西南林業大學
SOUTHWEST FORESTRY UNIVERSITY

硕士学位论文

MASTER THESIS

论文题目 一种高效回收僵尸进程资源的方法

学科专业 林业信息工程

学　　号 20151111002

作者姓名 罗志兵

指导教师 赵家刚 （副教授）

分类号 TP316.85 密级

UDC

学 位 论 文

一种高效回收僵尸进程资源的方法

罗志兵

指导教师 赵家刚 副教授

西南林业大学 昆明

申请学位级别 硕士 学科专业 林业信息工程

提交论文日期 2018/12/26 论文答辩日期 2018/12/19

学位授予单位和日期 西南林业大学

答辩委员会主席 杨毅

AN EFFICIENT METHOD FOR RELEASING RESOURCES FROM A ZOMBIE PROCESS

A Master Thesis Submitted to
Southwest Forestry University

Major: **Forestry information engineering**

Author: **LUO Zhibing**

Advisor: **ZHAO Jiagang (Associate Professor)**

School: **College of Big Data and Intelligence Engineering**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得西南林业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：_____ 日期： 年 月 日

论文使用授权

本学位论文作者完全了解并同意西南林业大学有关保留、使用学位论文的规定，学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权西南林业大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期： 年 月 日

摘　要

资源管理是操作系统内核的一项主要工作，而内存则是计算机系统里不可或缺的硬件资源之一。如何提高内存回收利用率是提高系统整体性能的一个重要途径。在 Linux 系统中，僵尸进程是指已经运行结束，但其所占用资源尚未被系统完全回收的进程。为提高系统对内存的回收利用率，应当尽量缩短僵尸进程在系统内的驻留时间，尽快完成对僵尸进程所占的资源回收利用。Linux 对僵尸进程的处理方式是通过母进程调用 `wait()` 函数来对僵尸进程所占资源进行回收。这种依赖于母进程的等待操作来进行资源回收的方式使得 Linux 在处理僵尸进程的过程中存在着三个问题：一、占用母进程时间；二、资源回收速度过慢；三、系统安全性过低。为了解决这些问题，本文提出了一种无需依赖于母进程的等待操作即可快速回收僵尸进程所占资源的方法，即采用 `do_exit()` 和调度器函数 `schedule()` 合作的方式来替代母进程进行僵尸进程资源回收的方法。

具体而言，Linux 处理僵尸进程效率不高的关键是 Linux 在处理进程退出的流程中把回收僵尸进程资源的工作交给了进程各自的母进程来完成。所以本文对 Linux 处理进程退出的流程模型进行了重新设计，首先把所有能够通过进程自身进行释放的资源统统交给 `exit()` 函数来负责回收处理，然后将剩下的无法由进程自身进行释放的少部分资源交给调度器进程来回收处理。这样做的好处是利用 `exit()` 函数和调度器均会在进程退出时自动介入的特点来达到快速、安全地回收僵尸进程资源的目的。最后，本文在 Linux v1.0 内核的基础上，分别对其内核函数 `do_exit()` 和 `schedule()` 进行了重新实现，并通过实验测试证明，与依赖母进程调用 `wait()` 函数的传统方法相比较，本方法能够大大缩短僵尸进程在系统内的驻留时间，并且能成功避免因无法通过母进程回收僵尸进程的资源而造成的系统资源耗尽的问题。

本文主要研究利用调度器进程来进行僵尸进程资源回收的工作，以达到消除因依赖于母进程的 `wait()` 操作而造成的资源回收利用率低，甚至资源被恶意进程耗尽的问题。本文所做的主要工作如下：

1. 详细回顾了 Linux 进程退出过程的处理机制，以及僵尸进程产生的根本原因，并在此基础上，提出了僵尸进程驻留时间的概念。

2. 基于僵尸进程驻留时间的概念，提出了通过缩短僵尸进程驻留时间来提高系统资源的回收利用率。
3. 重新设计了进程退出的处理流程，并提出了利用 `exit()` 函数和调度器的合作来替代母进程进行僵尸进程资源回收的新方案。这样既可以从根源上缩短僵尸进程的驻留时间，又避免了母进程因等待子进程退出而使自身工作受到耽搁的问题，从而提高了系统资源的回收利用率，同时消除了因依赖母进程的 `wait()` 操作而产生的安全隐患。
4. 对本文所提出的新方案进行编程实现，并通过实验测试，将本文所实现的新方案与 Linux 原来的处理方案进行了对比，证明本文所提方案不仅能有效提高系统对僵尸进程资源回收的效率，还能提高系统的安全性。

关键字： 操作系统，资源回收，Linux，调度器，僵尸进程

ABSTRACT

Resource management is one of the most important jobs of an operating system. Since memory is a crucial and limited resource of a computer system, it is important for the OS kernel to improve reusing of it. In a Linux system, a zombie process is a process that already terminated but still stay in the system with memory resources hold. As a consequence, to improve resource reusing, it is necessary to make the zombie process' residence time as short as possible, that is to release the zombie process' resources as quickly as possible. The way a Linux system handles zombie processes is to pass every zombie process to its parent process and let the parent do the resources releasing jobs. In this way, there will be three problems need to be solved when Linux dealing with an exiting process. Firstly, it takes too much time of a parent process while it's waiting; Secondly, it takes too long to get the resources released from a zombie process. And thirdly, it weakens the system security. To solve the three problems mentioned above, a method is proposed to quickly release a zombie process' resources without the parent waiting. That is to use the `exit()` function and the `schedule()` function together to complete the releasing jobs.

More specifically, the process exiting workflow in Linux will be redesigned and all the releasing jobs that can be done in the exting process's own address space will be handled by the `exit()` function and those very little jobs that left by `exit()` will be completed by the scheduler process. Since the `exit()` function and the `schedule()` function will be involved sequentially whenever there is a process exiting, every zombie process will be reaped more quickly and safely. In the end, based on the Linux v1.0 kernel, this paper rewrites the `do_exit()` and the `schedule()` kernel function and then conducts an experiment to prove that compared to the original parent waiting method, the apparatus provided by this paper can largely shorten the zombie resident time as well as avoid the security problems caused by malicious process.

This paper focuses on using the `schedule()` function to release zombie processes's resources to improve the reusing of resources and to avoid the potential security problems. The work involved are as follows:

1. Firstly, elaborate the Linux's processing of task exiting and the origin of a zombie process and based on these define the zombie resident time.
2. Secondly, propose cutting off the zombie resident time can markly improve the reusing of resources based on the zombie resident time definition.
3. Thirdly, redesign the process exiting workflow and propose a new apparatus that uses the `exit()` function and the `schedule()` function together to complete the jobs of releasing resources from a zombie process instead of relying on a parent's waiting.
4. Fourthly, implement the apparatus in C language and conduct an experiment to prove that the apparatus is more efficient compared to the original method in Linux.

Key words: operating system, resources reusing, Linux, scheduler, zombie process

目录

1 引言 **1**

1.1 研究背景 1

1.1.1 进程管理概述 1

1.1.2 内存管理概述 2

1.1.3 僵尸进程 3

1.2 选题依据 4

1.3 问题引出 5

1.4 主要工作 7

1.5 论文的创新 8

1.6 论文组织结构 8

2 研究现状和相关基础知识 **10**

2.1 研究现状 10

2.2 相关基础知识 12

2.2.1 进程资源 13

2.2.2 进程退出 14

2.2.3 进程切换 15

3 回收僵尸进程资源的新方法 **18**

3.1 Linux 处理进程退出的原流程 18

3.2 解决原处理流程中存在的问题 20

3.2.1 解决僵尸进程存在时间过长的问题 20

3.2.2 解决母进程工作被延迟的问题 22

3.2.3 解决安全问题 24
3.3 新方案设计 26
3.3.1 exit() 函数和 schedule() 函数的设计 26
3.3.2 新方案流程图 28

4 流程实现 **29**
4.1 重写 do_exit() 函数 29
4.1.1 关闭文件 29
4.1.2 释放地址空间 30
4.1.3 释放页表空间 31
4.2 重写 schedule() 函数 33
4.2.1 切换 CPU 的运行环境 35
4.2.2 清理僵尸进程 35
4.2.3 自动识别僵尸进程 36

5 实验 **38**
5.1 处理方式对比 38
5.1.1 工具实现 39
5.1.2 测试结果对比 41
5.2 资源回收的速度对比 42
5.2.1 工具实现 43
5.2.2 实验结果分析对比 45
5.3 安全性对比 49
5.3.1 工具实现 50
5.3.2 输出结果分析对比 50

6 总结与展望 **53**

参考文献 **55**

附录 A 主要代码清单 **57**

A.1 处理方式对比 57
A.1.1 waytest.c 57
A.1.2 waytest1.c 58
A.2 资源回收速度对比 59
A.2.1 ztime.c 59
A.2.2 ztime1.c 60
A.3 安全性对比 61

个人简介 **62**

导师简介 **64**

获得成果目录 **66**

致　谢 **67**

1 引言

1.1 研究背景

1.1.1 进程管理概述

进程是正在执行的程序的一个实例，这个实例包含了程序代码和当前的运行状态信息[1]。进程是操作系统进行资源分配的最基本的单位，是操作系统结构的基础。进程这一概念是在 60 年代初首先由麻省理工学院的 MULTICS 系统和 IBM 公司的 CTSS/360 系统开发团队引入的[2]。现代操作系统的一个最主要功能就是为进程提供服务和管理多进程。因此，进程管理是现代操作系统的重要功能之一，特别是多任务处理的状况下，这是不可或缺的功能。操作系统将资源分配给各个进程，让进程间可以分享与交换信息，保护每个进程拥有的资源，不会被其他进程抢走，以及使进程间能够同步。为了达到这些要求，操作系统为每个进程分配了一个数据结构，用来描述进程的状态，以及进程拥有的资源。操作系统可以透过这个数据结构，来控制每个进程的运作。进程管理主要功能包括：进程创建、进程间同步、进程调度和进程终结几个部分。创建新进程的时候，包括 Linux 在内的现代操作系统都采用了一种通过已有对象创建新对象的技术，即通过已有进程来创建新进程的方式。其中，创建新进程的进程叫做母进程，而被创建的新进程叫子进程。进程调度是为了实现多任务的并发运行而设计的功能，主要用到的技术有时间片轮转法 (RR)、优先级算法等[3]。进程终结主要是指处理进程的退出事宜，包括关闭进程打开的文件、回收 CPU 资源和内存资源等。在多任务操作系统里，一个进程的存在时长往往大于运行时长，进程从被创建到终结，在系统中的存在方式也在不断地变化着。为了方便管理进程，操作系统引入了进程状态这一概念，用于反映进程存在过程的变化[2]。进程状态主要有就绪态、运行态、阻塞态和终

结态，其中终结态也叫僵尸（zombie）态，是指进程运行终结后，但尚未被回收资源时的状态。进程在系统中的任意一时刻，都只能以其中一种状态存在。进程状态被记录在一个叫作进程控制块（PCB）的数据结构内[4]。除此之外，进程控制块还记录了所有其他与进程相关的信息，如进程 id、内存大小等。操作系统就是通过进程控制块来实现对进程的管理和控制的。

1.1.2 内存管理概述

内存管理是对计算机内存资源进行分配和管理的技术，其最主要的目的是如何高效，快速的分配，并且在适当的时候释放和回收内存资源。内存管理在多任务系统中是必不可少的功能，并且内存管理的性能是影响系统整体响应能力和安全性的最重要因素之一。提高内存管理性能的方法主要围绕两个方面来实现：即少分配和快释放。少分配是指在该进程分配内存空间的时候，要做到能少分配尽量少分配，能不分配的时候尽量不分配，如写时复制、延迟复制、按需分配等技术[5]。快释放是指，一旦出现不再被使用到的内存对象时，应尽早释放，回收其所占的内存。早期的系统在内存回收上主要依赖于垃圾回收法。垃圾回收也叫自动内存管理，是指通过专门的程序来回收内存中那些不再被使用到的对象所占用的内存资源，最早是由 John McCarthy 于 1959 年提出的[6]。垃圾回收主要依赖于特定的算法来识别内存中哪些对象是不会再被使用到的，然后把这些对象的内存资源进行回收。用于垃圾回收的算法主要有标记清除法、三色标记法、复制回收法、和分代回收法等[7]。垃圾回收法减轻了程序员的工作量，使得程序员不再需要为如何释放所申请的每一个内存对象而费脑筋，对于技术不熟练的程序员来说更是福音，所以从这个角度来看，垃圾回收确实能在一定的程度上减少因程序员技术水平参差不齐而产生的程序错误或漏洞，如悬空指针、内存重复释放等漏洞。然而，这些都是需要付出巨大的时间和空间作为代价的。如标记清除法需要花费大量的 CPU 时间来对差不多整个内存进行两次遍历标记，才能找到所有需要回收的内存对象[8]。三色标记法较之标记清除法虽然减少了二次遍历时间，却增加了三个区域间的对象的移动时间[9]。复制回收法虽然将内存分为了相等的两个区域，在遍历的时候只需对其中一个内存区域进行遍历，节省了不少的时间，然而，却是

需要拿出一半的内存作为复制交换用。分代回收法是按照对象在内存中的存活时间差异将对象分为年轻代分区和年老代分区，年轻代分区会较为频密进行较为激进垃圾回收行为，而年老代则较少进行垃圾回收的行为。通过分代，使得存活在局限域，小容量，寿命短的对象会被快速回收；而存活在全局域，大容量，寿命长的对象就较少被回收行为处理干扰。这在很大程度上减少了垃圾回收的次数，然而，却也带来了新的问题，如算法在对年轻代进行垃圾回收的时候必须在没有遍历老年代分区的情况下从年轻代分区里识别出哪些对象是会被老年代对象所用到的，而把它们移动到老年代分区里，这就增加了算法本身的难度。

