间仍然完全放在物理内存中？

4．运行和第 3 题相同的操作，但使用较大的地址空间（-a）和物理内存（-p）。

5．作为界限寄存器的值的函数，随机生成的虚拟地址的哪一部分是有效的？画一个图，使用不同随机种子运行，限制值从 0 到最大地址空间大小。

第 16 章 分段

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Segmentation 目录下的文件 README-segmentation 和 segmentation.py。都可以用文本编辑器打开）

1. 先让我们用一个小地址空间来转换一些地址。这里有一组简单的参数和几个不同的随机种子。你可以转换这些地址吗？

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2

2. 现在，让我们看看是否理解了这个构建的小地址空间（使用上面问题的参数）。段 0 中最高的合法虚拟地址是多少？段 1 中最低的合法虚拟地址是多少？在整个地址空间中，最低和最高的非法地址分别是多少？最后，如何运行带有-A 标志的 segmentation.py 来测试你的答案是否正确？

3. 假设我们在一个 128 字节的物理内存中有一个很小的 16 字节地址空间。你会设置什么样的基址和界限，以便让模拟器为指定的地址流生成以下转换结果：有效，有效，违规，……，违规，有效，有效？假设用以下参数：

segmentation.py -a 16 -p 128

-A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

--b0 ? --l0 ? --b1 ? --l1 ?

4. 假设我们想要生成一个问题，其中大约 90%的随机生成的虚拟地址是有效的（即不产生段异常）。你应该如何配置模拟器来做到这一点？哪些参数很重要？

5. 你可以运行模拟器，使所有虚拟地址无效吗？怎么做到？

第 17 章 空闲空间管理

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Freespace 目录下的文件 malloc.py 和 README-malloc。都可以用文本编辑器打开）

1．首先以标志 flag -n 10 -H 0 -p BEST -s 0 运行程序来产生一些随机的分配和空闲。你能预测 malloc()/free()会返回什么吗？你可以在每次请求后猜测空闲列表的状态吗？随着时间的推移，你对空闲列表有什么发现？

2．使用最差匹配策略搜索空闲列表（-p WORST）时，结果有何不同？什么改变了？

3．如果使用首次匹配（-p FIRST）会如何？使用首次匹配时，什么变快了？

4．对于上述问题，列表在保持有序时，可能会影响某些策略找到空闲位置所需的时间。使用不同的空闲列表排序(-l ADDRSORT,-l SIZESORT +,-l SIZESORT-)查看策略和列表排序如何相互影响。

5．合并空闲列表可能非常重要。增加随机分配的数量（比如说-n 1000）。随着时间的推移，大型分配请求会发生什么？在有和没有合并的情况下运行（即不用和采用-C 标志）。你看到了什么结果差异？每种情况下的空闲列表有多大？在这种情况下，列表的排序是否重要？

6．将已分配百分比-P 改为高于 50，会发生什么？它接近 100 时分配会怎样？接近 0 会怎样？

7．要生成高度碎片化的空闲空间，你可以提出怎样的具体请求？使用-A 标志创建碎片化的空闲列表，查看不同的策略和选项如何改变空闲列表的组织。

第 18 章 分页：介绍

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Paging-LinearTranslate 目录下的文件 paging-linear-translate.py 和 README-paging-linear-translate。都可以用文本编辑器打开）

1.在做地址转换之前，让我们用模拟器来研究在给定不同参数的情况下线性页表如何改变大小。在不同参数变化时，计算线性页表的大小。一些建议输入如下，通过使用-v 标志，你可以看到填充了多少个页表项。首先，理解线性页表大小如何随着地址空间的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

然后，理解线性页表大小如何随页大小的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

在运行这些命令之前，请试着想想预期的趋势。页表大小如何随地址空间的增长而改变？页表大小如何随着页大小的增长而改变？为什么一般来说，我们不应该使用很大的页呢？

2．现在让我们做一些地址转换。从一些小例子开始，使用-u 标志更改分配给地址空间的页数。例如：

```
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

如果增加每个地址空间中分配的页的百分比，会发生什么？

3.现在让我们尝试一些不同的随机种子，以及一些不同的（有时相当疯狂的）地址空间参数：

