应用程序稳态测试系统

计 52 于纪平 2015011265

计 55 张瑞喆 2015011371

计 55 王逸松 2015011369

二〇一八 年 五 月

摘要

NOI（全国青少年信息学奥林匹克竞赛）系列赛、ACM-ICPC（ACM 国际大学生程序设计竞赛）是一类开展范围较为广泛的程序设计竞赛，每年共计有上万人参加这些比赛。比赛以算法为设计为主，要求参与者编写算法程序完成指定的任务。在这些竞赛中，编写的程序往往较为简洁，通过处理指定格式的输入数据计算得到结果，再将得到的结果输出，以进行评判。而对程序评测的过程中，由于操作系统调度等多方面的因素，可能会出现多次运行得到的运行时间、内存空间等项目测定不准确的问题，导致同一份代码多次运行得到的分数不同，也给评测工作引入了不稳定因素。

JOS 操作系统是 MIT 的一款教学操作系统，采用微内核的结构，支持内存管理、进程管理、多任务、文件系统、网络等功能，可谓“麻雀虽小，五脏俱全”。关于我们工作关心的重点部分，JOS 的内存管理与进程管理模块实现简明，代码结构清晰，利于修改。

本工作以实现稳态操作系统为目标，对 JOS 操作系统的各方面进行了一定的修改，使得 JOS 操作系统通过特殊的系统调用限制，在满足让应用程序在真实环境上运行的前提下，减少了外部不稳定因素对被测用户程序带来的影响。为竞赛工作提供更好的支持，提高竞赛程序评测工作的公平性、稳定性。

关键词：用户程序稳定性；JOS；程序设计竞赛

目录

引言

问题背景

在 NOI（全国青少年信息学奥林匹克竞赛）系列赛、ACM-ICPC（ACM 国际大学生程序设计竞赛）等算法与程序设计竞赛中，赛题以编写算法为主要考察点，对参赛选手的算法能力进行考察。

这类竞赛中，赛题要求选手实现的程序包含以下几个部分：

1、 输入——作为程序从外部获取数据的环节

2、 计算与处理——程序通过简单的循环、判断语句，而不加调用外部程序地，得到输入数据所对应的答案

3、 输出——作为程序向外部反馈算法运行结果的步骤，将结果通过输出反馈给外部程序

赛题还同时规定选手程序所能使用的时间（通常为 1 秒或数秒），程序能使用的内存空间限制（程序所能使用的内存空间大小）。

输入和输出的形式都由赛题规定，通常是由文件读入保存在指定的文件中的规定格式的数据，在运行结束后，以赛题规定的形式输出到指定的文件中。再由其他程序比对输出结果与标准答案，两者一致方可认为结果正确。

在程序的运行过程中，如果出现下列情况之一，则会判定选手程序不符合要求，不比对结果而直接判为 0 分。

1、 超过时间限制——选手程序在规定的时间限制内没有结束，在超出时间限制时，会从外部结束选手程序，并判定为超时。

2、 超过内存限制——选手程序使用的内存空间超过了规定的内存空间，程序会被终止，

并判定为超出内存限制。

3、 运行时错误——选手程序在运行时崩溃，常见的情况包括访问了未定义的地址空间，或除数为 0 等。

4、 非法系统调用——选手程序进行了在竞赛规则中不被允许的系统调用，可能具有攻击系统的行为，因此被终止，判定为非法系统调用。

问题现状

在现有环境下，所有大型比赛的评测都是在 Linux 操作系统（通常为 Ubuntu 的 LTS 版本）下进行。评测软件为运行在操作系统下的专用软件，本质为可执行程序。

这些软件在进行评测时，先将被测程序编译成可执行文件，然后将可执行文件存放到一个空的文件夹中。再将输入文件移动到同一目录下。

接着，评测软件开始执行可执行文件，并将输入重定向到该输入中。若程序结束，则比对输出文件和标准答案，当二者一致时则给出满分的评价，否则给出零分的评价。特别地，在可执行文件执行过程中，若使用时间超过时间限制太多，则评测系统会杀死该进程，并判定程序运行超时。

在这个过程中，程序完全暴露在 Linux 系统环境中运行，既没有对其读取非法文件进行限制，也没有对其进行危险的或不合理的系统调用进行限制；目前，在这方面的改进主要是通过相关工具对程序的系统调用及其参数做限制。同时，程序的实际运行效率并非只和程序本身及输入输出相关，和系统中运行的其他进程也有关系。

总之，程序在运行过程中，没有与系统进行隔离，程序对系统的影响和系统对程序的影响都有可能导致不公平的发生。因此，有必要开发一款操作系统，可以在效率等方面稳定评

测参赛者程序，同时能阻止非法系统调用的发生，达到相对的安全。

本文的工作和组织

本文第一段中引入了问题核心和解决目标——编写一个能准确测定程序运行效率，对程序进行客观评价的操作系统。

本文第二段中详细介绍了现有竞赛的具体形式和内容，描述了评测方对选手程序进行评价时的手段和方法，并指出其中的不稳定性，这些不稳定性来源于系统设计，并可能导致评测不公平。同时给出在交互形式上进行调整以减少不确定性的方案。

本文第三段介绍了我们基于的操作系统 JOS，我们选择 JOS 这样一个轻量级操作系统，进行了一些尝试。

本文第四段中对我们在 JOS 上的各方面改动做出了详细介绍，并说明了这些改动能对实际问题产生的意义。

本文第五段介绍了我们的实验结果，并对不稳定性进行了量化的分析。

本文第六段最后总结了相关工作并为未来得及进行的工作进行了设想，对将工作更完善地移植到 Linux 系统上做出了规划。也讨论了进一步对程序进行分析而非只进行评判的可能性。

竞赛形式、内容与评测不稳定问题

竞赛形式、内容

在现有的主流算法与程序设计竞赛中，命题通常以考察具体的数据结构、图论、动态规划等方面的知识为目的。要求选手在限定的时间（通常为 3.5 至 5 小时）内编写程序对输入数据进行高效、正确的处理来解决这些问题。题目通常会规定以下几个部分的内容，来告知