因此，不论是哪一种垃圾回收算法，其本身运行起来都是要耗费巨大的时间和空间代价的，在大型的程序里，垃圾回收所花费的时间甚至占到了程序运行时间的 25% ~ 40%[10]。基于以上这些原因，现代依赖于垃圾回收法的系统已经不多见了，除了在一些高级语言中会用到外，最常见的例子就是应用于 Java 虚拟机里。而如今，内存回收则成了操作系统内存管理的一部分。以 Linux 和 Unix 为代表的 POSIX 操作系统在回收进程内存时，采取的做法主要是把负责为进程回收内存资源的工作交给每个进程的母进程来完成。概括来讲，母进程回收子进程资源的工作流程大致是子进程退出时通过调用 `exit()` 函数来通知母进程准备为其回收资源，而母进程接收到通知后，通过调用 `wait()` 函数从子进程的进程控制块里直接找到子进程的各项内存资源，并将其一一回收释放。这种做法相比垃圾回收的好处是：首先，母进程不需要花大量的时间对内存进行遍历和对象标记；其次，在清除的时候，也不需要额外的内存作为暂存或交换空间。因此，较垃圾回收法节省了大量的 CPU 时间和内存资源。

1.1.3 僵尸进程

僵尸进程是指已经运行完毕，但因尚未被母进程回收资源而继续存在系统内的进程[11]。Linux/Unix 在处理进程退出时，资源回收采取的是分两步走的方式，即首先由子进程（要退出的进程）通过调用 `do_exit()` 函数来通知母进程，在母进程接收到通知后调用 `wait()` 函数来为其完成资源回收工作。在这两步走的方式下，使得进程自身任务执行完毕后到彻底退出系统前，即被 `wait()` 前的这段时

间内，进程以一特殊的状态存在，即僵尸（zombie）态。处在僵尸态的进程就叫僵尸进程，跟僵尸这个词自身的定义一样，僵尸进程意味着一个进程不再存活或者说是不再占用 CPU 资源，但是其在内存中的实体仍然存在[12]。从僵尸进程的产生来看，僵尸进程是系统里正常存在的一种进程，是不可避免的。然而，僵尸进程虽然已经不再运行了，但是进程本身还是消耗系统资源的，如页表空间、地址空间、进程 ID、进程控制块等。如果进程的母进程由于某种原因，如程序本身设计的缺陷导致母进程无法及时（甚至根本无法）为已成为僵尸进程的子进程进行资源回收的话，将会导致大量的僵尸进程长时间存在于系统内，则会造成系统资源的浪费甚至造成更严重的后果，如系统无法再继续创建新进程。因此，为了提高系统的整体性能，应该尽量缩短僵尸进程在系统内的存在时间。本文关注的就是如何缩短僵尸进程在系统里的存在时间。具体来说，本文要解决的问题是：在不改变处理进程退出事宜必须分两步走的前提下，通过取消依赖于母进程的等待来进行资源回收，并通过重组和优化操作系统处理进程退出的各项工作来缩短僵尸进程在系统内的驻留时间，从而做到既可以快速回收利用每一个僵尸进程所占的资源，又可以从根源上有效防止恶意进程通过大量产生僵尸进程来耗光系统资源的情况的出现，使系统的整体性能得到提高。

1.2 选题依据

僵尸进程是已经运行完毕但尚未被回收资源的进程，僵尸进程的存在是一种正常现象，是现行操作系统处理进程退出时不可避免出现的情况。然而，僵尸进程的存在却是毫无意义的，占用系统资源的，并且，大量的僵尸进程还会造成系统资源的耗光，导致无法系统继续创建新进程，所以，僵尸进程必须被尽快清除。目前，关于如何快速回收僵尸进程所占的资源的研究还比较少见。已有的相关研究也主要是关于如何检测到系统内存在的僵尸，以及如何防止恶意进程产生大量的僵尸进程。这些研究都需要依赖于复杂的算法来解决一次性问题，如每运行一次算法，只能清除当前存在的僵尸进程，所以系统需要定时运行这些算法来定期对大批量的僵尸进程进行清理。这些做法不仅时间代价高，而且没有能从根源上减少僵尸进程，包括正常进程产生的和恶意进程产生的僵尸进程，在系统内的驻

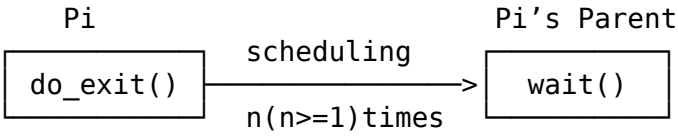

图 1-1 n 次进程切换
Fig. 1-1 N times of context switch

留时间。本文从僵尸进程产生的根源出发，通过分析僵尸进程从产生到被系统清除的这个过程中的各种工作细节，找到流程中各种工作之间的联系，利用这些联系对这些工作进行重组和优化，并将优化后的各项工作进行重新分配，最后实现了一种不需要依赖母进程进行僵尸进程资源回收的方法，从而可以缩短僵尸进程在系统内的驻留时间。

1.3 问题引出

僵尸进程是在Linux/Unix系统在处理进程退出事务的过程中产生的。以Linux为例，在处理进程退出时采用了分两个步骤完成的方式：首先由子进程通过调用 `do_exit()` 函数来通知母进程准备为其回收资源。而母进程在接收到通知后,通过调用 `wait()` 函数来完成资源回收。在这个过程中,从子进程到母进程的切换,实际上是需要经过调度器调度器进行若干次进程切换才能真正轮到母进程运行 `wait()` 函数这一步，如图 1-1 所示。进程 $\mathrm{P_i}$ 通过调用 `do_exit()` 函数交出 CPU 使用权后，上下文切换到调度器，调度器经过 n 次（$n \geq 1$）调度后，轮到了 $\mathrm{P_i}$ 的母进程运行 `wait()` 函数为子进程（$\mathrm{P_i}$）回收各种内存资源。

这个过程存在的问题是：

1. 容易出现僵尸进程在系统里停留的时间过长而造成内存资源的无谓耗费；
2. 如果遇到恶意程序产生大量的僵尸进程，会出现因系统资源被耗光而无法继续创建任何进程的问题；
3. 由于母进程需要等待子进程的退出，以便为其回收资源，所以，等待的过程会耽误母进程自身的工作执行。

在这个过程中，我们假设 Z_t 为僵尸进程存在的时长，那么，Z_t 的大小主要取决于就绪进程队列（ready queue）里排在 $\mathrm{P_i}$ 的母进程前面的进程数量的多少，也

就是说，在 P_i 调用 `exit()` 之后，需要经过多少次的进程切换才轮到其母进程运行。在理想状态下，P_i 退出后，如果就绪队列里只剩下 P_i 的母进程这一个进程，那么，只需要进行一次进程切换就可以了，所以 Z_tt 的值就比较小。如图 1-2 所示。

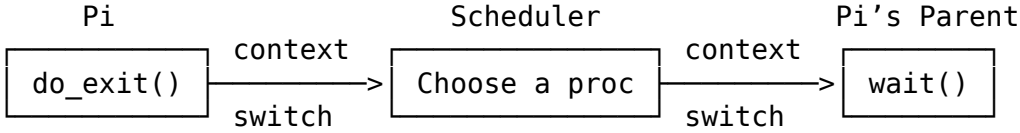

图 1-2　理想状况下，只需一次切换
Fig. 1-2　One context switch is needed

$Z_t = n \times (2C + S)$，其中，n 为进程切换的次数；C 为一次上下文切换的时间；　S 为调度器选中一个进程的时间。在这种理想状态下，由于就绪队列里只有 P_i 的母进程，所以 $n = 1$，$Z_t = 2C + S$;

在多数情况下，就绪队列里不会只有一个 P_i 的母进程，而是有多个进程，如图 1-3 所示。

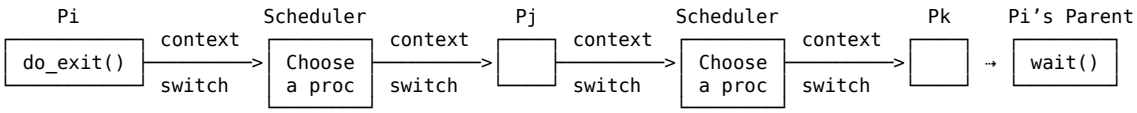

图 1-3　队列里有多个进程的情形
Fig. 1-3　Many processes in queue

这个时候，就得考虑有多少个进程排在 P_i 的母进程的前面了。如果有 $n-1$ 个进程排在 P_i 的母进程前面的话，则需要经过 n 次的进程切换才能为 P_i 回收内存资源。这时候 $Z_t = n \times (2C + S) + (n - 1) \times qum$，其中，$qum$ 为每个时间片的长度。在 Round-Robin 调度算法里，qum 是个定值；在基于优先级调度算法里，qum 则与进程的优先级相关。通常优先级越高，qum 越大；反之则越小。于是，在基于优先级的调度算法下，如果母进程的优先级非常低的话，僵尸进程在系统内的驻留时间 (Z_t) 则会更长。

如果遇上恶意程序通过母进程产生大量的僵尸进程后，却故意一直不对任何子进程进行 `wait()` 操作，并且母进程一直没有退出系统的情况，即使母进程被调度了无数次，系统仍然无法回收这些僵尸进程，所以这些僵尸进程会一直存在。这种情况最直接的后果就是会导致系统由于资源耗光而无法创建任何进程，无法执行任何新任务。

为了解决以上所列举的问题，本文提出了一种无需依赖于母进程的等待即可快速回收僵尸进程资源的方法。

1.4 主要工作

本文提出了一种不需要母进程等待的高效回收僵尸进程所占资源的方法。该方法首先提取 Linux 处理进程退出的的流程并分析其中存在的问题，然后通过对流程中的各项工作进行详细分析，并找到各项工作间的联系后，对这个处理进程退出的流程进行了重新设计。在重新设计的时候，采取了把资源回收的大部分工作交给进程自身来完成，而把少部分清除和回收工作交给调度器来完成的方案，以达到在无需依赖于母进程的等待操作来完成资源回收的同时又能在很大程度上缩短每一个僵尸进程在系统内的驻留时间，从而实现了对僵尸进程资源的快速回收利用。最后本文对重新设计后的流程进行详细的编程实现并以实验测试的方式验证本文所实现方法的正确性和有效性。本文所做的工作内容主要如下：

- 本文从僵尸进程的产生根本原因出发，首先对 Linux 系统处理进程退出的完整过程过程进行详细的分析和研究，然后建立了 Linux 系统处理进程退出的完整工作流程模型，并总结出该流程模型中存在着三个问题：一、母进程需要花太多的时间来等待子进程的退出；二、回收僵尸进程资源的速度过慢；三、系统安全过低以及存在问题的根本原因是 Linux 系统采用了一种依赖于母进程的等待操作来完成僵尸进程资源回收的方法。
- 为了解决以上提出的三个问题，本文提出了一种无需母进程等待的高效回收僵尸进程资源的方法。首先对 Linux 处理进程退出的流程模型进行了重新设计，在重新设计的时候采用了 `exit()` 和 `schduele()` 共同协作的新方案来替代母进程进行进程资源回收。具体来说，本文将 `wait()` 函数的工作进行重新分配，并且将其中所有能够在进程自身的地址空间内执行的工作统统交给 `exit()` 来完成，而将剩余的无法在进程自身的地址空间内执行的工作交给 `schedule()` 函数来完成。这样做的好处是利用 `exit()` 函数和调度器均会在进程退出时自动介入的特点来达到无需占用母进程的等待时间，同时又能快速、安全地回收僵尸进程资源的目的。

- 本文实现了一种无需母进程等待的高效回收僵尸进程的方法,用于解决 Linux 类系统在回收僵尸进程资源的过程中潜在的问题。同时，通过实验，分别从三个方面验证了本文所实现的方法与 Linux 原来的处理方法相比，具有三个优点，即无需占用母进程的等待时间，同时能够更快、更安全地回收僵尸进程所占的资源。证明了本文实现方法确实能够正确有效地解决 Linux 处理僵尸进程的流程中存在的三个问题。

1.5 论文的创新

- 首次提出了通过无需经过母进程等待的快速回收僵尸进程所占资源的方法，即无需母进程 `wait()` 的方法，可有效提高僵尸进程资源的回收利用率并能从根源上避免恶意进程通过产生大量的僵尸进程来耗光系统资源的问题，有效提高系统的安全性。
- 本文首次提出了通过 `exit()` 函数和调度器进程之间的合作方式来代替母进程回收僵尸进程资源的工作。通过将母进程 `wait()` 函数所做的大部分工作交给 `exit()` 函数来完成，而只把其中少部分无法由 `exit()` 完成的工作交由调度器来完成，确保在基本不改变系统响应性能的前提下实现僵尸进程资源的快速回收。

1.6 论文组织结构

第一章 引言，介绍了论文的研究背景和研究现状，引出需要解决的问题，并提出了解决方案，最后总结了主要工作以及创新性；

第二章 研究现状和相关基础知识，主要介绍了跟论文相关的课题的已有研究现状以及进程的切换过程、调度器的工作方式、进程所占的内存资源有哪些以及进程退出的流程等相关的基础知识；

第三章 回收僵尸进程资源的新方法，本文首先提取了 Linux 系统处理进程退出的原流程并对原流程中的各项工作进行详细分析，并指出了原流程中潜在的问题。然后，重新设计了 Linux 处理进程退出的新流程，并提出了使用 `exit()`

函数和调度器合作的方式代替母进程回收僵尸进程资源的新方法，最后对新方法进行了可行性分析与详细的方案设计；

第四章 流程实现，对本文所提出的新方法进行详细设计与编程实现；

第五章 实验测试，对本文所实现的新方法进行了实验测试，并从三个方面对本文所实现方法的正确性和和有效性进行了实验评估；

第六章 总结与展望，总结了本文所做的工作，并对未来的研究进行了展望。

2 研究现状和相关基础知识

2.1 研究现状

Linux 或类 Unix 系统在处理进程退出的过程中都会产生僵尸进程。僵尸进程本身对系统并没有什么害处，但由于僵尸进程是毫无意义地消耗系统资源的，大量的僵尸进程的长时间停留更是会造成资源的无谓耗光，于是，僵尸进程因该尽快被系统清除处理。现有的关于僵尸进程的处理方法比较有限，主要可分为以下几类:

第一类是对在母进程退出前的僵尸进程的处理。如 Love 提出的子进程在退出的时候通过发信号的方式请求母进程对僵尸进程进行处理的方法[13]。该方法只能适用于子进程退出时，母进程已经执行完了自身的任务，但尚未退出而是处在等待子进程退出的状态，这时候母进程在接收到子进程的退出通知后便能即刻进行僵尸进程资源回收处理。该方法存在的不足之处是: 容易给母进程带来不便, 如果在进程给母进程发通知的时候, 母进程还在执行自身任务, 这种时候, 母进程只能选择暂停自身任务的运行转而处理僵尸进程，或者是选择忽略子进程发来的请求而继续执行自身的任务。另外一种方法是：子进程在退出后一直以僵尸态存在直到自己的母进程自身任务执行完毕后在退出时，统一对所有的子僵尸进程进行统一回收处理[14]。这种方法虽然不会对母进程的自身工作的执行造成任何影响，然而却使得所有的子僵尸进程都必须等待母进程退出后才能被处理，导致这些僵尸进程在系统内的驻留时间过长而造成资源的无畏浪费。

第二类是对母进程退出后的僵尸进程的处理。一个进程在退出的时候，如果它还存在尚未运行结束的子进程，母进程就会解除与这些子进程的母子关系，并把这些子进程交给系统的 `init` 进程处理。对于 `init` 进程来说, 这些被母进程 “抛弃” 后的子进程都是孤儿进程[15]。对这一类由孤儿进程发展来的僵尸进程的处理

方法主要有：统一由新的母进程 init 进行回收处理[16]。这种方法的不足之处是 init 进程需要处理来自很多不同母进程的子进程，这使得 init 进程的负担过大而导致响 init 对僵尸进程的处理响应速度过慢。为了解决这种问题，Kroah 提出了对孤儿进程进行标记，并且定期启用系统的空闲进程（swapper 进程）对进程表进程进行扫描找到这些被标记过的僵尸进程进行资源的回收处理[17-18]。该方法通过 swapper 的处理来解决了 init 进程负担过大的问题，然而却需要花时间定期对整个进程表进行扫描寻找被标记过的僵尸进程，于是 Luke 进一步提出了通过把所有的被标记过的僵尸进程统统放到一个专用的链表里，每次启用 swapper 进程时只需按顺序一一回收处理链表里的僵尸进程，因此，省去大量遍历进程表的时间[19]。

第三类是对无法通过母进程回收处理的僵尸进程的处理。由于编写用户软件的程序员技术水平参差不齐，用户程序可能因编程不当或错误，而导致母进程在创建了子进程后忘记为子进程回收资源，或者是无法成功完成回收子进程资源的工作。为了解决这些问题，Roger 提出了通过给母进程绑定一个专门代理僵尸进程资源回收的线程来帮助母进程进行资源回收以及 Daniel 提出的实现一种机制，用来允许母进程在回收僵尸进程资源失败后可以自动创建一个代理线程来处理僵尸进程[20-21]。这些通过代理线程来处理僵尸进程的方法的不足之处在于创建代理线程本身需要花费时间，导致僵尸进程没有尽快被清除，其次把创建代理进程的工作交给普通用户进程容易导致系统的安全性下降。

第四类是对恶意进程产生的大量僵尸进程的处理。如果遇到了恶意用户进程，创建了大量的子进程，这些子进程并没有执行任何的任务而是立即退出，这时候系统会在短期内产生大量的僵尸进程，同时，因为母进程故意不对任何的僵尸进程进行处理并且母进程一直没有退出系统，就会导致这大批量的僵尸进程一直驻留系统内，浪费大量的资源甚至出现因资源耗光而无法继续创建新进程执行新任务的情况。已有的处理这种情况的方法非常有限，而且代价非常高，如某些文献提出的通过找到产生大批量恶意僵尸进程的母进程并把其杀死来强行让其处理僵尸进程或者通过关闭整个系统电源的方法来清除一切进程，包括所有的僵尸进程[12]。这样的处理方法使得一切正在运行的进程，包括那些正在进行重要计算

的进程都被迫关闭。显然这种方法代价太高，实在意义不大。

综述所述，现有的关于如何高效处理僵尸进程的解决方法都普遍存在不足，没有能同时做到以下几点：

1. 不耽误母进程自身任务的执行；
2. 尽量缩短僵尸进程在系统内的驻留时间，即将 Z_tt 缩短到最小；
3. 防止恶意进程产生大量僵尸进程而导致系统资源被耗光；
4. 不显著增加系统处理僵尸进程的代价。

为了弥补现有方法的不足，做到能同时满足以上所提到的各项要求，本文从僵尸进程产生以及消失的根源出发，找到了一种利用 `exit()` 和调度器来代替母进程调用 `wait()` 函数完成僵尸进程资源的回收工作，做到了（1）点；把僵尸进程资源回收的工作交给调度器进程来处理，是利用了调度器在每一次进程切换的时候都会自动介入，而进程的退出必定会发生进程切换这一点，如此一来，进程在成为僵尸进程后到调度器切换到任何别的进程之前就已经被回收清除了，也就是将 Z_tt 的值缩短到最小，做到了（2）这一点；同时，由于当已退出的进程把 CPU 资源交给调度器后，所产生的僵尸进程就自动被调度器回收清除了，因此，进程一旦退出了就被调度器自动清除而根本上不会产生大量僵尸的情况，于是做到了（3）这一点。最后，由于本文是通过对现有的 `do_exit()` 函数和 `wait()` 函数所做的工作进行详细分析后进行重组和优化，并把优化后的工作重新分配给 `exit()` 和调度器来完成，而在分配工作的时候，本文尽量把大部分的回收工作交给 `exit()` 函数来完成，而仅仅只把最少部分无法由 `exit()` 完成的工作交由调度器来完成。因此，调度器进程基本不会因为工作强度增加而导致进程切换的时间增加而降低整个系统的响应速度，即做到了（4）这一点。

2.2 相关基础知识

本文主要研究的是如何利用调度器替代母进程进行僵尸进程资源回收，以此达到缩小僵尸进程在系统内的驻留时间从而提高系统对僵尸进程所占资源的回收利用率。为了完成这个目标，本文对所有的相关基础知识和理论进行了详细研究，现将所涉及的理论知识作如下简介。

2.2.1 进程资源

进程是计算机进行资源分配的一个最基本单位。一个进程在系统内的存活，从被创建到完全退出系统的这个生命过程内都需要占用一定的系统资源，如 CPU、内存、文件等。其中，CPU 是进程任务的执行者，是进程存活的根本。文件则是用以存储进程的输入输出。而内存资源则是用以存储进程自身的程序代码、运行状态信息等。一个进程即使不再存活，如僵尸进程，仍然会占用一定的内存资源，如进程控制块、内核栈等。

2.2.1.1 进程控制块

进程控制块也叫任务控制块[3]，是系统为了方便管理进程而为每一个进程创建的一个数据结构，专门用于存放与进程相关的所有信息，如进程号（PID）、文件信息、页表指针、内存空间大小、内核栈指针、进程状态等[22]。系统对进程的所有操作，基本上都需要访问进程控制块，如给进程发信号时需要访问进程 PID；给进程分配或回收地址时，需要访问进程的页表指针；为进程处理中断时需要访问内核栈指针；进程切换时则需要访问进程的状态信息等一系列对进程的操作都需要访问进程控制块里的相应内容。

2.2.1.2 地址空间与页表

地址空间是一个进程可用的地址范围大小，对于程序员来说，这个大小是整个 CPU 的可寻址范围，即所有的地址对于程序员来说都是可用的，程序员不用考虑实际物理内存的大小，也不用考虑所使用的每一个地址是否会跟别的程序里的地址重合等问题。因为程序里每一个地址对操作系统来说都只不过是虚拟地址，而并非真正的物理地址。这些虚拟地址的集合就叫虚拟地址空间，以 32 位系统为例，每个进程的虚拟地址空间可达 4G[23]。在程序加载的时候，虚拟地址空间里的地址会被操作系统按需加载到物理内存的不同页面中，至于具体被加载哪些个物理页面，则是随机的。因此，为了方便找到每一个虚拟地址所对应的物理内存地址，需要对每一个虚拟地址与物理地址的对应关系进行记录。页表就是一个专门用于记录虚拟地址与物理地址对应关系的数据结构，通常是由若干个物理页面组

成[24]。每一个进程都有自己独立的页表，并且进程只能访问记录在自己页表内的物理页面，而无法侵犯别的进程地址空间。

2.2.1.3 内核栈

进程在运行的过程中通常会频繁地请求内核服务，如系统调用或中断，因此，为了减少处理中断的时间，现代操作系统多把内核代码映射到每一个进程的地址空间中，以避免处理进程的系统调用和中断请求时频繁发生进程切换。也就是说，系统调用和中断处理这些内核函数是运行在每一个用户进程的地址空间中，为了避免与用户代码共用一个用户栈而带来的安全问题，系统给每一个进程分配了一个内核栈[25-26]，专门用于进程执行内核代码时的函数调用，并且规定用户代码不可访问内核栈。

2.2.2 进程退出

正常情况下，当一个进程完成自己的使命，即运行结束后就会被终结。然而，除了正常运行结束外，还有很多其他的原因会导致一个进程的终结，如进程自身的程序错误、进程试图访问非法地址、死锁、内存资源不够等原因[27]。不管是哪一种原因引起的进程终结，最终的结果都会导致进程退出。当一个进程退出时，它并非立马就能消失在系统中，相反，进程的退出是一个复杂的过程，由于进程自身所占据着一整套的资源，在进程运行结束后，系统必须对这些资源进行一一回收利用，所以进程退出的过程，实际上主要是系统对进行资源回收的一个过程，在系统完成对这一整套的系统资源的全部回收工作之后，进程才会彻底从系统里消失。Linux 系统对退出进程的资源回收工作主要分为两部分，分别由 `do_exit()` 函数和 `wait()` 函数来完成。其中，`do_exit()` 函数由正在退出的进程自身调用，而 `wait()` 函数则由对应的母进程负责调用。

2.2.2.1 `do_exit()` 函数

在 Linux 系统中，正在退出的进程通过 `do_exit()` 函数来完成一部分可在当前进程的运行空间内完成的清理工作，并让出 CPU 使用权。概括来讲，`do_exit()`

函数主要负责完成以下工作：

1. 关闭所有被进程打开的文件；
2. 通知母进程准备为自己完成资源回收工作；
3. 把尚未运行结束的子进程交给 `init` 进程，即重新安排这些子进程的母子关系；
4. 把进程自身的状态标记为 zombie 态；
5. 交出 CPU 使用权；

2.2.2.2 `wait()` 函数

进程一旦把自身的状态设置为 `zombie` 态，并让出 CPU 使用权后，进程便成为了僵尸进程，无法再执行任何的代码。而对僵尸进程所保留的一切资源的回收工作就只能交给其他的进程来执行。在 Linux 系统下，这个“其他的进程”就是该进程的母进程，母进程则是通过调用 `wait()` 函数来完成工作。概括来讲，`wait()` 函数的主要工作如下：

1. 释放进程所占的内核栈；
2. 释放进程的地址空间和页表空间；
3. 清空进程表里该进程控制块所占的位置；
4. 向系统交回进程 PID 号；

2.2.3 进程切换

进程切换是指操作系统暂停当前进程的运行，按照规定的算法从就绪进程队列里选取一个进程，让其运行的过程。进程切换使得共存于内存中的多个进程能共享 CPU 资源，是操作系统实现多任务并发执行的手段[3]。进程切换的过程包括进程的上下文切换以及选中下一个能运行的进程。

2.2.3.1 上下文切换

进程的上下文是进程运行状态的静态描述（快照）。具体说，进程的上下文包括一切与进程运行相关的状态信息，如 CPU 寄存器的值、程序计数器指针的

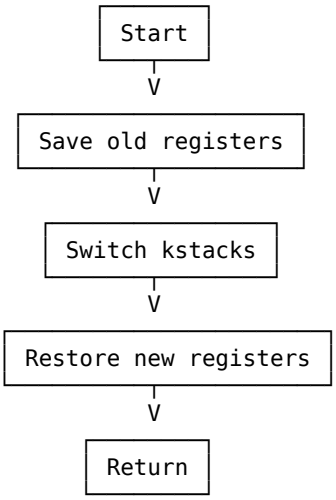

图 2-1　`switch_to()` 函数工作流程
Fig. 2-1　The workflow of `switch_to()`

值以及堆栈指针的值等[28]。在上下文切换的时候，内核所做的工作主要是首先把当前运行进程的上下文保存到该进程的内核栈中，然后切换到下一个要运行的进程的内核栈，最后把该栈里的上下文内容加载到 CPU 中，即完成了上下文切换。Linux 系统里负责上下文切换的函数是 `switch_to()`，该函数的主要工作流程如图 2-1 所示。

2.2.3.2　**进程调度器**（Schedule）

进程调度器（Schedule）是内核为了在多进程中合理分配使用 CPU 资源而专门创建的一个进程。进程调度器的工作原理是按照一定的调度算法每次从就绪队列里选取一个符合条件的进程，并把 CPU 资源交给这个进程执行[29]。如在分时调度算法下，调度器从就绪队列里按顺序每次选取一个进程，让它运行一个时间片段后，再按顺序选取下一个进程并让其运行相同长度的时间片段，如此周而复始。进程调度器在进程切换过程中的作用是负责决定接下来应该轮到哪一个进程运行然后进行上下文切换。然而，调度器本身也是一个进程，也有自己的上下文，于是，当一个进程被剥夺 CPU 使用权的时候，会首先从这个进程的上下文切换到调度器的上下文，然后调度器才能运行起来。所以，从一个进程（如进程 A）切换到下一个进程（如进程 B）的完整过程大致如图 2-2 所示。