```
paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
```

paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2

paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3

哪些参数组合是不现实的？为什么？

4. 利用该程序尝试其他一些问题。你能找到让程序无法工作的限制吗？例如，如果地址空间大小大于物理内存，会发生什么情况？

第 19 章 分页：快速地址转换（TLB）

（完成这些作业题目前，阅读英文版第 19 章第 15 页 Homework 的说明）

1．为了计时，可能需要一个计时器，例如 gettimeofday()提供的。这种计时器的精度如何？一个操作需要花费多长时间，才能让你对它精确计时？（这有助于确定需要循环多少次，需要重复访问内存页才能对它成功计时。）

2．写一个程序，命名为 tlb.c，大体测算一下每个页的平均访问时间。程序的输入参数有：页的数目和尝试的次数。

3．用你喜欢的脚本语言（csh、Python 等）写一段脚本来运行这个程序，访问页面从 1 增长到几千(每次迭代以 2 倍增加)。在不同的机器上运行这段脚本，同时收集相应数据。需要试多少次才能获得可信的测量结果？

4．接下来，将结果绘图（类似于图 19.5）。可以使用 ploticus， zplot 之类的绘图工具。可视化使数据更容易理解，你认为是什么原因？

5．要注意编译器优化带来的影响。编译器做各种聪明的事情，包括优化掉循环，如果循环中增加的变量后续没有使用。如何确保编译器不优化掉你写的 TLB 大小测算程序的主循环？

6．还有一个需要注意的地方，今天的计算机系统大多配备有多个 CPU，每个 CPU 当然有自己的 TLB 结构。为了得到准确的测量数据，我们需要只在一个 CPU 上运行程序，避免调度器把进程从一个 CPU 调度到另一个去运行。如何做到？（提示：上网搜索“pinning a thread”相关的信息）如果没有这样做，代码从一个 CPU 移到了另一个，会发生什么情况？

7．另一个可能发生的问题与初始化有关。如果在访问数组 a 之前没有初始化，第一次访问将非常耗时，由于初始访问开销，比如要求置 0。这会影响你的代码及其计时吗？如何抵消这些潜在的开销？

第 20 章 分页：较小的表

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Paging-MultiLevelTranslate 目录下的文件 paging-multilevel-translate.py 和 README-paging-multilevel-translate。都可以用文本编辑器打开）

1．对于线性页表，你需要一个寄存器来定位页表，假设硬件在 TLB 未命中时进行查找。你需要多少个寄存器才能找到两级页表？三级页表呢？

2．给定随机种子 0、1 和 2，使用模拟器进行地址转换，并使用-c 标志检查你的答案。执行每次查找时需要多少次内存引用？

3.根据你对内存高速缓存的工作原理的理解，你认为对页表的内存引用在缓存中会如何表现？它们是否会导致大量的缓存命中（并导致快速访问）或者很多未命中（并导致访问缓慢）？

第 21 章 超越物理内存：机制

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Paging-BeyondPhys-Real 目录下的文件 Makefile、mem.c 和 README。都可以用文本编辑器打开。探索工具 vmstat 的使用。）

1. First, open two separate terminal connections to the same machine, so that you can easily run something in one window and the other. Now, in one window, run vmstat 1, which shows statistics about machine usage every second. Read the man page, the associated README, and any other information you need so that you can understand its output. Leave this window running vmstat for the rest of the exercises below. Now, we will run the program mem.c but with very little memory usage. This can be accomplished by typing ./mem 1 (which uses only 1 MB of memory). How do the CPU usage statistics change when running mem? Do the numbers in the user time column make sense? How does this change when running more than one instance of mem at once?

2. Let's now start looking at some of the memory statistics while running mem. We'll focus on two columns: swpd (the amount of virtual memory used) and free (the amount of idle memory). Run ./mem 1024 (which allocates 1024 MB) and watch how these values change. Then kill the running program (by typing control-c) and watch again how the values change. What do you notice about the values? In particular, how does the free column change when the program exits? Does the amount of free memory increase by the expected amount when mem exits?

3. We'll next look at the swap columns (si and so), which indicate how much swapping is taking place to and from the disk. Of course, to activate these, you'll need to run mem with large amounts of memory. First, examine how much free memory is on your Linux system (for example, by typing cat /proc/meminfo; type man proc for details on the /proc file system and the types of information you can find there). One of

the first entries in /proc/meminfo is the total amount of memory in your system. Let's assume it's something like 8 GB of memory; if so, start by running mem 4000 (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with 5000, 6000, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total) are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)

4. Do the same experiments as above, but now watch the other statistics (such as CPU utilization, and block I/O statistics). How do they change when mem is running?