选手需要解决的问题及解决的具体形式：

1、 题目描述

2、 输入格式

3、 输出格式

4、 输入样例

5、 输出样例

6、 数据规模与约定

下面分别介绍这几部分内容的具体含义：

题目描述：

这一部分中会介绍题目要求选手完成的任务，即要计算出何种信息的描述。

例如：对 n 个非负整数进行升序排序，得到这 n 个非负整数从小到大排序的结果。

输入格式：

这一部分会描述输入数据的具体格式，以便于程序进行读入。

例如：输入数据包括两行，第一行包括一个正整数 n，表示要被排序的非负整数的个数；第二行依次给出这 n 个数，相邻两个整数之间用一个空格隔开。

输出格式：

这一部分会描述输出数据的具体格式，程序必须严格按照这个要求输出正确的答案才能得到分数，输出答案错误或格式不正确的不得分。

例如：输出数据只有一行，包含 n 个正整数，为输入的 n 个整数按照从小到大的顺序排序后得到的结果。相邻两个整数之间用一个空格隔开。

输入样例：

这一部分会给出输入数据的一个具体例子，以帮助选手理解输入格式。

例如：

3

5 2 7

输出样例：

这一部分会给出输出数据的一个具体例子，对应于输入样例的答案，以帮助选手理解题意和输出格式。

例如：

2 5 7

数据规模与约定：

这一部分给出输入数据的具体限制，如大小范围等。

例如：$1 \leq n \leq 10^7$ ，a 中的元素都在 unsigned 范围内。

交互方式本身的不稳定性及其修改

根据上述题目要求写出的程序，其本身就具有一定的不稳定性，这一部分不稳定性通过对题目约束要求本身进行一些修改就可以得到解决。这些修改对题目考察选手的内容没有本质的改变，却能在某些情况下大大减少选手程序的不稳定性。

具体来说，用户程序在整个输入或输出的过程中，需要进行大量的系统调用，同时和文件系统进行交互。虽然，等待硬盘读取数据的时间不会被全部计入用户时间，但是却会因进程切换等因素对程序运行效率造成了影响。

要想消除这些因素，就必须要设法减少程序输入输出过程中的中断。而输入输出又是程序与外界交互的重要手段。因此，既不能要求程序不进行输入输出，又不能将输入输出使用的这段不稳定的时间计入用户时间。要想达到这目标，就必须要对题目中的输入输出方式进

行修改。

一种合理的考虑是，既然我们要考察参赛者的算法与程序设计水平，输入输出不作为其中考察的重点，可以舍去。但舍去输入输出后，给程序以输入和输出的渠道仍然是必要的。

我们可以将输入输出的时间从被测程序的用户时间中去掉，即：让参赛者编写一个函数而非完整的程序，该函数需要按照题目的要求编写，实现指定的功能。这样一来，我们可以把这个函数和我们编写的主程序放在一起共同编译，得到一个可执行文件。在这个可执行文件中，参赛者编写的部分是确定的，我们设法先将需要读入的内容从文件中载入到内存中，之后再开启计时功能，并在选手函数结束后或超时时停止计时，再进行比对。

这样一来，通过对函数传递参数（数个具体的数字和一些指针），其中指针可以用来描述一段连续的地址来进行输入。对于程序输出运行结果的部分，也做类似的处理，即传入参数中包括需要反馈的结果的地址，程序将结果按规定的格式存储到这些地址即可。通过这两个操作，可以解决函数与外部的输入输出交互所带来的不稳定性问题。

而函数的具体形式和传递内容应当是和题目及要完成的目标相关的，因此该函数本身应当由题目提供定义，并同时规定参数传递的方法。

下面还是以排序问题为例，介绍在这样的改动后，题目的形式和发生的变化：

题目描述：

要完成的任务和之前一致，不会发生任何变化。

例如：对 n 个整数进行排序，按照从小到大的顺序重新排列。

定义函数：

给出要完成的函数在头文件中的定义形式，参赛者应当写出函数的实现。

例如：void sort(unsigned *a,int n);

传入参数格式：

规定函数中传入参数的格式，同时定义每个参数在问题中的具体含义。

例如：传入参数中，n 表示需要排序的整数的个数。待排序的 n 个整数依次存放在 a[]中。

任务描述与结果传递方法：

规定了任务的结果放置在哪一块内存空间里（给出具体需要存放地址的指针），这个地址通常在函数参数中进行描述。

例如：在 sort 函数中，将传入的 n 个整数排序后，存放在 a[]数组中，并正常结束 sort 函数。(这个例子有些特别，在这个例子中，传入参数和输出地址使用了同一块内存空间，实际上并非每次都必须如此)。

函数举例：

函数的一个例子及其正确的执行结果的描述。

例如：调用 sort 时，a[]中存放的内容为三个 unsigned 类型变量 5,2,7，n 中传入 int 型变量 3。函数结束时，a 中存放的内容应当被修改为三个 unsigned 类型变量 2,5,7。

数据规模与约定：

数据规模约定与之前一致，不会有任何变化。

例如：$1 \leq n \leq 10^7$ ，a 中的元素都在 unsigned 范围内。

评测工作介绍

现有的主流评测软件均在 Linux 系统上运行，在评测一个程序时，先在 Linux 系统中，利用命令 **time -U ./[test] <[infile] >[outfile]**运行被测程序。其中，[test]为被测可执行程序名，[infile] 和 [outfile] 分别为输入输出文件，-U 表示测定用户时间。

在运行过程中，若用户程序使用的实际时间（realtime）超过时间限制 1s 以上，则会终

止被测程序，并认为用户时间也超过时间限制，否则，在选手程序运行结束后再根据反馈信息判定选手程序是否超时。

在运行过程中，若出现被测程序崩溃（运行时错误）或监测到被测程序使用了非法的系统调用或监测到被测程序使用的空间超过限制，则终止评测，并做出相应的判定。

除此之外，当选手的程序在限制的时间、存储空间内求出答案并正常结束时，评测软件将先判定选手程序的 user time 是否也符合要求，之后比对被测程序的输出文件和标准答案，若二者一致，则判定选手程序结果正确，得到满分。