从图 2-2 中可以看出，一次完整的进程切换的时长包含了两次的上下文切换时

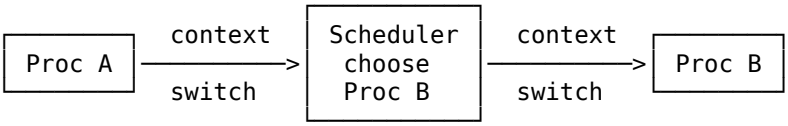

图 2-2　进程切换
Fig. 2-2　Context switch

长和调度器选中一个进程的时长。如果以 T_{psw} 表示一次进程切换的时长，C 表示一次上下文切换的时长，S 表示调度器选中一个进程的时长，则 $T_{psw} = 2 * C + S$。其中 S 主要取决于调度器所使用的算法算法，不同的调度器在做出决策上所花费的时间不一样，如 $O(1)$ 调度器的时间复杂度是 $O(1)$[30]，而 CFS 调度器的时间复杂度则是 $O(\log n)$[31]。

3 回收僵尸进程资源的新方法

进程在系统中的存活需要依赖于各种系统资源，如 CPU 资源、内存资源等。在进程运行终结要退出时，操作系统需要为其回收所占用的各项资源，以实现对资源的回收利用。所以，操作系统对进程退出处理的流程实际上可以看成是系统对退出进程进行资源回收的一个过程。由于 Linux 处理进程退出的流程中存在着一些前文曾提到过问题，所以，本文将通过重新设计这个流程来解决这些潜在的问题，以提高系统的整体性能。接下来，本文首先建立 Linux 进程退出处理的流程模型，然后对这个流程进行详细分析找出其中所存在的问题，再一一对这些问题提出相应的解决方法，最后基于这些解决方法设计出一个能同时解决所有潜在问题的流程模型。

3.1 Linux 处理进程退出的原流程

在 Linux 系统中，进程退出的处理过程分为两个步骤，首先通过子进程（要退出的进程）自身调用 `do_exit()` 函数完成一部分工作，如关闭进程所打开的所有文件、通知母进程为准备自己回收资源、释放 CPU 资源等工作。其次，通过母进程调用 `wait()` 函数来为进程（已经成为僵尸进程的子进程）进行各种资源回收，如内核栈、页表等资源。Linux 对进程退出处理的原流程可用图 3-1 表示。

在这个处理流程模型下，当一个进程（P_k）运行结束，也就是调用 `exit()` 函数后，进程 P_k 就变成了僵尸进程，由于已成为僵尸进程的 P_k 进程不可能再占用任何 CPU 资源，所以它所占的各项资源需要依赖于自己的母进程来为其释放。而当上下文从 P_k 进程切换到调度器进程后，调度器进程会按照既定的调度算法从就绪队列里选中下一个能够运行的进程，如果这”下一个“进程恰巧是 P_k 的母进程的话，则僵尸进程很快就会被 `wait()` 函数所清除，如果不是的话，则需要经过若干

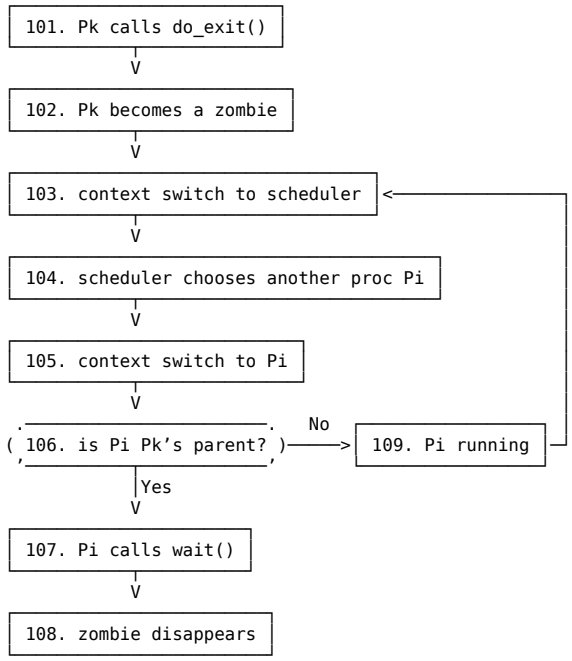

图 3-1　Linux 处理进程退出事务的原流程
Fig. 3-1　Process exit procedure in Linux

次的进程切换后调度器才能选中 P_k 的母进程。所以，步骤（103 ~ 107）只是一个理想状态下的情形，也就是就绪队列里只有 P_k 的母进程唯一一个进程; 在非理想状态下，就绪队列里有 n 个进程排在这个母进程的前面或者优先级别比它高，这个流程就需要在步骤（103 ~ 109）处重复 n 次后才能选中 P_k 的母进程。在 P_k 的母进程被选中后，如果母进程立马调用 `wait()` 函数并且函数成功返回的话，僵尸进程便立即消失在系统中。但是，如果 n 的值很大的话，那么僵尸进程可能等待很长的时间才能被母进程处理，又或者是，子进程退出的时候，母进程尚未调用 `wait()` 函数或者母进程一直忘记调用 `wait()` 函数，那么这些已经成为僵尸进程的子进程将如何得到处理？因此，以上 Linux 处理进程退出的流程实际存在着一些隐性问题，如僵尸进程等待时间过长或者僵尸进程可能无法得到处理等。概括来说，这个流程主要存在以下几个问题：

1. 首先，存在着因僵尸进程存在时间过长而降低系统对资源的回收利用率；
2. 其次，由于母进程需要等待子进程的退出从而影响母进程自身任务的执行；
3. 最后，依赖于进程各自的母进程来完成处理工作存在着安全隐患，可能遭到恶意用户进程的攻击。

3.2 解决原处理流程中存在的问题

在上一节，本文通过分析，总结出了 Linux 在处理进程退出的流程中存在的三个问题：（1）僵尸进程存在时间过长；（2）母进程因等待子进程的退出而耽误自身工作的执行；（3）安全问题。接下来，本文将致力于如何高效地解决这三个问题。

3.2.1 解决僵尸进程存在时间过长的问题

由于 Linux 采用的是把进程退出的处理工作分散到两个进程中来完成（进程本身与母进程），这就导致了在这个过程中必然会发生进程切换，而且至少发生一次（从进程到母进程的切换）。并且，每一次进程切换必然会有调度器的介入，而调度器却是根据系统所规定的调度算法从就绪队列里挑选一个符合条件的进程来作为下一个能运行的进程，如在分时算法里，调度器会选择下一个进程，而在优先级算法里，调度器则会选择就绪队列里优先级最高的进程。所以，在调度器选择下一个的进程时候，它未必会选中这个母进程。当然，如果就绪队列里只剩唯一一个进程，也就是这个母进程，这时候调度器当然是选中了这个母进程。母进程被选中后便能通过调用 `wait()` 函数来处理进程退出的任务了，于是僵尸进程便能很快得到清除，也就是图中的（102 ~ 108）的这个过程。这种情况下，僵尸进程在系统内的存在时间可表示为：

$$Z_t = 2C + S \tag{3.1}$$

其中 C 为一次上下文切换的时间，S 为调度器选中下一个进程的时间；然而，在实际情况中，就绪队列里只有该母进程的情况很少存在，大多时候却是，有很多进程排在这个母进程的前面，或者，在就绪队列里有许多优先级别较高的进程。这个情况下，调度器需要经过多次的进程切换后才能选中这个母进程。举个例子，如果有 n 个进程排在这个母进程的前面或者说有 n 个较高优先级的进程的话，则需要等待 n 次的进程切换，即需重复步骤（103 ~ 109）n 次才能轮到这个母进程运行 `wait()` 函数。而每增加一次进程切换，则需要增加一次进程切换的时间以及一次进程运行的时间，直接导致了这种情况较之只有一个进程的情况，僵尸进程在

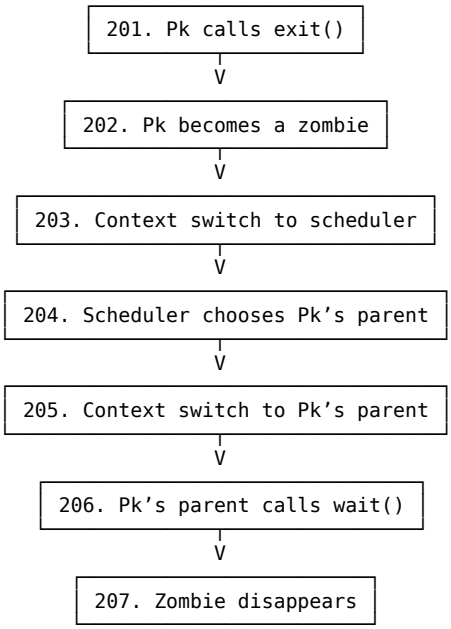

图 3-2　进程退出的过程
Fig. 3-2　Process exit procedure

系统内的存在时间大大增加，这个时间可表示为：

$$Z_t = 2C + S + (2C + S + qum) * n \tag{3.2}$$

其中，n 表示排在这个母进程前面的进程数量，qum 表示一个进程运行的时间片长度。在大多数情况下，一个进程运行结束后所变成的僵尸进程，往往要经过很长等待的时间，如表达式 3.2 所示，才能被母进程回收资源。这就导致了系统无法对僵尸进程所占的资源进行快速回收，降低了系统对资源的回收利用率。所以，为了提供系统对资源的回收利用率，应该想尽量缩短僵尸进程在系统内的驻留时间，即将 Z_t 的值缩小到如表达式 3.1，即 $Z_t = 2C + S$。一个能够实现这个目标的比较简单的方法是：在一个进程运行结束后，强行让调度器选中僵尸进程的母进程作为下一个运行的进程，而不是按既定的规则来选择进程。这样一来，不管就绪队列里有多少个进程，也不用关心母进程的优先级，只要进程运行结束后便能很快被母进程处理。如果采用这种方法，进程退出处理的过程如图 3-2 所示。

在进程 P_k 调用 `do_exit()` 函数运行结束成为僵尸进程后，CPU 从 P_k 进程的上下文切换到调度器的上下文，调度器继续运行起来后，首先从就绪队列里选中

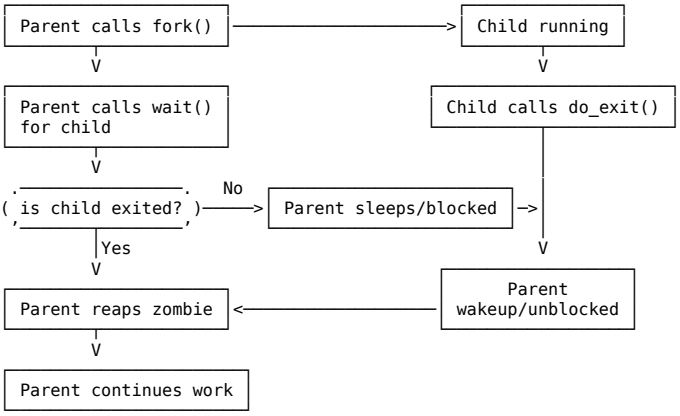

图 3-3　母进程等待子进程退出的过程
Fig. 3-3　Parent process waits for child to exit

P_k 进程的母进程，然后再进行上下文切换到这个母进程，最后母进程调用 `wait()` 函数便能立刻为僵尸进程 P_k 进行资源回收。在这个过程中，只需要发生两次进程切换，即在步骤 203 和步骤 205 两处。所以，采用这种方法的话，僵尸进程在系统内的存在时间可成功缩短到 $Z_t = 2C + S$。这种方法看似简单，并且也能成功做到将 Z_t 的值缩短到基本最小，然而却存在着以下两个问题：

1. 在每次发生进程退出的时候，调度器选中的都是该进程的母进程作为下一个运行的进程，这就打破了调度器原本的工作原则，使得原本应该作为下一个运行的进程不得不向后推迟，引起的后果可能是调度器的公平性被打破，或者本来作为下一个运行的实时进程因没有能及时得到运行而造成任务失败；
2. 母进程在被调度器选中时必须是已经处于等待状态，否则就会造成母进程凭空多占用了一次 CPU 使用权，而僵尸进程却没有得到清除。

此外，对于流程中原本存在的母进程因等待子进程退出而使自身工作受到耽搁的问题以及安全问题仍然没有得到解决。因此，以上这个方案显然不够完美，仍需进一步完善。

3.2.2　解决母进程工作被延迟的问题

在 Linux 系统中，当一个母进程创建完子进程后，便调用 `wait()` 函数以等待子进程的退出。如果母进程在等待的过程中，子进程尚未退出的话，母进程则会主动让出 CPU 使用权而进入睡眠状态。而子进程在退出的时候，会通过 `do_exit()` 函

数将正在睡眠的母进程唤醒。被唤醒的母进程在得到调度器的调度后执行 `wait()` 函数完成对僵尸进程的回收处理，当 `wait()` 函数成功返回后，母进程便继续执行自身的工作。这就是母进程等待子进程退出的整个过程，这个过程可用图 3-3 表示。

由于母进程必须在创建完子进程后等待子进程的退出，以便于能尽快处理僵尸进程，于是，这个等待的过程就导致了母进程自身任务的耽搁。这个问题对于 shell 进程来说，尤为突出。作为命令行解释器的 shell 进程的主要任务就是：等待用户输入命令，然后启动一个子进程来执行命令所要求的任务，接下来等待该子进程的退出，在该子进程退出返回后，shell 继续等待用户输入下一个命令，如此反复。由于 shell 必须等待子进程的退出返回后才能继续创建下一个任务，这就使得在当前创建的任务结束返回前，shell 进程无法继续创建新的任务。如果遇到任务繁重的子进程时，shell 就会需要经过漫长的等待后才能继续为用户创建新任务。这种等待对于用户来说，显然是糟糕的体验。因此，这个问题需要被很好地解决。解决这个问题最直接有效的办法就是：母进程不用再等待子进程的退出，换句话说就是把这个工作交给别的进程来完成。然而，这个工作不论是交给哪个进程，这个进程都需要花时间来完成这个等待工作。所以，如果不想任何进程因为需要为别的进程进行僵尸进程资源回收而陷入漫长的等待的话，那么最好的办法似乎就是，把每个进程的退出处理工作交还给每个进程自己来完成，也就是说把 `wait()` 函数所做的工作交给 `do_exit()` 函数来完成，于是，把原来的 `do_exit()` 函数改写成 `exit_wait()` 函数，如图 3-4 所示。