5. Now let's examine performance. Pick an input for mem that comfortably fits in memory (say 4000 if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say 12000 again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Can you make a graph, with the size of memory used by mem on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn't?

6. Swap space isn't infinite. You can use the tool swapon with the -s flag to see how much swap space is available. What happens if you try to run mem with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?

7. Finally, if you're advanced, you can configure your system to use different swap devices using swapon and swapoff. Read the man pages

for details. If you have access to different hardware, see how the performance of swapping changes when swapping to a classic hard drive, a flash-based SSD, and even a RAID array. How much can swapping performance be improved via newer devices? How close can you get to in-memory performance?

第 22 章 超越物理内存：策略

（完成这些作业题目前，阅读 HW\HW_MEM_Virtualization\HW-Paging-Policy 目录下的文件 paging-policy.py 和 README-paging-policy。都可以用文本编辑器打开）

1．使用以下参数生成随机地址：-s 0 -n 10，-s 1 -n 10 和-s 2 -n 10。策略依此为 FIFO、LRU、OPT。计算所述地址追踪中的每次访问是否命中或未命中。

2．对于大小为 5 的高速缓存，为以下每个策略生成最差情况的地址引用序列：FIFO、LRU 和 MRU（最差情况下的引用序列导致尽可能多的未命中）。对于最差情况引用序列，需要缓存增大多少，才能大幅提高性能，并接近 OPT？

3．生成一个随机追踪序列（使用 Python 或 Perl）。你预计不同的策略在这样的追踪序列上的表现如何？

4．现在生成一些局部性追踪序列。如何能够产生这样的追踪序列？LRU 表现如何？LRU 比 RAND 好多少？CLOCK 表现如何？使用不同数量时钟位的 CLOCK 表现如何？

5．使用像 valgrind 这样的程序来测试真实应用程序并生成虚拟页面引用序列。例如，运行 valgrind --tool = lackey --trace-mem = yes ls 将为程序 ls 所做的每个指令和数据引用，输出近乎完整的引用追踪。为了使上述仿真器有用，你必须首先将每个虚拟内存引用转换为虚拟页号引用（通过屏蔽偏移量并向右移位来完成）。为了满足大部分请求，你的应用程序追踪需要多大的缓存？绘制工作集随缓存大小增加的图行。

第 26 章 并发：介绍

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-ThreadsIntro 目录下的相关文件。都可以用文本编辑器打开。）

1．首先，阅读并理解程序 loop.s。然后，运行：

./x86.py -p loop.s -t 1 -i 100 -R dx

这些参数指定了单个线程，每 100 条指令产生一个中断，以及追踪寄存器%dx。你能弄清楚%dx 在运行过程中的值吗？你有答案之后，再次运行上面的代码并使用-c 标志来检查你的答案。注意答案的左边显示了右侧指令运行后寄存器的值（或内存的值）。

2．运行相同的代码，但使用这些标志：

./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx

这里指定了两个线程，并将每个%dx 寄存器初始化为 3。那么，%dx 的值是什么？使用-c 标志运行以查看答案。多个线程的存在是否会影响计算？这段代码有竞态条件吗？

3．运行以下命令：

./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx

这使得中断间隔非常小且随机。使用不同的种子（-s） 来查看不同的交替。中断频率是否会改变这个程序的行为？

4．接下来我们将研究一个不同的程序（looping-race-nolock.s）。该程序访问位于内存地址 2000 的共享变量（名为 value）。运行：

./x86.py -p looping-race-nolock.s -t 1 -M 2000

在整个运行过程中，value（即内存地址为 2000）的值是多少？使用-c 运行来检查你的答案。

5．以多个迭代和多线程运行：

./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000

你明白为什么每个线程中的代码循环 3 次吗？value 的最终值是多少？

6．以随机（改变随机种子，如，-s 1，-s 2 等）中断间隔运行：

./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0

。只看线程交替，你能说出 x 的最终值是什么吗？中断的时机是否重要？

在哪里发生是安全的？中断在哪里会引起麻烦？换句话说，临界区究竟在哪里？

7．使用固定的中断间隔运行：

./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1

共享变量 value 的最终值是多少？改用不同的中断间隔（如，-i 2，-i 3 等）时 value 的值是多少？对于哪个中断间隔，程序会给出“正确的”最终答案？

8．以更多循环运行相同的代码（例如 set -a bx = 100）。使用-i 标志设置哪些中断间隔会导致“正确”结果？哪些间隔会导致令人惊讶的结果？

9．运行：

./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000

这里将线程 0 的%ax 寄存器设置为 1，并将线程 1 的值设置为 0，在整个运行过程中观察%ax 和内存位置 2000 的值。代码的行为应该如何？线程是如何使用位于内存位置 2000 的值的？内存位置 2000 的最终值是多少？

10．运行：

./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000

线程行为如何？线程 0 在做什么？改变中断间隔（例如，-i 1000，或者可能使用随机间隔）会如何改变追踪结果？程序是否高效地使用了 CPU？

第 27 章 插叙：线程 API

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-Threads-RealAPI 目录下的相关文件。都可以用文本编辑器打开。使用 helgrind 工具发现程序中的问题）

1. First build main-race.c. Examine the code so you can see the (hopefully obvious) data race in the code. Now run helgrind (by typing valgrind --tool=helgrind main-race) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?

2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?

3. Now let's look at main-deadlock.c. Examine the code. This code has a problem known as deadlock (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?

4. Now run helgrind on this code. What does helgrind report?

5. Now run helgrind on main-deadlock-global.c. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like helgrind?