关于系统调用方面，现行评测软件则完全依赖于 Linux 的权限控制，既没有沙盒也没有其他的特殊限制。

这样进行的评测至少包括以下几个可能导致不公平的因素：

1、 选手程序可能会进行系统调用或非法读写，导致测试程序违规获得分数；

2、 由于外部程序或时钟导致的系统中断，会影响程序运行时间；

3、 由于进程切换导致的程序运行时间波动。

为了减少程序运行时的不确定性，需要对系统进行各方面的改进。以使被测程序运行在一个尽可能地“干净”的系统中。

评测工作的不稳定性及原因分析

在这一段中，详细讨论程序在上述的现有评测方式中运行会导致的程序出现的时间上的不稳定性，并给出其原因所在。

程序运行的不稳定性主要指的是运行时间的不稳定，可能的因素有以下几项：

1、 中断及调度带来的不稳定性

在程序运行的过程中，操作系统可能会产生一些中断或是系统调度。中断可以

是时钟中断，也可以是来源于其他设备的中断。调度则是操作系统对进程进行切换所带来的一种操作系统的特性。

对于中断来说，操作系统如果在执行被测程序的过程中处理中断，虽然处理中断的时间不会被计入被测程序运行时间，但是，中断至少会产生以下几个效果导致程序运行变慢：

- TLB 被清空，中断结束后程序需要重新建立 TLB，会消耗程序的运行时间。
- L1 Cache 被清空，导致原本存储在 L1 Cache 中的内容在访问时需要额外的时间。
- L2 Cache 和 L3 Cache 中内容也在中断中被修改，虽然没有全部丢失，但也会造成影响。

对于进程切换来说，会导致程序效率降低的内容除了和中断相同的几点外，L2 Cache、L3 Cache 的丢失往往比普通的中断来得更加严重。

2、 物理内存分配不同带来的不稳定性

在程序运行的过程中，CPU 会根据历史使用情况和 Cache 调度策略，确定出 Cache 中保存的内容。基于程序具有的时空局部性特征，这样的 Cache 规划有利于程序效率的提高。

然而，在实际分配过程中，Cache 映射中往往采用多路组相联的映射方式进行，即：物理内存中的每个页，可能分配到特定的一些 Cache 页中。有些页会映射到同一些 Cache 页中，另外一些页映射到的 Cache 页则和这些 Cache 页完全没有交集。这样一来，几个页是否被分配到同一个组相联中就和它们被分配的物理地址相关。在极端情况下，高频的若干个页在同一个组相联中，它们会互相淘汰，缺失率上升，而在另一些组相联中，空间则浪费严重。

而在程序开始时，程序被分配到的内存是和当前空闲的内存相关的，而非稳定

的。这就导致了 Cache 映射和程序开始时的系统映射相关，进而影响到 Cache 缺失率，并最终影响到程序运行效率。

3、 其他设备共享资源带来的不稳定性

在多核 CPU 中，L3 Cache 是由多个 CPU 共享的，因此，其他 CPU 进行的操作会影响到对 L3 Cache 的修改，进而影响到缺失率并最终对程序运行效率产生一定的影响。

同时，在多 CPU 共享内存带宽或是 DMA 对内存的访问都有可能挤占内存带宽，影响到运行着待测程序的 CPU 访问内存时的等待时间，进而影响到运行效率。

综上所述，以上三个方面的问题都会影响到（多是拖慢）程序运行的效率，最终造成评测的不公平。在我们的工作中，我们将针对这些问题，修改 JOS 操作系统，使其能稳定地进行评测工作。

其他需要改进的内容

除了效率方面，在评测安全性方面，直接在 Linux 系统上评测也存在不稳定因素。

在 Linux 系统中直接运行用户程序时，程序不仅有权访问系统上的其他文件，还可能通过系统调用，对整个评测系统及评测所用到的程序进行攻击。这种恶意的攻击不仅可能会导致评测软件崩溃，还有可能造成误判，甚至影响到其他被测程序的结果。

因此，在评测系统中注重安全性也是非常必要的，而系统调用往往不能全盘取消，否则会导致评测程序的某些功能无法实现。所以需要设计一种方法，既能保证评测程序能完成任务，又能保证被评测程序在运行过程中不能进行我们不期望的系统调用。目前，某些先进的基于 Linux 的评测方案中使用一些工具对系统调用及其参数进行了限制，已经取得了比较好的效果。

JOS 系统简介

JOS 设计思想

JOS 是 MIT 6.828 操作系统课程的教学操作系统。它的各部分功能简单而完善，作者设计的 6 个实验内容依次为：引导、内存管理、进程管理、多任务、文件系统、网络。

JOS 采用微内核的设计思想，相当多的操作均不是通过单个系统调用完成的，而是借助用户库，通过一系列的操作完成。并且，文件系统的相关操作由专门的文件系统服务进程完成，操作系统仅负责进程间通信部分；网络相关操作也有一个专门的网络服务进程，操作系统对其提供一些收发数据包的基础系统调用。

完成上述全部实验之后，JOS 仅有以下简单的系统调用：

屏幕/键盘输入输出：sys_cputs, sys_cgetc

进程管理：sys_getenvid, sys_env_destroy, sys_env_set_status, sys_env_set_trapframe

创建空白新进程：sys_exofork

页表管理：sys_page_alloc, sys_page_map, sys_page_unmap

自定义页错误处理程序：sys_env_set_pgfault_upcall

调度：sys_yield

进程间通信：sys_ipc_try_send, sys_ipc_recv

获取时间：sys_time_msec

网络：sys_net_try_transmit, sys_net_try_receive

通过上述系统调用，能够实现各种复杂的功能，例如支持 copy on write 的 fork、各类文件操作、socket 编程接口、抽象的文件描述符（可能是文件、套接字、管道、屏幕/键盘）……