由于进程自己完成自身资源的回收释放，而不需要依赖于母进程来完成，因此，`exit_wait()` 函数便可以省略了原来 `do_exit()` 函数流程中的“notify parent”、“pass children to init” 以及 “mark zombie” 这几个步骤。采用这个方法不仅能解决母进程工作受到耽搁的问题，同时还能以最快的方式回收进程所占的资源，即进程只要一退出，便能立即自动释放资源，这无疑是最大限度地提高了系统对进程资源的回收利用率。然而，在具体实现的时候，有些资源却是无法由进程自身来完成回收的，如内核栈。这是因为，进程在完成最后一次上下文切换的过程中，会用到内核栈，而一旦进程完成了最后一次上下文切换（最后一次交出 CPU 使用

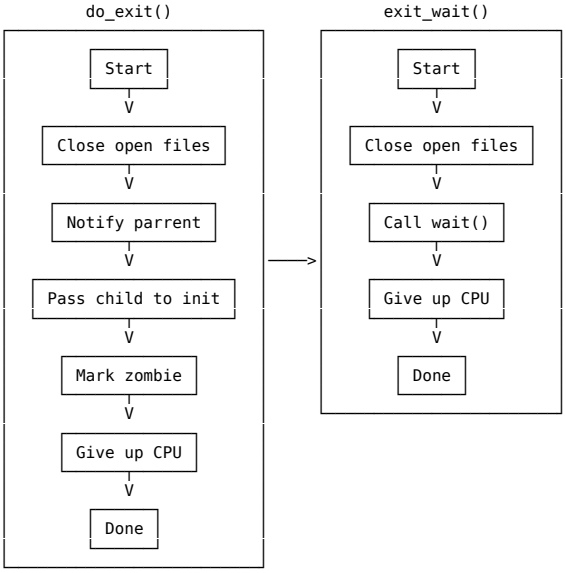

图 3-4　`exit_wait()` 函数的工作流程
Fig. 3-4　The workflow of `exit_wait()`

权）后，进程便无法再执行任何代码，所以，对于那些无法在进程上下文切换发生前释放的资源只能交给别的进程来释放。所以，这个由进程自身完成资源回收的办法仍需要进一步完善，以解决如何回收那些无法由进程自身释放的资源的问题。

3.2.3　解决安全问题

因为 Linux 是把处理僵尸进程的工作交给每个进程自己的母进程来完成，于是，僵尸进程能否成功被清除就主要取决于母进程是否能够成功调用 `wait()` 函数，或者母进程是否记得为子进程调用 `wait()` 函数。如果母进程由于程序自身的缺陷没有能成功调用 `wait()` 函数，或者说是程序员忘记调用 `wait()` 函数，那么僵尸进程将会一直存在系统中，直到母进程退出后才会被 `init` 进程处理。系统内有一两个僵尸进程驻留对系统来说，也许不会造成太大的麻烦。但是，如果是大量的僵尸进程长期驻留系统内的话，就容易导致系统的资源被耗光。比如说，系统遇上恶意用户程序，该程序通过母进程创建大量的子进程，然而这些子进程在被创建后并没有执行任何任务而是直接退出了，但是母进程却故意不调用 `wait()` 函数为任何的子进程进行僵尸进程处理，而且母进程一直没有主动退出系统，这导致系统里存在大量的僵尸进程而系统却无法对之进行处理。如果僵尸进程的数量多到一定程度，即可导致系统内有限的资源如进程表空间、PID 号被全部耗光而造成

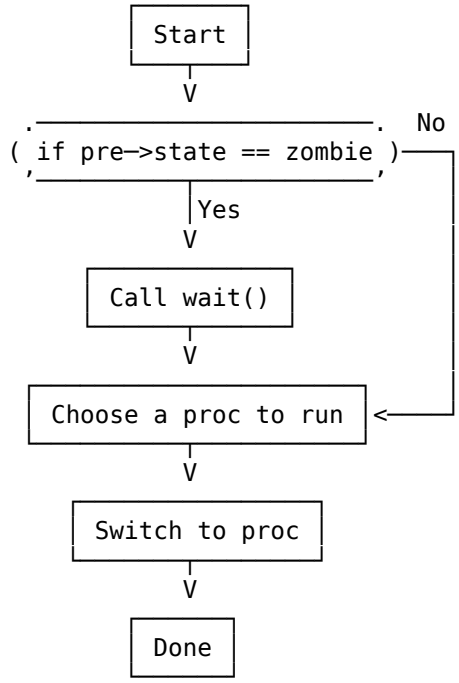

图 3-5　调用 wait() 函数后的调度器工作流程图
Fig. 3-5　The workflow of the scheduler after invoking wait()

系统无法再创建任何新进程的糟糕局面。这些安全隐患存在的根本原因是，系统需要依赖于不可靠用户母进程来完成僵尸进程的处理工作。因此，要从根本上杜绝这些安全隐患的最好办法就是把处理僵尸进程的工作交给可靠的系统进程，如 swapper 进程来处理。然而，swapper 进程是系统的空闲进程，这就意味着所有的僵尸进程都必须等待到 CPU 完全空闲的时候才能被 swapper 处理，这无疑也是让所有的僵尸进程在系统内较长停留。所以，必须找到一个可靠并且能迅速做出响应的系统进程来处理这个问题，而能满足这两个条件的就只有调度器进程而已了，于是，只能把这些工作交给调度器来完成。加入僵尸进程处理工作后的调度器的工作流程如图 3-5 所示。

把僵尸进程的处理工作交给调度器进程来处理，既能做到快速回收僵尸进程资源，又能避免原流程中存在的安全问题，这无疑是一个非常不错的选择，然而，调度器进程也有自己的工作，并且每次进程切换都必然需要调度器的介入，如果调度器的工作量明显增加的话必然会导致进程切换的所需要的时间增加，进而影响系统的响应速度。所以，这个方法还不够完美，仍需进一步改善，做到既能满足快速回收僵尸进程资源，又能避免安全隐患，同时又不会显著增加系统的响应

时间。

3.3 新方案设计

在上一节（第 3.2 节），本文从三个方面解决了 Linux 流程中存在的三个问题。接下来本文将从整体出发，综合考虑第 3.2 节提出的这三个问题，并在这三个局部解决方案的基础上，进行通盘筹划，找到全局最优方案，最终完成对 Linux 处理进程退出的流程的重新设计。

3.3.1 exit() 函数和 schedule() 函数的设计

第 3.1 节提出的让调度器强行选中母进程的局部方案做到了第一点，但是却无法做到第二、第三、第四点；第 3.2 节提出的让 exit() 函数负责全部的进程退出处理工作的局部方案从理论上讲做到了全部的四点，但是在实施的过程中却存在着部分无法由进程自身完成的任务；第 3.3 节提出的把回收僵尸进程资源的工作交给调度器的方案做到了第一、第二、第三点，但是却因调度器工作显著增加而降低了系统的整体性能，所以无法做到第四点。于是，综合考虑，在设计全局最优方案的时候，应该把进程退出的处理工作交给 exit() 函数和调度器共同处理，并且把一切能由进程自身处理的工作都交给 exit() 函数处理，而只把那些无法由 exit() 函数处理的工作交给调度器进程来处理。接下来所做的工作就是将 wait() 函数所做的工作划分为两部分：能被进程自身处理的部分和必须由其他进程代为处理的部分。wait() 函数完成的工作内容如下：

1. 释放进程所占的用户空间；
2. 释放进程的二级页表；
3. 释放进程所占的一级页表；
4. 释放进程所占的内核；
5. 清空进程表里该进程控制块所占的位置；
6. 向系统交回进程 PID 号；

由于进程在调用完 exit() 函数后，便不会再执行任何任务，所以不会使用到任何用户空间中的代码，所以，用于存放用户程序代码的用户空间便可以在 exit()

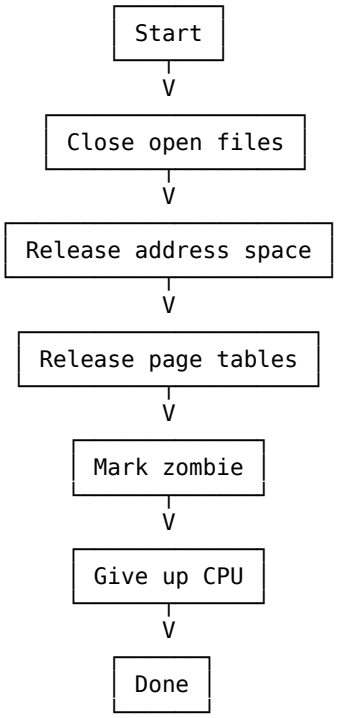

图 3-6 重新设计后的 exit() 函数

Fig. 3-6 Redesigned exit()

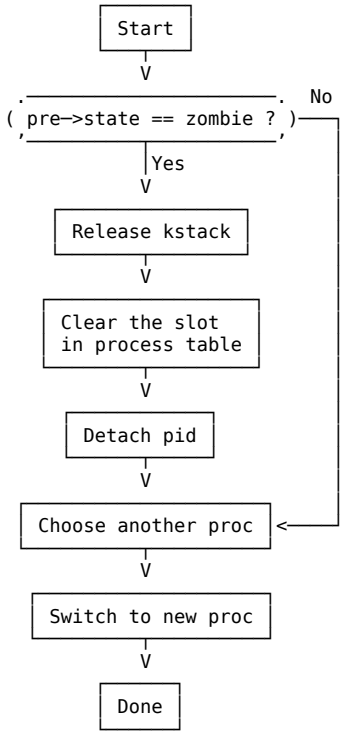

图 3-7 重新设计后的 schedule() 函数

Fig. 3-7 Redesigned schedule()

函数退出前释放，同理，用于映射用户空间的一级页表和二级页表也可以由进程自己完成释放。由于 exit() 函数在退出前会一直需要用到内核栈，故内核栈的释放必须在 exit() 函数完成后，也就是只能交给别的进程来完成。而为了防止在完成内核释放前，进程表里该进程控制块所占的位置以及进程的 PID 号被别的进程重复利用而引起冲突，所以，这两项资源的释放也必须由其他进程来完成。综上所述，wait() 函数里的释放进程所占的用户空间、二级页表以及以及页表这些工作应当交给 exit() 函数来完成，如图 3-6。而释放内核栈、清空进程表里的进程控制块所占的位置以及向系统交回进程 PID 号这几个工作应得交给 schedule() 函数来完成，如图 3-7 所示。

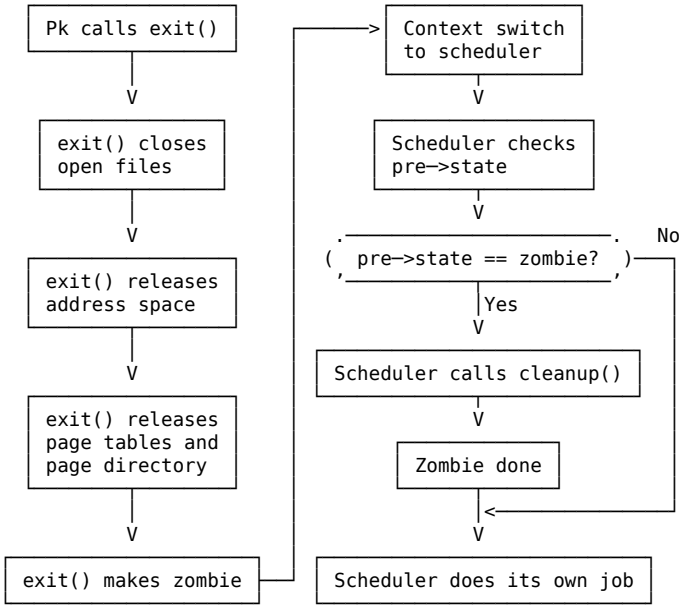

图 3-8　新设计后的进程退出处理流程图
Fig. 3-8　The redesigned workflow of process exit

3.3.2　新方案流程图

本文的最终方案是实现一种无需母进程等待的快速回收僵尸进程资源的方法,也就是通过 `exit()` 函数和 `schedule()` 函数的共同作用来代替母进程的 `wait()` 函数操作，以达到“即退即除”的快速清除僵尸进程的目的，同时还能避免 Linux 流程中因依赖于母进程的等待而存在的几个问题。如果在第 3.3.1 节的基础上，将调度器流程中，即图 3-7 中的第 3、第 4、第 5 步集成到一个 `cleanup()` 函数里的话，那么进程退出处理的最终流程如图 3-8 所示。

重新设计后处理流程具有以下几个优点:

1. 将僵尸进程的存在时间由原来的

$$Z_t = 2C + S + (2C + S + qum) * n \tag{3.3}$$

缩短到

$$Z_t = C \tag{3.4}$$

2. 无需依赖于母进程的等待，不会耽误母进程自身的工作执行;
3. 避免了 Linux 流程中存在的安全隐患。

4 流程实现

本文的最终目标是实现一种无需母进程等待的高效回收僵尸进程资源的方法，在第三章，本文已经对该方法进行了详细的方案设计，接下来，本文将阐述如何具体实现该方案。本章的主要工作包括：以 Linux v1.0 的内核版本为实验基础，对 `do_exit()` 函数和 `schedule()` 函数进行重新实现。