6. Let's next look at main-signal.c. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing,particularly if the child thread takes a long time to complete?)

7. Now run helgrind on this program. What does it report? Is the code correct?

8. Now look at a slightly modified version of the code, which is found in main-signal-cv.c. This version uses a condition variable to

do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

9. Once again run helgrind on main-signal-cv. Does it report any errors?

第 28 章 锁

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-ThreadsLocks 目录下的相关文件。都可以用文本编辑器打开。）

1．首先用标志-p flag.s 运行 x86.py。该代码通过一个内存标志“实现”锁。你能理解汇编代码试图做什么吗？

2．使用默认值运行时，flag.s 是否按预期工作？使用-M 和-R 标志跟踪变量和寄存器（并打开-c 查看它们的值）。你能预测代码运行最终的 flag 的值吗？

3．使用-a 标志更改寄存器%bx 的值（例如，如果只运行两个线程，就用-a bx = 2，bx =2）。代码是做什么的？对这段代码问上面的问题，答案如何？

4．对每个线程将 bx 设置为高值，然后使用-i 标志生成不同的中断频率。什么值导致产生不好的结果？什么值导致产生良好的结果？

5．现在让我们看看程序 test-and-set.s。首先，尝试理解代码，其中使用 xchg 指令构建简单锁原语。获取锁怎么写的？释放锁如何写的？

6．现在运行代码，再次更改中断间隔（-i）的值，并确保循环多次。代码是否总能按预期工作？有时会导致 CPU 使用率不高吗？如何量化呢？

7．使用-P 标志生成锁相关代码的特定测试。例如，执行一个测试计划，在第一个线程中获取锁，但随后尝试在第二个线程中获取锁。正确的事情发生了吗？你还应该测试什么？

8．现在让我们看看 peterson.s 中的代码，它实现了 Person 算法（在文中的补充栏中提到）。研究这些代码，看看你能否理解它。

9.现在用不同的-i 值运行代码。你看到了什么样的不同行为？确保线程 IDs 的设置如代码假定的那样（例如，使用 -a bx=0,bx=1）。

10．你能控制调度（带-P 标志）来“证明”代码有效吗？你应该展示哪些不同情况？考虑互斥和避免死锁。

11．现在研究 ticket.s 中 ticket 锁的代码。它是否与本章中的代码相符？然后以下列参数运行：-a bx=1000,bx=1000（使得每个线程循环通过临界区 1000 次）。观察发生了什么。所有线程是否都花费了大量时间自旋等待锁？

12. 加入更多的线程的话，代码的行为如何？

13.现在研究 yield.s，其中，一条 yield 指令使能一个线程放弃对 CPU 的

控制（实际情况中，这是一条 OS 的原语，为简单起见，我们假设有这样一条指令）。找到一个场景，在该场景中，test-and-set.s 浪费 CPU 时钟进行自旋（等待），而 yield.s 没有浪费。节省了多少指令？这些节省发生在什么场景下？

14. 最后研究 test-and-test-and-set.s。这个锁做了什么？与 test-and-set.s 相比，它引入了那种类型的节省？

第 29 章 基于锁的并发数据结构

（写代码）

1*. We’ll start by redoing the measurements within this chapter. Use the call gettimeofday() to measure time within your program. How accurate is this timer? What is the smallest interval it can measure? Gain confidence in its workings, as we will need it in all subsequent questions. You can also look into other timers, such as the cycle counter available on x86 via the rdtsc instruction.

2*. Now, build a simple concurrent counter and measure how long it takes to increment the counter many times as the number of threads increases. How many CPUs are available on the system you are using? Does this number impact your measurements at all?

3*. Next, build a version of the sloppy counter. Once again, measure its performance as the number of threads varies, as well as the threshold. Do the numbers match what you see in the chapter?

4*. Build a version of a linked list that uses hand-over-hand locking [MS04], as cited in the chapter. You should read the paper first to understand how it works, and then implement it. Measure its performance. When does a hand-over-hand list work better than a standard list as shown in the chapter?

5*. Pick your favorite interesting data structure, such as a B-tree or other slightly more interested structure. Implement it, and start with a simple locking strategy such as a single lock. Measure its performance as the number of concurrent threads increases.

6*. Finally, think of a more interesting locking strategy for this favorite data structure of yours. Implement it, and measure its performance. How does it compare to the straightforward locking approach?

第 30 章 条件变量

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-Threads-RealCV 目录下的相关文件。都可以用文本编辑器打开。）

1. Our first question focuses on main-two-cvs-while.c (the working solution). First, study the code. Do you think you have an understanding of what should happen when you run the program?

2. Run with one producer and one consumer, and have the producer produce a few values. Start with a buffer (size 1), and then increase it. How does the behavior of the code change with larger buffers? (or does it?) What would you predict num full to be with different buffer sizes (e.g., -m 10) and different numbers of produced items (e.g., -l 100), when you change the consumer sleep string from default (no sleep) to -C 0,0,0,0,0,0,1?