JOS 代码组织结构

inc 文件夹存放了一些通用的头文件，这些文件对于操作系统内核和用户程序都是适用的，是一些通用的定义。

boot 文件夹存放了有关引导的相关代码，编译后将放在硬盘的引导扇区；引导程序获得控制权后将从兼容 IDE 控制的硬盘中读取内核的代码并转交控制权。这部分代码仅在通过硬盘直接引导时执行；通过 GRUB multiboot 方式启动时不执行。

kern 文件夹存放了操作系统内核的相关代码，其中大部分是 C 代码，小部分是汇编代码；此文件夹的头文件仅被内核代码引用。这些源程序经编译（或汇编）并链接后形成了内核的二进制文件（实际链接时还会直接置入一些用户程序的 ELF 文件，以便在文件系统功能尚未建立时启动一些用户进程）。

lib 文件夹存放了用户库的代码，由于 JOS 不支持动态链接库，该文件夹下所有的 C 代码将会和每个用户程序的代码在被编译后一同链接。

user 文件夹存放了用户程序的代码，每个 C 代码将编译成为一个用户程序。用户程序的接口为 void umain(int argc, char **argv)。

fs 文件夹存放了文件系统服务进程的代码和一些需要被直接以文本文件形式置入文件系统；这里的 Makefrag 文件也指定了哪些文件需要被放到文件系统当中。文件系统在区别于内核映像磁盘的另一个磁盘中，JOS 的默认配置是引导与内核代码存储于 0 号 IDE 控制器的设备 0 中，而文件系统所在磁盘是 0 号 IDE 控制器的设备 1。文件系统的映像是编译内核的计算机上的某个 C 程序初始建立的，当然可以在 JOS 运行时进行写入。

net 文件夹存放了网络服务进程的代码和 lwIP 移植的相关代码（网卡驱动程序在 kern 文件夹中）。网络服务进程部分将编译为一个单独的文件，JOS 启动时将作为一个单独的进程运行；lwIP 将编译为一个（静态）库，最终与用户程序一同链接。

JOS 系统运行逻辑

内核映像被装入内存并接过控制权后，依次进行以下方面的初始化：

- 完成 ELF 加载过程，自我初始化 BSS 段为 0；初始化屏幕输出
- 启动物理内存管理，建立内核页表
- 初始化进程管理结构
- 建立中断表并应用
- 启动其他处理器，启动 Local APIC，配置 PIC
- 扫描 PCI 设备，初始化网卡
- 启动文件系统服务进程、启动网络服务进程
- 启动第一个用户进程

上述初始化完毕后，用户进程的系统调用和外部的中断将会使计算机返回到内核。此时内核将根据中断的情况进行进一步的处理：

- 系统调用：根据系统调用的接口定义，转对应的实现函数进行处理
- 页错误：如果可能，转用户定义的页错误处理程序，否则结束当前的用户进程；如果是内核运行出错，则 panic
- 时钟中断：记录时间，调度其他进程运行
- 键盘、串口中断：转对应的处理程序记录相应的信息
- 伪中断（spurious interrupt）：忽略
- 其他：结束当前的用户进程；如果是内核运行出错，则 panic

从 JOS 到竞赛评测系统的修改

运用 JOS 进行评测的方案

JOS 本身具有的系统结构简单且完善，核心代码相对于 Linux 操作系统来说较短的特性，使得它非常适合被选作竞赛评测用的操作系统。

但我们同时也注意到了，现在的 JOS 系统中，程序是在一般的操作系统环境中运行的。如第二节所述，程序在这样的环境下运行会受到系统调度和内存分布两方面的外部环境影响，测出的效率不能很好地说明程序实际的性能，因此，有必要进行一系列的修改。

分析可知，我们设计的这个操作系统，不仅需要满足在一般情况下能正常进行调度运行，还需要特别地进行修改，以便在评测时能达到稳定状态。同时，还必须注意安全性，被测程序在这里是不被信任的，它可能会通过非法系统调用来获取非法信息，也有可能由于内部错误发生崩溃。在这两种情况下，我们分别要做到在程序进行非法系统调用时将其杀死并判为 0 分，在被测程序错误发生崩溃时则能够正常退回到评测器中并判为 0 分，而非导致整个系统崩溃。

此外，我们还发现，JOS 操作系统存在一些错误，我们在工作中也予以修正。

我们对 JOS 主要进行了以下方面的修改：

- 对于评测开设特殊的环境，设置系统调用来进入和退出特殊模式
    - 在评测模式下，时钟中断的间隔被特殊设置
    - 在评测模式下，关闭其他外部中断
    - 在评测模式下，禁用系统调用（但在评测程序允许的情况下，可以开放某几种系统调用）
- 对于评测所使用的内存，设置固定的内存空间供评测程序使用

    - 对于这一点，需要在评测前整理内存，使得被评测程序依次占据了物理内存的同一段连续区域
- CPU 限制
    - 只开启一个 CPU，避免多 CPU 因为共享 L3 Cache 等原因互相影响
- 为了更完善的功能，我们做出了这些方面的修改
    - 在原有 e1000 网卡驱动的基础上，添加了 e1000e 网卡驱动（尚未找到合适的硬件）
    - 修改内核 entry（JOS 原有支持有错误，无法启动 multiboot）
    - 修复 JOS 的 bug（或 feature），原本 JOS 仅能支持最大 128M 物理内存
- 为了便于实现，我们还做出了这些方面的修改
    - 文件系统加载在内存中（由于没有合适的兼容 IDE 的物理机）

程序调度方式的修改

在操作系统中，每当一个时间片段结束后，会将当前正在运行的进程切出，放进等待序列中，并从等待序列中按照一定的规则取出进程，激活进程并进入执行状态。在一般情况下，这种机制保证了操作系统中各个进程之间 CPU 资源的较为合理的一种分配机制，避免一个进程占用过多的时间片，其他进程等待过久。

但是，进程的切入和切出需要耗费一定的代价，当进程重新被切入到 CPU 中进行执行时，面对的是一个空的 TLB 和被清空的 L1 Cache 和 L2 Cache 及部分被清除的 L3 Cache。程序无法使用到切出之前存在这些地方的内容，需要重新载入，而缺失导致的载入是计入到运行时间内的，也就是说，频繁的进程切换会导致用户程序不能很好地享受到系统带来的优化，程序效率会变慢，其中增加的时间是进程切换所直接或间接造成的。