4.1 重写 `do_exit()` 函数

经过重新设计后，`exit()` 函数较之原来的 `do_exit()` 函数在工作内容发生了较大的变化，如图 4-1 所示。在重新设计后，`exit()` 函数需要自己处理物理地址空间的释放工作，因此，与原来的 `do_exit()` 相比，`exit()` 函数需要在工作流程中增加几个部分，即图 4-1 中的步骤 103、104、105；同时由于进程不需要再依赖于母进程来完成资源回收工作，所以，原来函数中的步骤 003、004 的这两项工作便可以省去。具体来说，`exit()` 函数的详细工作流程如下：

1. 关闭所有被进程打开的文件；
2. 释放进程的用户地址空间；
3. 释放进程的二级页表空间；
4. 释放进程的一级页表空间；
5. 设置进程的退出状态并释放 CPU 资源。

4.1.1 关闭文件

进程在运行过程中，需要访问各种各样的数据信息，而这些数据信息都是以文件的形式存在于系统之中。因此，进程的运行过程就是不断地操作文件的过程，如打开文件、关闭文件、读文件、写文件等等。进程每打开一个文件，都会记录

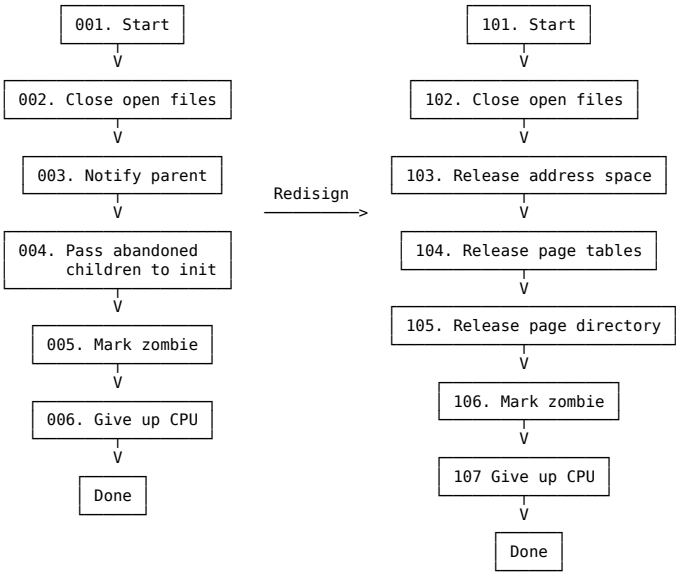

图 4-1　原 do_exit() 函数与 exit() 函数对比图
Fig. 4-1　The comparison of the new exit() and the old do_exit()

下该文件的信息，以供后期处理。为了便于管理，系统给每一个进程分配了一个文件数组 (ofile[])，用于记录所有被进程打开的文件的信息。当进程退出后，便需要对这些文件进行关闭。本文在重新设计的时候，采取的是把关闭文件的工作交给 exit() 函数来完成的方式。在具体实现 exit() 函数的时候，本文创建了一个名为 fileclose() 的函数来专门负责关闭所有打开了的文件。fileclose() 函数的具体实现参见 4-2，其主要的工作流程则如下：

1. 遍历进程的 ofile[] 数组，找到每一个被进程打开的文件；
2. 记录该文件的类型以及 INODE；
3. 将该文件对应的被引用数值设置为零；
4. 将该文件的类型设置为空；
5. 如果文件为管道类型，则释放该文件对应的内存页面和 INODE 资源；
6. 否则只需要释放该文件对应的 INODE 资源；
7. 最后将 ofile[] 数组里该文件对应的位置进行清零操作。

4.1.2　释放地址空间

每个进程都有自己独立的虚拟地址空间，这个虚拟地址空间被分为两个部分，内核部分和用户部分。虚拟地址空间通过页表映射到物理内存上，其中内核部分

```
for(int fd = 0; fd < NOFILE; fd++){
  if(curproc->ofile[fd]){
    struct *f = curproc->ofile;
    curproc->ofile[fd]->ref = 0;
    curproc->ofile[fd]->type = FD_NONE;
    if(f.type == FD_PIPE)
      pipeclose(f.pipe, f.writable);
    else if(f.type == FD_INODE){
      begin_op();
      iput(f.ip);
      end_op();
    }
    curproc->ofile[fd] = 0;
  }
}
```

图 4-2 fileclose() 函数的关键代码
Fig. 4-2 The key code of fileclose()

映射的是内核代码，而用户部分映射的则是进程相关的程序代码、数据、堆栈。进程退出后，系统需要回收进程所映射的物理内存。由于每个进程的内核部分映射的都是同一份内核代码，而这份内核代码禁止被释放，所以，当进程退出后，系统所需要释放的地址空间实际上只是用户部分映射物理内存页。如果用 KERNBASE 表示内核开始处的地址，而 PGSIZE 表示一个页面的大小的话，那么进程所需要释放的则是从零开始到 KERNBASE-PGSIZE 的这一部分空间。为了实现进程自己释放地址空间这一目标，需要在 exit() 函数里增加一个 deluvm(pgdir,KERNBASE,0) 函数，用于删除 pgdir 所指向的虚拟地址空间里从 0 到 KERNBASE 以下的这一部分用户空间。deluvm() 函数的工作流程如图 4-3 所示，deluvm() 函数的关键代码则如图 4-4 所示。

4.1.3 释放页表空间

页表是用于记录虚拟地址与物理地址的对应关系，在释放完进程的地址空间后，这些记录就可以删除了。在 32 位系统里，页表分为两级：页目录表（一级页表）和页表（二级页表）。页目录表记录的是每一个二级页表的开始地址，要找到二级页表必须访问页目录表，所以，必须在释放页目录表前释放所有的二级页表。在释放二级页表的时候，需要遍历页目录表，并从每一条记录里提取出二级页表的物理

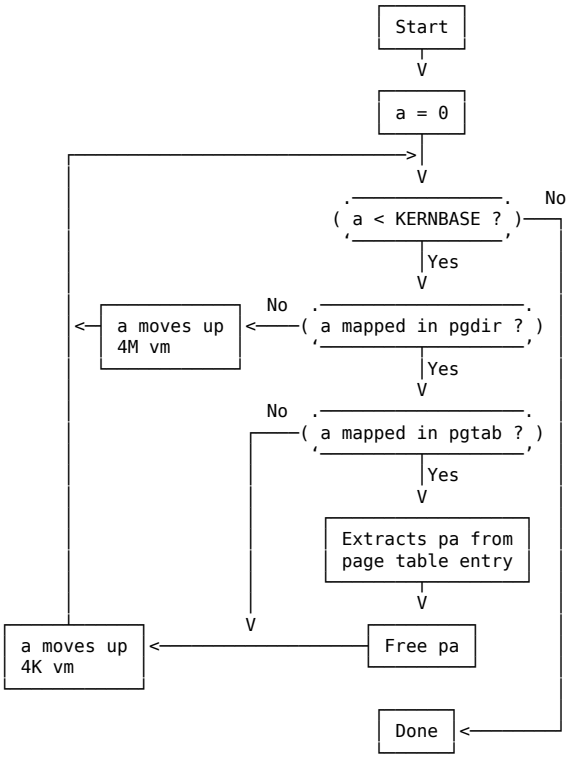

图 4-3　deluvm() 函数工作流程
Fig. 4-3　The workflow of deluvm()

```
for(a=0; a < KERNBASE; a += PGSIZE)
{
  pte = walkpgdir(pgdir, (char*)a, 0);
  if(!pte)
    a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
  else if((*pte & PTE_P) != 0){
    pa = PTE_ADDR(*pte);
    if(pa == 0)
      panic("kfree");
    char *v = P2V(pa);
    kfree(v);
    *pte = 0;
  }
}
```

图 4-4　deluvm() 函数的关键代码
Fig. 4-4　The key code of deluvm()

地址，然后将其释放。在释放页目录表的时候，就比较简单，因为页目录表只占用一个内存页面，并且这个页面的地址就是页目录指针（pgdir）的值。为了实现进程自己释放页表空间的这一目标，需要在 exit() 函数了增加一个 delpgt(pgdir) 函数，用于删除进程的二级页表以及页目录表。delpgt() 函数从头开始遍历 pgdir 指针所指向的页目录表里的每一条记录，首先检查该条记录是否为空，如果为空，则接着检查下一条记录，如果不为空则，检查该条记录的 p 标志位是否为非零，如果为非零则表明该条记录所对应的二级页表存在，需要进行删除操作，如果为零，则不需要进行删除操作，直接转向检测下一条记录。当 delpgt() 函数遍历完成整个页目录表后，即可对页目录表进行删除操作。delpgt() 函数的工作流程如图 4-5 所示，实现的关键代码则如图 4-6 所示。

4.2 重写 schedule() 函数

调度器（schedule() 函数）主要负责进程切换的工作，也就是按规定的调度算法从就绪队列里选取下一个符合调度条件的进程，并把 CPU 使用权交给它。所以，每当内核需要进行进程调度的时候，调度器都会自动介入。在 Linux 系统中，为了安全起见，每一个进程都有自己的运行环境，并且所有的进程都必须工作在自己的运行环境中。调度器本身作为一个内核进程，也有自己的运行环境。所以，调度器在每次工作之前都需要首先将 CPU 切换到自己的运行环境中。本文为了提高对僵尸进程资源的回收利用率而对 Linux 处理进程退出的流程进行了重新设计，schedule() 的工作内容因而发生了改变。schedule() 函数除了需要完成上述进程切换的工作外，还需要负责清理僵尸进程的工作。因此，重新设计后的 schedule() 函数的工作流程如下所示：

1. 将 CPU 切换到调度器自己的运行环境中；
2. 自动检测上一个运行的进程是否是僵尸进程；
3. 如果步骤 2 成立，则对僵尸进程进行清理；
4. 从就绪队列里选取下一个符号规定的进程 P_i；
5. 将 CPU 切换到进程 P_i 的运行环境。

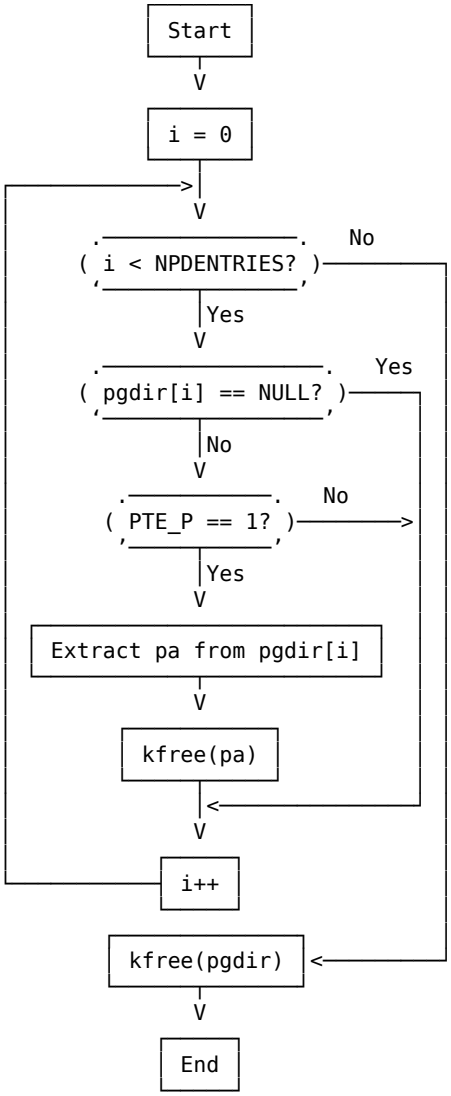

图 4-5　delpgt() 函数工作流程
Fig. 4-5　The workflow of delpgt()

```
for(i = 0; i < NPDENTRIES; i++){
  if(pgdir[i] & PTE_P){
    char * v = P2V(PTE_ADDR(pgdir[i]));
    kfree(v);
  }
}
kfree((char*)pgdir);
```

图 4-6　delpgt() 函数的关键代码
Fig. 4-6　The key code of delpgt()

```
pushcli();
mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                              sizeof(mycpu()->ts)-1, 0);
mycpu()->gdt[SEG_TSS].s = 0;
mycpu()->ts.ss0 = SEG_KDATA << 3;
mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
ltr(SEG_TSS << 3);
lcr3(V2P(p->pgdir));
popcli();
```

图 4-7 switchkvm() 函数的关键代码
Fig. 4-7 The key code of switchkvm()

4.2.1 切换 CPU 的运行环境

进程的运行环境包括：进程的上下文、页表、内核栈。所谓的进程的切换实际上就是将 CPU 从一个进程的运行环境切换到另外一个进程的运行环境。调度器作为一个内核进程，也有自己的上下文、页表、和内核栈。每次系统内发生进程调度的时候，会首先将 CPU 从当前运行的进程的上下文切换到调度器的上下文。而每一次有进程退出时系统内必然会发生进程调度，这个时候，系统会自动将 CPU 切换到调度器进程的上下文中。在调度器进程拿到 CPU 使用权之后，首先要做的是完成 CPU 运行环境的切换，即完成页表和内核栈的切换。只有在完成了页表和内核栈的切换工作后，调度器才能开始真正的工作，如清理僵尸进程、选取下一个能够运行的进程等。重新设计后的 schedule() 函数中负责切换 CPU 运行环境这项工作的函数是 switchkvm()，该函数的主要工作流程如下:

1. 屏蔽当前 CPU 的中断信号;
2. 将 CPU 切换到调度器对应的内核栈;
3. 将 CPU 切换到调度器对应的页表;
4. 重新恢复当前 CPU 的中断。

switchkvm(struct task_struct *p) 函数实现的关键部分如图 4-7 所示。

4.2.2 清理僵尸进程

一个要退出的进程在执行完 exit() 函数后便以僵尸进程的形式存在于系统内，等待被进一步清理。虽然经过重新设计后，exit() 函数已经对进程的大部分

```
cleanup(struct task_struct *pre)
{
  int pid=pre->pid;
  kfree(pre->kstack);
  pre->kstack = 0;
  pre->pid = 0;
  pre->parent = 0;
  pre->name[0] = 0;
  pre->killed = 0;
  pre->state = UNUSED;
  return pid;
}
```