3. If possible, run the code on different systems (e.g., a Mac and Linux). Do you see different behavior across these systems?

4. Let's look at some timings. How long do you think the following execution, with one producer, three consumers, a single-entry shared buffer, and each consumer pausing at point c3 for a second, will take? ./main-two-cvs-while

   -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0:0,0,0,1,0,0,0:0,0,0,1,0,0,0

   -l 10 -v -t

5. Now change the size of the shared buffer to 3 (-m 3). Will this make any difference in the total time?

6. Now change the location of the sleep to c6 (this models a consumer taking something off the queue and then doing something with it), again using a single-entry buffer. What time do you predict in this case? ./main-two-cvs-while

   -p 1 -c 3 -m 1 -C 0,0,0,0,0,0,1:0,0,0,0,0,0,1:0,0,0,0,0,0,1

   -l 10 -v -t

7. Finally, change the buffer size to 3 again (-m 3). What time do you predict now?

8. Now let's look at main-one-cv-while.c. Can you configure a sleep string, assuming a single producer, one consumer, and a buffer of size 1, to cause a problem with this code?

9. Now change the number of consumers to two. Can you construct sleep strings for the producer and the consumers so as to cause a problem in the code?

10. Now examine main-two-cvs-if.c. Can you cause a problem to happen in this code? Again consider the case where there is only one consumer, and then the case where there is more than one.

11. Finally, examine main-two-cvs-while-extra-unlock.c. What problem arises when you release the lock before doing a put or a get? Can you reliably cause such a problem to happen, given the sleep strings? What bad thing can happen?

第 31 章 信号量

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-Threads-RealSemaphores 目录下的相关文件。都可以用文本编辑器打开。）

1. The first problem is just to implement and test a solution to the fork/join problem, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See fork-join.c for details. Add the call sleep(1) to the child to ensure it is working.

2. Let's now generalize this a bit by investigating the rendezvous problem. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see rendezvous.c for details.

3. Now go one step further by implementing a general solution to barrier synchronization. Assume there are two points in a sequential piece of code, called P1 and P2. Putting a barrier between P1 and P2 guarantees that all threads will execute P1 before any one thread executes P2. Your task: write the code to implement a barrier() function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See barrier.c for details.

4. Now let's solve the reader-writer problem, also as described in the text. In this first take, don't worry about starvation. See the code in reader-writer.c for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?

5. Let's look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See reader-writer-nostarve.c for details.

6. Use semaphores to build a no-starve mutex, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in mutex-nostarve.c for more information.

7. Liked these problems? See Downey's free text for more just like them. And don't forget, have fun! But, you always do when you write code, no?

第 32 章 常见并发问题

（完成这些作业题目前，阅读 HW\HW_ConCurrency\HW-Threads-RealDeadlock 目录下的相关文件。都可以用文本编辑器打开。）

1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in vector-deadlock.c, as well as in main-common.c and related files.

Now, run ./vector-deadlock -n 2 -l 1 -v, which instantiates two threads (-n 2), each of which does one vector add (-l 1), and does so in verbose mode (-v). Make sure you understand the output. How does the output change from run to run?

2. Now add the -d flag, and change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?

3. How does changing the number of threads (-n) change the outcome of the program? Are there any values of -n that ensure no deadlock occurs?