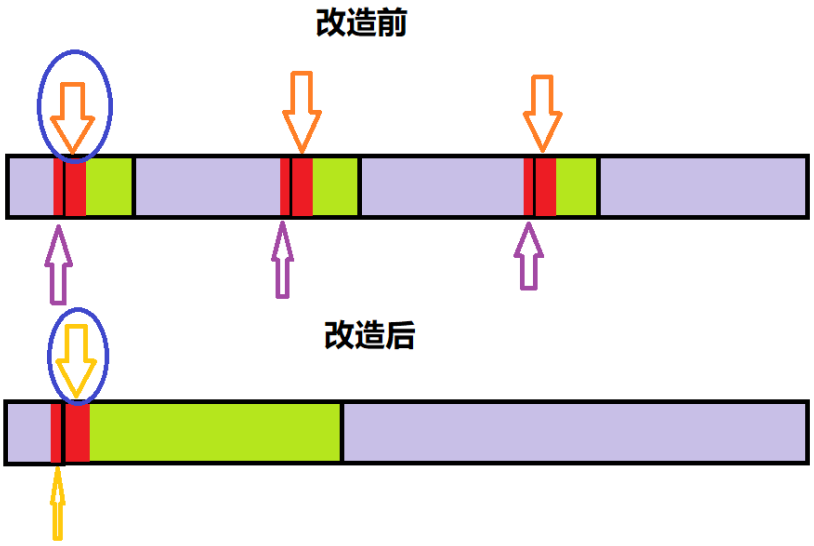
从稳定性上来说，进程的切换发生在何处也是一个未知数，如果一个程序在两次运行时，发生切换时，一次恰好运行结束一段连续访存的函数，之后不再访问这段内存，而另一次运行时则在运行到这个进程中间时发生，则后一次运行的 Cache 会多出一系列的必然缺失，导致效率被拖慢。这种差距的出现不仅不能自我修正，第一次数条指令的差距往往会导致第二次切换时数百条指令的差距，造成的影响值得重视。

此外，在多次运行时，还要面临的一个重要问题是：在计算时间时，使用的计时方法是把多个时间片段算术求和得到总时间。然而，系统计时本身会存在一定的误差，一次计时的本质是两个时钟相减得到的差值，如果一次计时的标准差是 σ ，则分 N 段计时（这 N 段的时间并非连续运行的，可以认为是独立同分布的），根据数学知识可以知道，标准差就会达到 $\sigma\sqrt{\mathrm{N}}$ 。

因此，解决调度问题非常重要，一个合理的做法是，对于被测程序，在测试过程中，我们不对其他程序进行调度，让它连续使用 CPU 和各级 Cache 资源进行运算。即在运行过程中，我们不进行时间片的划分，停止根据时间片调度的行为，以便于精确计时。

值得注意的是，我们还需要考虑这个程序超时时，系统不能一直让其运行下去，应当将其终止，并判为超时。

对于程序计时技术和超时判定，以下两个例子能说明这个问题：



在图中，整个大矩形从左到右代表了时间轴，淡紫色的部分代表了系统中其他程序运行的部分，绿色区域为被测程序运行的时间部分，和淡紫色区域相邻的红色部分表示进程切换的时间损失，这部分时间损失在系统时间内，不会计入程序运行的时间损失，对稳定性不造成影响。而和绿色区域相邻的红色区域代表的是刚进入进程时 TLB 和 Cache 缺失造成的时间损失，事实上，这种时间损失以一定形式分布在程序运行的整个过程中，而非一定仅仅处在开头。这部分时间损失会被计入总运行时间中，对评测稳定性造成影响。值得注意的是，在程序开头（蓝色圈出的箭头所指处），此处发生的缺失是程序开始运行时的必然缺失，与进程切换无关，且在下文中结合内存分配策略，我们可以发现这部分缺失是稳定的。这个例子对比了在一般操作系统下和我们改造过的操作系统下的程序运行状态，并标出了不稳定性发生的区域，可以看出，系统在经过我们设计的方法特殊处理后，稳定性更好。

系统中断设计的修改

尽管我们已经通过减少进程切换来降低程序的 Cache 缺失，但除了进程切换的因素之外，系统内部或外部发生的中断也会导致 TLB 和 Cache 信息的丢失，因此，还需要采用一

些手段来屏蔽中断。主要会发生的中断可以分为两类：

1、 时钟中断

2、 其他中断（键盘、网络等）

对于键盘、网络的中断信息，我们可以采用屏蔽的方式来加以解决，只要在评测时关闭这些中断，就可以进一步提升系统的稳定性。

但是，对于时钟中断，不能采取屏蔽的方法加以解决，屏蔽了时钟中断意味着将无法获取时钟，无法控制程序运行的时间。一旦程序超时，可能会导致整个系统无限运行。我们必须使用一种方法，既能够完成计时的任务，又能够避免评测过程中出现时钟中断，对效率造成影响。

这时我们发现，时钟中断的周期是可以进行调整的，因此，我们在这里进行了适当的修改。具体来说，对于时钟中断，可以在 JOS 系统中对其频率进行修改，并且将周期性的中断改为每次需求的单次中断，这就意味着，我们可以很好地利用这个性质来解决问题。我们在进入评测时，将下一个时钟中断的位置设为被测程序的时间限制上界，在被测程序退出（包括主动结束和被动退出）后，改回正常的频率。

通过这样的设计，不仅可以保证在被测程序运行过程中不会发生时钟中断，还能够很好地利用下一次的时钟中断，一旦时钟中断发生，就必然意味着被测程序超时。所以，系统一旦收到时钟中断，就可以立即终止被测程序的运行，并由评测器宣布其超时。