图 4-8　cleanup() 函数的关键代码
Fig. 4-8　The key code of cleanup()

资源进行了清理，但僵尸进程仍然占用着一部分系统资源，如内核栈、进程描述符、以及进程 PID。对于这一部分残留的资源，只能通过调度器进程进行清理才能达到最快回收僵尸进程所占资源的目的。因此，本文在实现 schedule() 函数的过程中增加了一个 cleanup() 函数用来专门负责清理僵尸进程。cleanup() 函数实现的关键代码如图 4-8 所示，该函数的主要工作流程则如下：

1. 调用内核函数 kfree() 释放进程的内核栈；
2. 清空进程所占的进程描述符数据结构；
3. 将进程描述符数据结构的状态标志为未使用；
4. 向系统返回进程的 PID 号。

4.2.3　自动识别僵尸进程

由于 schedule() 进程必须负责僵尸进程的清理工作，因此在每一次进程上下文切换到调度器的时候，调度器首先检测上一个运行的进程是否是僵尸进程，如果是僵尸进程则调用 cleanup() 函数为其进行资源回收；如果不是僵尸进程，调度器则继续自己的日常工作，即继续从就绪队列里选中一个符合规定的进程作为下一个能够运行的进程，并调用 switch_to() 函数进行上下文切换。所以，为了能让调度器具备自动识别上一个运行的进程是否是僵尸进程的能力，需要在 schedule() 函数里添加相应的代码。首先需要在 schedule() 函数里添加一个用于记录上一个运行进程的变量*pre。接下来在 schedule() 进行上下文切换之前将*pre 的值

```
task_struct *pre=nextproc;
switchuvm(nextproc);
nextproc->state = RUNNING;
swtich_to(&(c->scheduler), nextproc->context);
switchkvm(curproc);
if(pre->state==zombie)
{  cleanup(pre);}
```

图 4-9 识别僵尸进程的相应代码
Fig. 4-9 How to identify a zombie

设置为调度器当前选中的进程，即下一个要运行的进程，最后在上下文切换完成之后，也就是 CPU 使用权回到调度器手中后便对*pre 所指的进程（上一个进程）的状态进行检测，如果状态是僵尸态，则表明该进程已经退出，需要对其进行资源回收，于是便可调用 cleanup() 函数进行处理。其中，需要增加的相应代码则如图 4-9 所示。

5 实验

本文的目的是实现一种无需母进程的等待快速回收僵尸进程所占资源的方法。为了实现这一目标，本文在第三章，对 Linux 处理进程退出的流程进行了重新设计，并在第四章对重新设计后的流程进行了详细的编程实现。接下来，本章将通过实验测试，将本文所设计和实现的方法与 Linux 原来的处理方法从三个方面进行对比，以证明本文所提的方法的正确性和有效性。比较内容主要有：

1. 与 Linux 原来的处理方法对比，是否做到了无需依赖于母进程的等待即可回收僵尸进程所占资源；
2. 与 Linux 原来的处理方法对比，是否缩短了僵尸进程在系统内的驻留时间；
3. 与 Linux 原来的处理方法对比，是否避免了潜在的系统资源被僵尸进程耗光的安全问题；

5.1 处理方式对比

Linux 系统对进程退出处理的方式是：一个进程运行结束后，它并没有能立即消失在系统中，而是以僵尸进程的身份存在，直到被母进程调用 `wait()` 函数对僵尸进程进行资源回收处理后进程才会彻底消失。这种处理方式的弱点是进程的退出处理必须依赖于进程的母进程的操作。如果要实现随时对任何的已运行结束的子进程进行僵尸进程处理，则母进程必须在创建完成子进程后，立即调用 `wait()` 函数进入等待状态，直到子进程运行结束后，`wait()` 函数方能进行僵尸进行处理。最后当僵尸进程处理完毕后，`wait()` 函数才能返回，母进程才能继续自己的工作。所以，对于母进程来说，这种方式容易导致母进程因为长时间等待子进程的退出而使自身任务的执行受到耽搁的情况。而本文所采取的处理方式是一种无需母进程等待即可快速回收僵尸进程所占资源的方法，也就是通过 `exit()` 函数与调度器

进程的合作方式来替代母进程的等待操作。

接下来，为了验证本文所实现的方法是否做到不需要母进程进行等待操作即可完成僵尸进程资源的回收工作，本文做了一个测试，将 Linux 对僵尸进程进程的处理方式与本文重新设计后的处理方式进行对比，以突出本文所实现的处理方式能成功避免母进程因等待子进程的退出而使自身工作受到耽搁的问题的这一优点。测试所包含的主要工作如下：

1. 即实现一个系统调用 `lszombies()`，用做查看系统内僵尸进程的一个工具。
2. 编写两个小程序：`waytest.c` 和 `waytest1.c`, 分别用于测试 Linux 原来的处理方式与本文所实现的处理方式的区别。
3. 分析对比 `waytest.c` 和 `waytest1.c` 的测试结果，突出本文所实现的方法在处理方式上的优点。

5.1.1 工具实现

进程在最后一次交出 CPU 使用权后，便以僵尸进程的形势存在于系统内。在 Linux 原来的处理方式下，僵尸进程会一直存在直到被母进程 `waited` 后才消失，而在本文重新设计后的处理方式下，僵尸进程在调度器开始下一轮的进程切换前就已经消失。为了方便测试以及更好地观察实验结果，本文编写了两个实验用的小程序：`waytest.c` 和 `waytest1.c`，以及一个系统调用 `lszombies()`。其中 `waytest.c` 和 `waytest1.c` 分别运行于未改变前的 Linux 系统与经本文改变后的 Linux 系统，而 `lszombies()` 系统调用则专门用于查看系统内每一个僵尸进程的存在情况。通过 `lszombies()` 系统调用可非常直观地观察到僵尸进程在母进程调用 `wait()` 函数前后的变化情况，这种变化可以准确地说明在未经改变前的系统下，僵尸进程的处理必须依赖于母进程的等待操作。实现 `lszombies()` 系统调用的主要工作就是实现一个相应的系统调用处理函数。在 Linux 系统下，系统调用的名称与负责负责处理该系统调用的内核函数的名称往往是一致的，本文也遵循这种规则。负责处理 `lszombies()` 系统调用的内核函数 `lszombies()` 的主要工作是向终端打印出系统存在的每一个僵尸进程的 `PID` 号以及僵尸进程总数量，该内核函数的具体实现代码如图 5-1 所示。

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == ZOMBIE){
    c++;
    pid=p->pid;
    cprintf("zombie pid %d\n",pid);
  }
}
release(&ptable.lock);
cprintf("Total: %d zombies\n",c);
```

图 5-1　lszombies() 工具实现的关键代码
Fig. 5-1　The implementation of lszombies()

waytest.c 程序，具体实现参见附录 A.1, 用于在未经改变前的 Linux 系统下运行测试, 以突出在原 Linux 系统中, 僵尸进程的处理工作必须依赖于母进程的等待操作才能完成。在 waytest.c 里，母进程为了完成僵尸进程的处理任务只有两种选择: 第一, 要么选择在创建完成子进程后便立刻调用 wait() 进入等待模式, 而自身的任务推迟到僵尸进程处理完成后再继续，如 waytest.c 中的 test1；第二，要么选择先完成自身的工作后再调用 wait() 函数对僵尸进程进行处理 waytest.c 中的 test1。第一个选择的优点是能够实现对僵尸进程的及时处理, 缺点却是母进程必须花费大量的时间用于等待子进程工作的完成。第二种选择的优点是母进程可以直接开始自身的工作而不用花费大量的时间等待子进程的退出，缺点在母进程工作完成前出现的僵尸进程必须等待到母进程任务完成后才能被处理。所以，这种依赖于母进程的操作来处理僵尸进程的方式，不论母进程是先处理僵尸进程还是后处理僵尸，系统都要付出额外的开销（时间或空间上的）。

然而，在经过本文的重新设计后，僵尸进程处理的工作由调度器来完成，母进程不再需要为任何的子进程进行僵尸进程处理，所以在 waytest1.c 程序中，母进程在创建完成子进程后便可立即开始自身的任务执行，并且在任务完成后母进程可以直接退出而不需要调用 wait() 进行操作。waytest1.c 程序的具体实现参见附录 A.1。

5.1.2 测试结果对比

运行于未经本文重新设计前的系统内的 waytest.c 程序的运行输出结果如图 5-2 所示。waytest.c 的输出结果（图 5-2）分为两部分：Test1 和 Test2，分别是对母进程先等待子进程退出后继续自身工作和先完成自身工作后处理僵尸进程的两种情况进行测试。在 Test1 中，母进程在完成子进程的创建后，便立刻进入了等待状态。然而，子进程图 5-2 中的 child 4 却需要花费一定的时间才能结束运行，所以，在子进程运行的这段时间里母进程只能保持等待。当子进程最终运行结束后，wait() 函数便能迅速检测到僵尸进程，并当即对其进行资源回收处理。所以在 wait() 函数成功返回后，通过 lszombies() 系统调用并不会检测到系统内存在着僵尸进程。在 Test2 中，母进程在完成子进程的创建后，选择首先继续自身的工作执行，这时候尽管子进程 (图 5-2 中的 child 5) 已经早早运行结束退出了，通过 lszombies() 工具也能检测到僵尸进程（图 5-2 中的 zombie pid 5）的存在，但是由于母进程忙于自身工作的执行，对这个僵尸进程无暇顾及，所以这个僵尸进程一直存在于系统内，等待被母进程 waited。最后母进程工作完成后，调用了 wait() 函数对僵尸进程进行处理，此时，再通过 lszombies() 工具检测到系统里的僵尸进程数量为零，即表明（zombie pid 5）已经被处理了。

waytest1.c 运行于本文重新设计后的系统内，其运行结果输出图则如图 5-3 所示。由于经过重新设计，所有的僵尸进程的处理工作都交给了调度器来完成，也就是说任何的母进程都不再需要担负起为子进程处理僵尸进程的任务，所以，在 waytest1.c 程序中，母进程自始至终都不需要调用 wait() 进行僵尸进程处理，而是在创建完成子进程（图 5-3 中的 child pid 4）后便直接开始自身的工作。虽然在运行的过程中，子进程先于母进程而退出了，但是母进程并不需要停下手中的工作来为其进行僵尸进程处理而是继续执行自身的工作。最后当母进程执行完毕后，通过 lszombies() 工具查看先前已退出的子进程的僵尸进程是否还存在，而此时 lszombies() 的返回结果是：系统里的僵尸进程总数为零，也就是说僵尸进程已经被别的进程（调度器）处理了，这就表明了经过本文的重新设计后，僵尸进程的处理工作不再需要依赖于母进程。

对比 waytest.c 与 waytest1.c 的输出结果，可以看出本文所实现的处理方式

```
$ waytest
Test1: parent waits zombies before doing its job
parent waiting for child to complete ......
child 4 working
......
child 4 is exiting:
waited child 4
Total: 0 zombies!
parent doing its own work:
......
parent's work done

_____________________________________________

Test2: parent waits zombies after its job done:
child 5 exiting:
zombie pid 5
Total: 1 zombies!
parent doing its own work:
......
parent's work done, calling wait:
......
Total: 0 zombies!
$ ~/.../src(master)$
```

图 5-2　waytest.c 的输出结果
Fig. 5-2　The output of waytest.c

的优点是：不需要母进程的任何等待操作即可完成僵尸进程的处理工作，这就从根本上解决了母进程因等待子进程的退出而造成自身任务被推迟的问题。

5.2　资源回收的速度对比

为了提高系统对资源的回收利用率，系统需要及时对每一个已经运行结束的进程进行资源回收。系统处理进程退出的过程实际上就是系统对僵尸进程资源回收的一个过程。由于 Linux 系统把僵尸进程的处理工作交给了母进程来处理，导

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ waytest1
$
parent created child pid: 4
parent doing its own work
......
child 4 is working
......
child 4 exiting!
parent's work is done
Total: 0 zombies
```

图 5-3　waytest1.c 的输出结果
Fig. 5-3　The output of waytest1.c

致了系统对僵尸进程所占资源的回收利用存在着时间延迟。本文通过把僵尸进程的处理工作交给调度器来完成以达成对僵尸进程所占资源的快速回收，以此提高系统的资源的回收利用率。接下来，本文将通过实验测试来证明，本文所实现的方法确实能够大大提高回收僵尸进程所占资源的速度。测试的所需完成的主要工作包括：

1. 实现一个系统调用 `getfreepages()`，用于随时查看系统内物理内存空闲页面总数，以及两个小程序：`ztime.c` 和 `ztime1.c`，用于测试资源回收的速度。其中 `ztime.c` 运行于未经本文改变前的 Linux 系统，而 `ztime1.c` 则运行于经本文重新设计后的系统。
2. 对实验结果进行对比分析，以突出本文所实现方法能够在很大程度上提高资源的回收利用率的这一优点。

5.2.1 工具实现

在未经改变前的 Linux 系统中，当一个进程调用 `exit()` 函数进行退出后，进程只是释放了 CPU 资源，而本身所占用的内存资源却尚未被释放。接下来，进程则以僵尸进程的身份继续存在系统中，以等待被母进程处理。母进程则通过调用 `wait()` 函数来释放僵尸进程的资源，所以当 `wait()` 函数返回的时候就表明僵尸进程的内存资源已经被释放回收了。为了验证僵尸进程的内存资源是否已经被释放了，本文实现了一个工具：`getfreepages()` 系统调用，用于输出系统内当前时刻物理内存的空闲页面总数。用户可通过这个系统调用对比在调用 `wait()` 函数前与调用 `wait()` 函数后，内存空闲页面的总数的变化情况来判断内存是否得到了释放回收。负责处理 `getfreepages()` 系统调用的内核函数 `getfreepages()` 的具体实现如图 5-4 所示。

```
uint getfreepages(void)
{
  uint numfreepgs;
  acquire(&kmem.lock);
  numfreepgs=kmem.freepages;
  release(&kmem.lock);
  return numfreepgs;
}
```