4. Now examine the code in vector-global-order.c. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this vector add() routine when the source and destination vectors are the same?

5. Now run the code with the following flags: -t -n 2 -l 100000 -d. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

6. What happens if you turn on the parallelism flag (-p)? How much would you expect performance to change when each thread is working on adding different vectors (which is what -p enables) versus working on the same ones?

7. Now let's study vector-try-wait.c. First make sure you understand the code. Is the first call to pthread mutex trylock()

really needed? Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

8. Now let's look at vector-avoid-hold-and-wait.c. What is the main problem with this approach? How does its performance compare to the other versions, when running both with -p and without it?

9. Finally, let's look at vector-nolock.c. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

10. Now compare its performance to the other versions, both when threads are working on the same two vectors (no -p) and when each thread is working on separate vectors (-p). How does this no-lock version perform?

第 33 章 基于事件的并发（进阶）

（写代码）

1*. First, write a simple server that can accept and serve TCP connections. You'll have to poke around the Internet a bit if you don't already know how to do this. Build this to serve exactly one request at a time; have each request be very simple, e.g., to get the current time of day.

2*. Now, add the select() interface. Build a main program that can accept multiple connections, and an event loop that checks which file descriptors have data on them, and then read and process those requests. Make sure to carefully test that you are using select() correctly.

3*. Next, let's make the requests a little more interesting, to mimic a simple web or file server. Each request should be to read the contents of a file (named in the request), and the server should respond by reading the file into a buffer, and then returning the contents to the client. Use the standard open(), read(), close() system calls to implement this feature. Be a little careful here: if you leave this running for a long time, someone may figure out how to use it to read all the files on your computer!

4*. Now, instead of using standard I/O system calls, use the asynchronous I/O interfaces as described in the chapter. How hard was it to incorporate asynchronous interfaces into your program?

5*. For fun, add some signal handling to your code. One common use of signals is to poke a server to reload some kind of configuration file, or take some other kind of administrative action. Perhaps one natural way to play around with this is to add a user-level file cache to your server, which stores recently accessed files. Implement a signal handler that clears the cache when the signal is sent to the server process.

6*. Finally, we have the hard part: how can you tell if the effort to build an asynchronous, event-based approach are worth it? Can you create an experiment to show the benefits? How much implementation complexity did your approach add?

第 37 章 磁盘驱动器

（完成这些作业题目前，阅读 HW\HW_Persistence\HW-Disk 目录下的相关文件。都可以用文本编辑器打开。运行 disk.py）

1．计算以下几组请求的寻道、旋转和传输时间：-a 0，-a 6，-a 30，-a 7，30，8，-a 10，11，12，13。

2．执行上述相同请求，但将寻道速率更改为不同值：-S 2，-S 4，-S 8，-S 10，-S 40，-S 0.1。寻道、旋转和传输时间如何变化？

3．同样的请求，但改变旋转速率：-R 0.1，-R 0.5，-R 0.01。几个时间如何变化？

4．你可能已经注意到，对于一些请求流，例如，-a 7，30，8，FIFO 并不是最好的，应该按什么顺序处理请求流？在相同的工作负载上运行最短寻道时间优先（SSTF）调度程序（-p SSTF）。每个请求服务需要多长时间（寻道、旋转、传输时间）？

5．使用最短的访问时间优先（SATF）调度程序（-p SATF）。它是否对-a 7，30，8 指定的一组请求表现有所不同？找到一组请求使得 SATF 表现明显优于 SSTF。或者更一般的情况，SATF 何时比 SSTF 更好？

6．对于请求流-a 10，11，12，13，运行 diks.py 会表现出什么糟糕的情况？你可以引入一个磁道偏斜来解决这个问题（-o skew，其中 skew 是一个非负整数）？考虑到默认寻道速率，偏斜应该是多少，才能尽量减少这一组请求的总时间？对于不同的寻道速率（例如，-S 2，-S 4）呢？一般来说，考虑到寻道速率和扇区布局信息，你能否写出一个公式来计算偏斜？