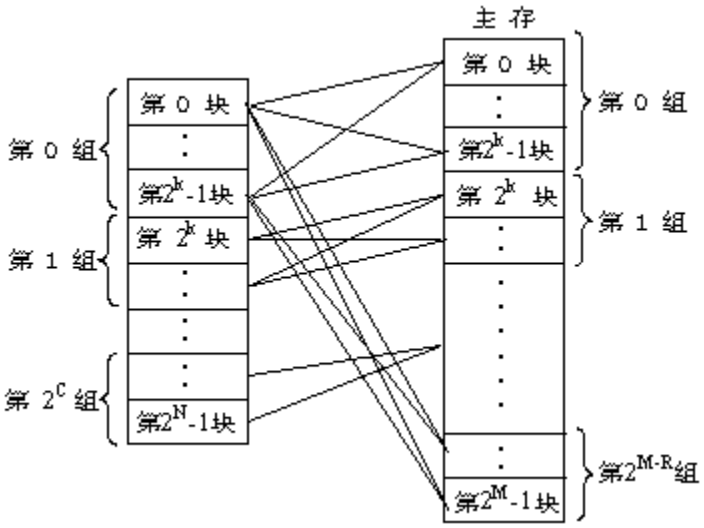
一个具体的例子可以见下图，我们的系统实现的效果从理论上要远好于原有的 JOS 系统，

内存分配方案的修改

在内存分配方案中，考虑到稳定性要求，我们需要设计一个功能，保证每次运行同一个

程序的时候 Cache 的命中是一致的。我们发现，Cache 的命中除了在系统中断中会受到影响外，也会受到程序开始运行时内存空间分配的影响。由于现代计算机中的 Cache 映射通常会采用多路组相联的方法进行，如果相同的一个程序在两次运行中，分配到的初始内存不同，那么，这两个程序的运行效率可能会完全不同。

多路组相联的 Cache 设计意味着，内存块被划分成了若干组，每一组内的内存块都只能和与其特定的一组 Cache 相连。一种常见的映射方式如下图所示，主存中第 i 组都只能映射到 Cache 中的第 $\mathrm{i} \bmod 2^C$ 组。



我们先假设内存的映射情况为：

内存块编号	映射到 Cache 块编号
$0-15$	$0-15$
$16-31$	$16-31$
……	……
$256-271$	$0-15$
$272-287$	$16-31$
……	……

$512-527$	$0-15$
$528-543$	$16-31$
……	……

那么，同一个程序两次运行发生的 Cache 冲突的例子是容易构造的，因为普通的操作系统在分配内存时，会将空闲的内存空间直接作出分配。因此，这种构造本质上可以归结为构造同一个程序在内存块内的地址分配不同的两种情况即可。这两种情况下，Cache 命中会因为映射的关系有较大的不同，从而产生明显的实际上影响程序效率的后果。

下列情形中的 for 循环语句，与一般的 for 循环没有区别，特别地，用 visit（x）来代表一个访问内存中第 x 个块的指令。

情形 1：

```
for(k=0;k<1000;++k)
   for(i=0;i<256;++i)
      visit(i)
```

情形 2：

```
for(k=0;k<1000;++k)
   for(i=0;i<16;++i)
      for(j=0;j<16;++j)
         visit( (i<<4) or j)
```

以上两个程序，表示的内容可能是同一段代码执行时因为系统内存分配不同所发生的不同的访存，从上面两种情况我们可以看出：在第一种情形中，只在 k=1 时发生了缺失，之后被访问的内容都在缓存中。而对于第二种情形，情况就糟糕地多，被访问的内容因为互相挤占 Cache 中地址为 0-15 的部分，互相竞争、剔除，每一次访存都有缺失发生。

通过对比可以知道，物理内存地址的分配也会对程序造成影响，因此必须对其进行规范化的改造。

如果每次运行时，系统都会分配同一段内存给被测程序，则不会发生这种问题。然而，固定的一段内存在评测时并不总是空着的，往往还因为其他程序的运行被不同程度的占用。为此，我们可以对这部分内存进行调整，具体的改造方法是，在操作系统中实现一个内存整理功能，每当需要进行评测时，将所有已经被使用的内存片段（包括被测程序即将使用的内存）进行整理，整理的原则是同一进程的物理内存连续，且被测程序的物理内存放在所有物理内存的结尾。这样方可保证被测程序分配到的空间是稳定的。

需要特别说明的是，内存的稳定分配和以上两点不同，它仅能增加程序运行时间效率的稳定性，但对于性能方面的影响，并不会起到“加速”的作用。这是因为，不同的程序对缓存的依赖是不同的，有时零散的或特定形式的内存分配反而能提高程序的运行效率。

下面是一个内存整理的具体例子：

整理前

整理后

如图所示，图中，绿色的部分表示的是空闲的内存单元，而其他颜色则表示被不同的程

序所占用的内存单元。内存整理是一个较为简单的功能，其目的是为了将指定区域空余出来，用于提供给被测试程序专用。而整理的方法则是将内存中存储了信息的片段，平移到内存空间的尾部，而将低地址段的内存空间留给被测程序使用。可以看出，在这个例子中，整理之后，低地址段出现了大量空闲的内存空间可以提供给被测程序使用，被测程序将分配到连续的内存空间。

Arbiter 评测器的思路与目的

有必要说明的是，我们在操作系统上实现的评测软件——arbiter 评测器所具有的作用。Arbiter 评测器首先作为一款软件，运行与我们的系统之上。作为软件的一部分而非操作系统的一部分，它更易于修改，添加新功能等。其中集成了一些对于我们适用范围内的所有题目都可以复用的功能，例如解析操作系统对于评测结果的返回值并将其以友好的方式输出等。

而对于交互库来说，它根据题目而异，由命题人编写，根据题目特性和需求不同，具有丰富多彩的功能。灵活性更高，但安全性和可信度更差，因此在设计中，它不能独立进行系统调用，必须经过 Arbiter 来确保系统安全。另一种说法是：交互库编写的错误和失败会导致程序评测失败，没有评测结果，但不会也不应该会导致系统的崩溃。在交互库出错时，我们仍然能通过 Arbiter 得到“交互库出错”的提示信息，并继续评测其他程序。我们认为交互库是部分可信的，某些参数（例如允许的运行时间等）对于每道题目有所不同，应当由交互库进行设置。我们也不能因此认为这个系统是“不安全的”，因为一个通常的 Linux 系统也没有对此类似的交互程序进行防范（例如产生大量线程或写满磁盘）。

对于选手程序，这是整个系统中最不可信的部分，选手可能会采取各种手段来攻击系统，因此，选手程序具有最低的权限，大量的不必要的系统调用被限制，以防止选手进行恶意操

作。选手程序只能进行（由交互库规定的）少量的系统调用，一旦使用规定之外的调用，则选手程序会被判为违规，并立即结束。

实验验证

实验验证环节分为两部分内容介绍：

第一部分先介绍了面向用户的系统调用交互接口的设计，从用户的角度理解接口的具体功能。接着介绍了真机测试环境（安装到真机）的具体操作方法，最后介绍了 Arbiter 评测器（作为系统内置软件）的运行和调用方式。

第二部分是实验测试部分，一共做了三个实验，分别是：

- 和 Linux 系统的稳定性对比
- 控制变量法研究中断对程序造成的影响
- 控制变量法研究内存整理对程序造成的影响

系统调用的实现

为了实现上述功能，我们在修改 JOS 系统的过程中，将其整合成了三个系统调用，以便于与用户交互。

具体为：

int sys_enter_judge(void *eip, struct JudgeParams *prm)

被测程序调用，来申请进入评测状态。操作系统将阻塞当前进程，直到有其他进程（arbiter）同意该进程进入评测状态。操作系统将安排被测程序跳转到 eip 处执行，直到评测结束后（或传入了非法参数）该系统调用返回。

该系统调用的返回值可能为：

- -E_INVAL，表示参数非法，例如时间限制过长
- 0，正常结束

struct JudgeParams *prm 中包括下列内容：

- int ms, kb；表示评测的时间限制（以毫秒计）和空间限制（以 KB 计）
- int syscall_enabled[NSYSCALLS]；表示每种系统调用是否被允许
- void *data_begin；表示被测程序在评测状态下允许写的虚地址起始位置；终止位置由该地址加上空间限制得出，且评测开始时栈指针 esp 指向终止位置
- bool defrag_mem；表示是否进行内存整理

满足以下任何一个条件时，评测状态结束：

- 被测程序调用 sys_quit_judge()正常结束
- 被测程序超过时间限制仍未结束
- 被测程序运行错误（例如不预期的页错误、除数为 0 等）
- 被测程序进行了上述参数中规定的以外的系统调用

int sys_accept_enter_judge(envid_t envid, struct JudgeResult *res)

arbiter 调用，来允许一个程序进入评测状态。envid 表示被测程序的进程号，res 传入一个指针，用来接收评测结果，其中有如下项目：

- enum JudgeVerdict verdict；表示评测结果（正常、超时、运行错误、非法系统调用等）
- uint64_t time_cycles；通过 rdtsc 指令读取的运行时间，以时钟周期数计
- uint64_t time_ns；通过 Local APIC 的 TCCR 读取的运行时间，以纳秒计（实践上的分辨率在 1 微秒左右）
- int mem_kb；使用的内存量，以 KB 计（分辨率为页大小 4K）

- struct Trapframe tf; 结束时的 trapframe，以便分析更详细的信息（例如非法系统调用的系统调用号、不预期的中断的中断号）

该系统调用的返回值可能如下：

- -E_INVAL，表示传入的参数非法，例如进程不存在或未在调用 sys_enter_judge 的阻塞状态
- 0，表示正常结束

int sys_quit_judge()

被测程序调用，来结束自己的评测状态。返回值可能为：

- -E_INVAL，表示非法（当前不在评测状态）

注意该系统调用在合法情况下不返回。被测程序在成功调用 sys_quit_judge 后将回到当初调用 sys_enter_judge 的下一条指令继续进行。

真机环境搭建

我们通过 multiboot 的方式，从 U 盘中启动整个系统。启动成功后，方可同在 qemu 中一样，运行相应的应用程序。相较于在 qemu 中进行模拟，在真实机器上，测得的数据更加客观、可信，可以作为论证和研究的依据。

我们通过在 grub 中运行 multiboot 来实现从镜像中引导，具体方式如下：

启动设备后，进入到 grub 中，之后，使用命令：*multiboot /jos-grub /jos-grub* 引导到系统中。

引导到系统中后，即可运用 JOS 中的对应命令进行测试。

Arbiter 评测器的运行

在我们的系统下，设计了 arbiter 程序评测器，作为一款系统上运行的软件来主导评测工作。

从用户运行的角度来说，其使用方式与其他用户程序无差别，具体地，运行评测器时需要使用命令：

arbiter <被测程序> <TL> <ML> <是否整理内存> <其他参数>

其中：

- <被测程序>是指待测定的算法，已经和交互库功能编译生成的可执行文件，如 select，quick 等。
- <TL>是指时间限制，以毫秒（ms）为单位，用于告知系统何时发出时钟中断终止程序运行。
- <ML>是指内存限制，以千字节（kb）为单位。
- <是否整理内存>为 0 或 1，分别表示在运行前不整理、整理内存。整理内存需要耗费一定的时间，但整理之后程序效率更为稳定。
- <其他参数>指传递给交互库的程序，具体形式和内容与交互库有关。

以排序算法为例的测试

在这里，我们选择了几种排序算法进行测试，但这并不意味着我们的系统只能运行排序算法，在解决其他问题时，该系统一样稳定、有效。

在我们的系统下与现有竞赛评测环境 Linux 系统下运行了选择排序（select）与快速排序（quick）算法各 8 次，测定的结果如下：

排序算法	选择排序	快速排序

操作系统	我们的工作	Linux	我们的工作	Linux
第 1 次	52477.86	50359.28	4461.37	3309.27
第 2 次	52478.19	50339.96	4463.03	3301.92
第 3 次	52504.00	50358.29	4462.19	3302.59
第 4 次	52473.39	50246.15	4461.68	3305.13
第 5 次	52494.64	50322.35	4461.62	3302.01
第 6 次	52486.15	50262.92	4462.98	3302.62
第 7 次	52476.70	50231.44	4463.95	3299.42
第 8 次	52468.66	50238.17	4462.03	3309.01
均值	52482.45	50294.82	4462.36	3304.00
标准差	11.01849	51.95194	0.828762	3.303084
标准差/均值	0.021%	0.103%	0.019%	0.100%