7．多区域磁盘将更多扇区放到外圈磁道中。以这种方式配置此磁盘，请使用-z 标志运行。具体来说，尝试运行一些请求，针对使用-z 10，20，30 的磁盘（这些数字指定了扇区在每个磁道中占用的角度空间。在这个例子中，外圈磁道每隔 10 度放入一个扇区，中间磁道每 20 度，内圈磁道每 30 度一个扇区）。运行一些随机请求（例如，-a -1 -A 5，-1，0，它通过-a -1 标志指定使用随机请求，并且生成从 0 到最大值的五个请求），看看你是否可以计算寻道、旋转和传输时间。使用不同的随机种子（-s 1，-s 2 等）。外圈，中间和内圈磁道的带宽（每单位时间的扇区数）是多少？

8. 调度窗口确定磁盘一次可以接受多少个扇区请求，以确定下一个要服务的扇区。生成大量请求的某种随机工作负载（例如，-A 1000，-1，0，可能用不同的种子），并查看调度窗口从 1 变为请求数量时，SATF 调度器需要多长时间（即-w 1 至-w 1000，以及其间的一些值）。需要多大的调度窗口才能达到最佳性能？制作一张图并看看。提示：使用-c 标志，不要使用-G 打开图形，以便更快运行。当调度窗口设置为 1 时，你使用哪种策略重要吗？

9. 在调度程序中避免饥饿非常重要。对于 SATF 这样的策略，你能否想到一系列的请求，导致特定扇区的访问被推迟了很长时间？给定序列，如果使用有界的 SATF（bounded SATF ，BSATF）调度方法，它将如何执行？在这里，你可以指定调度窗口（例如-w4）以及 BSATF 策略（-p BSATF）。这样，调度程序只在当前窗口中的所有请求都被服务后，才移动到下一个请求窗口。这是否解决了饥饿问题？与 SATF 相比，它的表现如何？一般来说，磁盘如何在性能与避免饥饿之间进行权衡？

10. 到目前为止，我们看到的所有调度策略都很贪婪（greedy），因为它们只是选择下一个最佳选项，而不是在一组请求中寻找最优调度。你能找到一组请求，使得这种贪婪方法不是最优的吗？

第 38 章 廉价冗余磁盘阵列（RAID）

（完成这些作业题目前，阅读 HW\HW_Persistence\HW-Raid 目录下的相关文件。都可以用文本编辑器打开。运行 raid.py）

1．使用模拟器执行一些基本的 RAID 映射测试。运行不同的级别（0、1、4、5），看看你是否可以找出一组请求的映射。对于 RAID-5，看看你是否可以找出左对称（left-symmetric）和左不对称（left-asymmetric）布局之间的区别。使用一些不同的随机种子，产生不同于上面的问题。

2．与第一个问题一样，但这次使用-C 来改变大块的大小（chunk size）。大块的大小如何改变映射？

3．执行上述测试，但使用-r 标志来反转每个问题的性质。

4．现在使用反转标志，但用-S 标志增加每个请求的大小。尝试指定 8KB、12KB 和 16KB 的大小，同时改变 RAID 级别。当请求的大小增加时，底层 I/O 模式会发生什么？请务必在顺序工作负载上尝试此操作（-W sequential）。对于什么请求大小，RAID-4 和 RAID-5 的 I /O 效率更高？

5．对于使用 4 个磁盘的不同级别 RAID（0、1、4、5），使用模拟器的定时模式（-t）来估计 100 次随机读取 RAID 的性能。

6．按照上述步骤操作，但增加磁盘数量。随着磁盘数量的增加，每个 RAID 级别的性能如何扩展？

7．执行上述操作，但全部用写入（-w 100），而不是读取。每个 RAID 级别的性能现在如何扩展？你能否粗略估计完成 100 次随机写入所需的时间？

8．用顺序的工作负载（-W sequential）运行上次的定时模式。性能如何随 RAID 级别而变化，在读取与写入时有何不同？性能如何随每个请求的大小的改变而变化？使用 RAID-4 或 RAID-5 时写入 RAID 请求的大小应该是多少？

第 39 章 插叙：文件和目录

（写代码）

1．**Stat**：实现一个你自己的命令行工具 stat，通过调用 stat()系统调用，作用于给定的文件或目录即可。将文件的大小、分配的磁盘块数、引用数等信息打印出来。当目录的内容发生变化时，目录的引用计数如何变化？有用的接口：stat()。

2．**列出文件**：编写一个程序，列出指定目录内容。如果没有传参数（比如 myls），则程序仅输出文件名。当传入-l 参数时（比如 myls -l），程序需要打印出每个文件的所有者、所属组、权限以及利用 stat()系统调用获得的一些其他信息。另外还要支持传入要读取的目录作为参数，比如 myls -l directory。如果没有传入目录参数，则用当前目录作为默认参数。有用的接口：stat()、opendir()、readdir()和 getcwd()。

3．Tail：编写一个程序，输出一个文件的最后几行。这个程序运行后要能跳到文件末尾附近，然后一直读取指定的数据行数，并全部打印出来。运行程序的命令是 mytail -n file，其中参数 n 是指从文件末尾数起要打印的行数。有用的接口：stat()、lseek()、open()、read()和 close()。

4.递归查找:编写一个程序,打印指定目录树下所有的文件和目录名。比如,当不带参数运行程序时,会从当前工作目录开始递归打印目录内容以及其所有子目录的所有内容,直到打印完以当前工作目录为根的整棵目录树。如果传入了一个目录参数,则以这个目录为根开始递归打印。可以添加更多的参数来限制程序的递归遍历操作，可以参照 find 命令。有用的接口：自己想一下。

第 40 章 文件系统实现

（完成这些作业题目前，阅读 HW\HW_Persistence\HW-VSFS 目录下的相关文件。都可以用文本编辑器打开。运行 vsfs.py）

1．用一些不同的随机种子（比如 17、18、19、20）运行模拟器，看看你是否能确定每次状态变化之间一定发生了哪些操作。

2．现在使用不同的随机种子（比如 21、22、23、24），但使用-r 标志运行，这样做可以让你在显示操作时猜测状态的变化。关于 inode 和数据块分配算法，根据它们喜欢分配的块，你可以得出什么结论？

3．现在将文件系统中的数据块数量减少到非常少（比如两个），并用 100 个左右的请求来运行模拟器。在这种高度约束的布局中，哪些类型的文件最终会出现在文件系统中？什么类型的操作会失败？

4．现在做同样的事情，但针对 inodes。只有非常少的 inode，什么类型的操作才能成功？哪些通常会失败？文件系统的最终状态可能是什么？

第 41 章 局部性和快速文件系统

（完成这些作业题目前，阅读 HW\HW_Persistence\HW-FFS 目录下的相关文件。都可以用文本编辑器打开。运行 ffs.py）

1*. Examine the file in.largefile, and then run the simulator with flag -f in.largefile and -L 4. The latter sets the large-file exception to 4 blocks. What will the resulting allocation look like? Run with -c to check.

2*. Now run with -L 30. What do you expect to see? Once again, turn on -c to see if you were right. You can also use -S to see exactly which blocks were allocated to the file /a.

3*. Now we will compute some statistics about the file. The first is something we call filespan, which is the max distance between any two data blocks of the file or between the inode and any data block. Calculate the filespan of /a. Run ffs.py -f in.largefile -L 4 -T -c to see what it is. Do the same with -L 100. What difference do you expect in filespan as the large-file exception parameter changes from low values to high values?

4*. Now let's look at a new input file, in.manyfiles. How do you think the FFS policy will lay these files out across groups? (you can run with -v to see what files and directories are created, or just cat in.manyfiles). Run the simulator with -c to see if you were right.