（单位为百万时钟周期数，CPU 的智能变频功能被关闭，频率恒定为 3.4GHz，Linux 下测的是用户态时间）

可以看出，在我们的实现当中，降低了 80% 的不稳定性，程序运行的稳定性得到了非常有效的提升，在没有故意干扰（例如键盘等的中断）的情况下，Linux 下运行时间的标准差大约为总时间的 0.1%，我们的系统下这个比值降到了约 0.02%。

测定中断的影响

考虑到中断是影响评测的重要因素，我们选取在不屏蔽中断时拍打键盘的方式来人为制造一些中断。在程序运行时，我们发现拍打键盘会影响到程序的正常运行，降低程序运行效率，具体如下：

排序算法	选择排序		快速排序	
键盘中断	无	允许并持续敲击	无	允许并持续敲击
第 1 次	52477.86	53833.64	4461.37	4582.91
第 2 次	52478.19	53869.78	4463.03	4580.85
第 3 次	52504.00	53830.32	4462.19	4577.52
第 4 次	52473.39	53740.85	4461.68	4579.94
第 5 次	52494.64	53795.52	4461.62	4583.15
第 6 次	52486.15	53762.04	4462.98	4582.19
第 7 次	52476.70	53700.34	4463.95	4578.56
第 8 次	52468.66	53799.22	4462.03	4582.19
均值	52482.45	53791.46	4462.36	4580.92
标准差	11.01849	51.51438	0.828762	1.945422
标准差/均值	0.021%	0.096%	0.019%	0.042%

从中看出，插入中断会导致系统中出现一系列的变化，从而明显地影响到程序效率和对程序效率测定的稳定性。

测定内存整理的影响

而对于 Cache 的稳定性，在这里也进行了同样的分析，通过改变运行时的参数<是否整理内存>，我们可以测定整理内存和不整理内存时程序运行效率的区别，具体如下：

排序算法	选择排序		快速排序	
内存整理	不整理	整理	不整理	整理
第 1 次	52477.86	52479.88	4461.37	4462.191

第 2 次	52478.19	52475.78	4463.03	4462.649
第 3 次	52504.00	52480.03	4462.19	4463.559
第 4 次	52473.39	52488.56	4461.68	4461.651
第 5 次	52494.64	52486.48	4461.62	4462.654
第 6 次	52486.15	52475.86	4462.98	4463.094
第 7 次	52476.70	52510.8	4463.95	4463.716
第 8 次	52468.66	52489.39	4462.03	4464.335
均值	52482.45	52485.85	4462.36	4462.981
标准差	11.01849	10.68601	0.828762	0.81629
标准差/均值	0.021%	0.020%	0.019%	0.018%

可以看出，在整理内存后再运行被测程序虽然效率没有比不整理更快，但稳定性更好，测出的结果更有说服力。

需要注意的是，在测定内存整理的影响时，不宜连续运行两次相同程序，应当在运行后对内存进行一些扰乱，以更好地模拟正常运行时的情况，也明显地区分不整理内存时的情况。

可以看出，中断对程序造成的性能影响明显要高于内存整理，在实验中作为影响稳定性的主要因素。

阶段性总结与未来工作

在本工作中，程序的稳定性和安全性已经通过对 JOS 操作系统在各个方面的修改得到了一定的保证。但本工作未来还有进一步完善的可能性和必要，因时间仓促，在本次报告中不能及时提交，但在未来如有可能，还能继续完善。主要包括以下几个方面：

- 硬件匹配方面

- 网卡

  现在设备使用的网卡相比于之前，有了较大的更新，JOS 系统中驱动不能兼容，需要编写新的驱动。我们编写了 e1000e 网卡驱动，但发现该版本接口的设备也较少。目前该功能已经在 qemu 上正常运行，如能找到合适的设备，则可直接使用。亦可重新编写现有网卡的驱动。

- 硬盘

  现有设备中，没有找到合适的兼容 IDE 的设备，文件系统无法从硬盘中加载，因此，暂时将内存拆出一定的空间作为文件系统。未来计划将其挂载到硬盘上。

- 多 CPU 多进程方面

  - 可行性论证

    在多 CPU 方面，除了去除外部中断的影响、解决 CPU 调度问题外，还需要额外考虑内存和 L3 Cache 方面的竞争与覆盖问题。

  - L3 Cache 管理方案

    L3 Cache 是由多个 CPU 共享的缓存资源，稳定性是希望各 CPU 在 L3 Cache 资源上不会互相冲突，都能稳定运行。一种简单的方法是，对 L3 Cache 上的区域也作出划分，每个 CPU 只能使用被分配到的部分。这样就可以避免 CPU 之间互相干扰。

  - 内存管理方案

    内存和 L3 Cache 有一定的不同之处，第一在于内存共享问题不仅仅有容量问题，还有带宽问题。因此，除了在各程序使用内存上限较大的情况下考虑启用虚拟内存（这就会涉及更复杂的硬盘资源调度），

还需要对内存带宽进行一定的分配，即：每个 CPU 能使用的带宽上限应当进行限制，这一点可能需要通过硬件设计来实现，但一定程度上就违背了多核设计的初衷。如果从软件层面限制，配合时钟，减慢读取速度的效率，应该也是能够实现的。

- 比较论证方面
    - 未来还计划测定在我们的系统下进行评测的稳定性，和在 Linux、Windows 系统上进行评测的稳定性。并进行比较，尝试论证我们设计系统的稳定性比通用操作系统要好。
- 系统调用限制方面
    - 未来计划除了稳定性之外，也从安全性出发，限制程序所能使用的系统调用，对于使用的超出限制的系统调用的程序，予以杀死或鉴别。