5*. A metric to evaluate FFS is called dirspan. This metric calculates the spread of files within a particular directory, specifically the max distance between the inodes and data blocks of all files in the directory and the inode and data block of the directory itself. Run with in.manyfiles and the -T flag, and calculate the dirspan of the three directories. Run with -c to check. How good of a job does FFS do in minimizing dirspan?

6*. Now change the size of the inode table per group to 5 (-I 5).

How do you think this will change the layout of the files? Run with -c to see if you were right. How does it affect the dirspan?

7*. Which group should FFS place inode of a new directory in? The default (simulator) policy looks for the group with the most free inodes. A different policy looks for a set of groups with the most free inodes. For example, if you run with -A 2, when allocating a new directory, the simulator will look

at groups in pairs and pick the best pair for the allocation. Run ./ffs.py

-f in.manyfiles -I 5 -A 2 -c to see how allocation changes with

this strategy. How does it affect dirspan? Why might this policy be good?

8*. One last policy change we will explore relates to file fragmentation. Run ./ffs.py -f in.fragmented -v and see if you can predict how the files that remain are allocated. Run with -c to confirm your answer. What is interesting about the data layout of file /i? Why is it problematic?

9*. A new policy, which we call contiguous allocation (-C), tries to ensure that each file is allocated contiguously. Specifically, with -C n, the file system tries to ensure that n contiguous blocks are free within a group before allocating a block. Run ./ffs.py -f in.fragmented -v -C 2 -c to see the difference. How does layout change as the parameter passed to -C increases? Finally, how does -C affect filespan and dirspan?

第 42 章 崩溃一致性：FSCK 和日志

（完成这些作业题目前，阅读 HW\HW_Persistence\HW-Journaling 目录下的相关文件。都可以用文本编辑器打开。运行 fsck.py）

1. First, run fsck.py -D; this flag turns off any corruption, and thus you can use it to generate a random file system, and see if you can determine which files and directories are in there. So, go ahead and do that! Use the -p flag to see if you were right. Try this for a few different randomly-generated file systems by setting the seed (-s) to different values, like 1, 2, and 3.

2. Now, let's introduce a corruption. Run fsck.py -S 1 to start. Can you see what inconsistency is introduced? How would you fix it in a real file system repair tool? Use -c to check if you were right.

3. Change the seed to -S 3 or -S 19; which inconsistency do you see? Use -c to check your answer. What is different in these two cases?

4. Change the seed to -S 5; which inconsistency do you see? How hard would it be to fix this problem in an automatic way? Use -c to check your answer. Then, introduce a similar inconsistency with -S 38; is this harder/possible to detect? Finally, use -S 642; is this inconsistency detectable? If so, how would you fix the file system?

5. Change the seed to -S 6 or -S 13; which inconsistency do you see? Use -c to check your answer. What is the difference across these two cases? What should the repair tool do when encountering such a situation?

6. Change the seed to -S 9; which inconsistency do you see? Use -c to check your answer. Which piece of information should a check-and-repair tool trust in this case?

7. Change the seed to -S 15; which inconsistency do you see? Use -c to check your answer. What can a repair tool do in this case? If no repair is possible, how much data is lost?

8. Change the seed to -S 10; which inconsistency do you see? Use -c to check your answer. Is there redundancy in the file system structure here that can help a repair?

9. Change the seed to -S 16 and -S 20; which inconsistency do you see? Use -c to check your answer. How should the repair tool fix the problem?