图 5-4　getfreepages() 函数的代码
Fig. 5-4　The implementation of getfreepages()

ztime.c 与 ztime1.c 程序的主要作用是测试系统回收僵尸进程资源的速度，其中 ztime.c 运行于未经本文改变前的 Linux 系统，而 ztime1.c 则运行于经本文重新设计后的 Linux 系统。由于僵尸进程被清除完成的时刻即是资源回收完成的时刻，所以，回收僵尸进程资源的速度直接与系统清除僵尸进程的快慢程度相关。在未经本文改变前，Linux 通过母进程的等待操作来清除僵尸进程，而经过本文的重新设计后，则是通过调度器来完成清除工作。两者的区别是：前者完成僵尸进程的清除工作需要花费的时间是子进程调用 exit() 结束后与母进程调用 wait() 返回后的这两个时间点的差值，而后者需要花费的时间则是子进程调用 exit() 结束后与 cleanup() 函数返回后的这两个时间点的差值。在测试的时候，为了获得这两个差值，本文在 exit()、wait() 和 cleanup() 这几个函数返回前输出系统当前的时刻值（以时钟 ticks 值表示），再通过这些返回值来计算出时间差值。

ztime.c 与 ztime1.c 程序首先通过调用 getfreepages() 系统调用来观察母进程在创建子进程前后的物理内存空闲页面的总数变化来获知子进程总共占用的内存页面数目；然后在子进程调用完成 exit() 函数后再次观察内存空闲页面总数是否有所变化；最后在 ztime.c 中的母进程调用 wait() 函数后再次查看内存空闲页面总数的变化情况。由于经过本文重新设计后，僵尸进程资源的回收处理不需要通过 wait() 来完成，所以，这一步在 ztime1.c 中可以省略。ztime.c 与 ztime1.c 的程序代码参见附录 A.2。

5.2.2 实验结果分析对比

程序 ztime.c 和 ztime1.c 的输出结果分别如图 5-5 和图 5-6 所示。这两个输出结果都分为两部分：第一部分是在子进程退出时，就绪队列里只存在一个进程即母进程的情况下测试系统回收僵尸进程资源所需的时间的输出结果；第二种情况则是在就绪队列里有多个进程的情况下的测试输出结果。从图 5-5 的 Test1 输出结果中，可以看出创建一个子进程（pid 4）需要花费 $56727 - 56659 = 68$ 个空闲页面。接下来，子进程在第 431 个 tick 的时候退出了，此时系统内的空闲页面总数仍然是 56659, 并没有因为子进程的退出而增加。而当僵尸进程 pid 4 在第 432 个 tick 的时刻被清除后，系统内空闲页面总数增加到了创建子进程前的总数 (56727)，这就表明了，只有在僵尸进程被清理完成后，系统才能回收进程所占用的资源。并且，在就绪队列里只有一个进程的情况下，僵尸进程从产生到被完全处理的这个过程，经历了 $432 - 431 = 1$ 个 tick。一个 tick 的值相当于 10ms，如果以 Z_t 表示这个过程需要花费的时间的话，那么 $Z_t = 1 \times 10ms = 10ms$。而从 Test2 的输出结果可以看出，当就绪队列里的进程数量增加时，Z_t 的值也随之增加。接下来，将以 np 表示进程的数量，从输出结果可以看出：

- 当 np 的值为 10 的时候, Z_t 的值就增加到 $854-844 = 10$ 个 ticks, 即 $100ms$。
- 当 np 的值为 20 的时候，Z_t 的值就增加到 20 个 ticks，即 $200ms$。
- 当 np 的值为 30 的时候，Z_t 的值就增加到 30 个 ticks，即 300ms。
- .....
- 当 np 的值为 100 的时候，Z_t 的值就增加到 100 个 ticks，即 1000ms。

这就表明：在没有经过本文的重新设计的情况下，系统回收一个僵尸进程的时间与系统内的进程数量有直接关系：进程数量越多，系统回收僵尸进程资源所需要的时间就越长，即速度越慢。回收僵尸进程资源所需的时间与就绪队列里进程的数量之间的关系可用 $Z_t = K * T_0$ 表示，其中 K 代表就绪队列中的进程数量，T_0 则代表在就绪队列里只有母进程唯一一个进程的理想情况下回收僵尸进程所需要的时间。从实验结果图 5-5 可得出，当 $K = 1$ 时，$Z_t = T_0 = 10ms$。

从 ztime1.c 的输出结果（图 5-6）可以看出，母进程创建一个子进程同样消耗了 $56729-56661 = 68$ 个内存页面。不一样的地方是, 在只有一个母进程的情况下，

子进程退出成为僵尸进程的时刻是第 263 个 `tick`，而这个僵尸进程被处理的时刻也是在第 263 个 `tick`, 这就表明了在经过本文的重新设计后, 僵尸进程从产生到被清除只经历了 0 $(263-263=0)$ 个 tick, 即僵尸进程的处理时间降到了 10ms 以下。如果将这个时间精确到微秒 (microsecond) 级别后则是 2395 $(9383-6988=2395)$ s。在僵尸进程被清除后，系统内的空闲页面总数增加到了创建子进程前的总数 (56729)，这就表明了随着僵尸进程被清除的同时，系统也完成了对进程资源的回收。最后，从这个输出结果（图 5-6）可以得出，经过重新设计后系统回收僵尸进程所需的时间（Z_t）与进程数量（np）之间的对应关系如下所示：

- 当 np 的值为 1 的时候，Z_t 的值是 2395（$9383-6988=2395$） s。
- 当 np 的值为 10 的时候，Z_t 的值则是 660（$1770-510=1260$） s。
- 当 np 的值为 20 的时候，Z_t 的值则是 2456（$3693-1237=2456$） s。
- 当 np 的值为 30 的时候，Z_t 的值则是 2417（$3152-735=2417$） s。
- .....
- 当 np 的值为 100 的时候，Z_t 的值则是 2602（$3092-490=2602$） s。

如果以毫秒（millisecond）的形式表示 Z_t 的值的话，则 np 与 Z_t 之间的对应关系如下：

np：	1	10	20	30	40	50	60	70	80	90	100
$\mathbf{Z_t}$：	2.395	1.260	2.456	2.417	2.315	2.381	2.453	2.544	2.924	2.505	2.602

对比 `ztime.c` 的输出结果图 5-5 和 `ztime1.c` 的输出结果图 5-6，可知：在原来的 linux 系统中，回收僵尸进程资源所需要的时间可由公式 $Z_t = K * T_0$ 表示，其中 `K` 代表就绪队列中的进程数量，T_0 则代表在就绪队列里只有母进程唯一一个进程的理想情况下回收僵尸进程所需要的时间，也就是 10ms。然而，在经过本文的重新设计后，系统回收僵尸进程所需的时间与就绪队列里进程的数量无关，无论就绪队列里是有 1 个进程还是 100 个进程，Z_t' 的值都处于 $1.260ms$ $2.924ms$ 之间，都小于 10ms。因此，可得出结论：经过本文的重新设计后，在很大程度上提高了系统回收僵尸进程资源的速度。最后，本文将原来的 Linux 系统回收僵尸进程所需的时间 Z_t 与经本文重新设计后的系统回收僵尸进程上所需的时间 Z_t' 的对比情况做成图表，如图 5-7 所示。

```
$ ztime

Test1: With only 1 proc in ready queue when child exited
56727 freepages in system before forked
56659 freepages in system after forked
pid 4 became a zombie at 431 ticks
56659 freepages in system after exited
zombie pid: 4 got reaped at 432 ticks
56727 freepages in system after waited

Test2: With 10 procs in ready queue when child exited
pid 5 became a zombie at 844 ticks
zombie pid: 5 got reaped at 854 ticks

Test2: With 20 procs in ready queue when child exited
pid 16 became a zombie at 1281 ticks
zombie pid: 16 got reaped at 1301 ticks

Test2: With 30 procs in ready queue when child exited
pid 27 became a zombie at 1748 ticks
zombie pid: 27 got reaped at 1778 ticks

Test2: With 40 procs in ready queue when child exited
pid 38 became a zombie at 2224 ticks
zombie pid: 38 got reaped at 2264 ticks

Test2: With 50 procs in ready queue when child exited
pid 49 became a zombie at 2731 ticks
zombie pid: 49 got reaped at 2781 ticks

Test2: With 60 procs in ready queue when child exited
pid 60 became a zombie at 3287 ticks
zombie pid: 60 got reaped at 3347 ticks

Test2: With 70 procs in ready queue when child exited
pid 71 became a zombie at 3873 ticks
zombie pid: 71 got reaped at 3943 ticks

Test2: With 80 procs in ready queue when child exited
pid 82 became a zombie at 4469 ticks
zombie pid: 82 got reaped at 4549 ticks

Test2: With 90 procs in ready queue when child exited
pid 93 became a zombie at 5057 ticks
zombie pid: 93 got reaped at 5147 ticks

Test2: With 100 procs in ready queue when child exited
pid 104 became a zombie at 5713 ticks
zombie pid: 104 got reaped at 5813 ticks
$
```

图 5-5 `ztime.c` 的输出结果
Fig. 5-5 The output of `ztime.c`

```
$ ztime1
Test1: With only 1 proc in ready queue when child exited
56729 freepages in system before forked
56661 freepages in system after forked
child 4 became zombie at 263 ticks, 6988 us
zombie 4 got reaped at 263 ticks,9383 us
56729 freepages in system after exited

Test2: With 10 procs in ready queue when child exited
child 5 became zombie at 476 ticks, 510 us
zombie 5 got reaped at 476 ticks,1770 us

Test2: With 20 procs in ready queue when child exited
child 16 became zombie at 845 ticks, 1237 us
zombie 16 got reaped at 845 ticks,3693 us

Test2: With 30 procs in ready queue when child exited
child 27 became zombie at 1273 ticks, 735 us
zombie 27 got reaped at 1273 ticks,3152 us

Test2: With 40 procs in ready queue when child exited
child 38 became zombie at 1849 ticks, 2992 us
zombie 38 got reaped at 1849 ticks,5307 us

Test2: With 50 procs in ready queue when child exited
child 49 became zombie at 2476 ticks, 433 us
zombie 49 got reaped at 2476 ticks,2814 us

Test2: With 60 procs in ready queue when child exited
child 60 became zombie at 3146 ticks, 1012 us
zombie 60 got reaped at 3146 ticks,3465 us

Test2: With 70 procs in ready queue when child exited
child 71 became zombie at 4005 ticks, 396 us
zombie 71 got reaped at 4005 ticks,2941 us

Test2: With 80 procs in ready queue when child exited
child 82 became zombie at 4868 ticks, 428 us
zombie 82 got reaped at 4868 ticks,3352 us

Test2: With 90 procs in ready queue when child exited
child 93 became zombie at 5766 ticks, 645 us
zombie 93 got reaped at 5766 ticks,3150 us

Test2: With 100 procs in ready queue when child exited
child 104 became zombie at 6866 ticks, 490 us
zombie 104 got reaped at 6866 ticks,3092 us
```

图 5-6　ztime1.c 的输出结果

Fig. 5-6　The output of ztime1.c

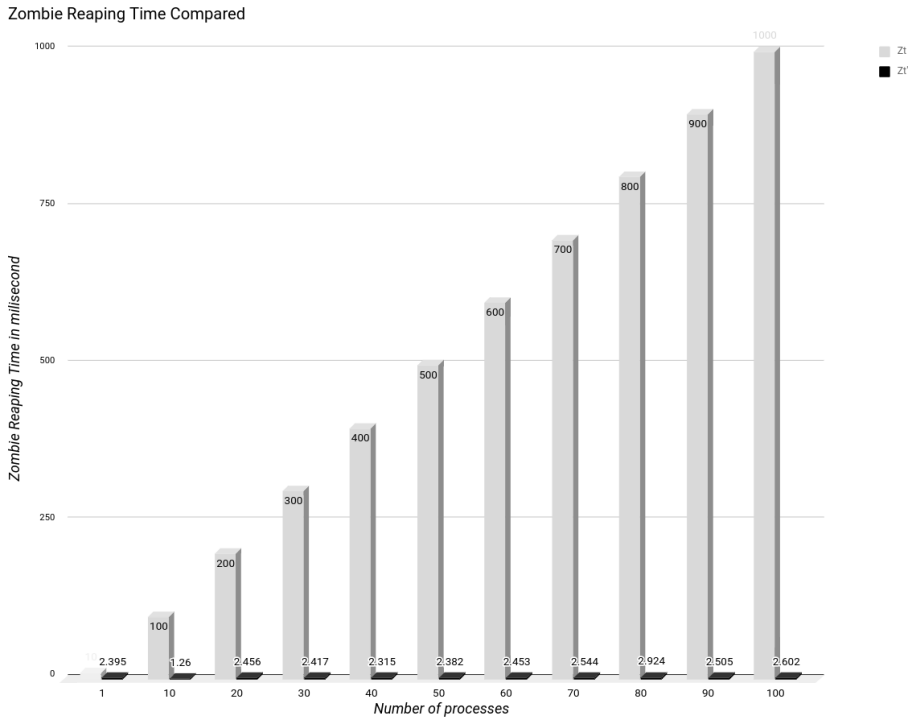

图 5-7 回收僵尸进程资源所需的时间对比图
Fig. 5-7 The comparison of resources releasing time of zombie collection

5.3 安全性对比

Linux 系统把处理每个僵尸进程的工作都交给了进程各自的母进程来完成。这种依赖于母进程的等待操作来完成资源回收的方式存在着安全隐患，毕竟用户进程不一定是安全可靠。如果一个用户编程不当，可能造成系统内长时间存在着若干个僵尸进程，这种情况虽然不会造成什么严重的后果。但是如果遇上一个恶意用户进程，情况就没有这么乐观了。如果一个母进程在短时间内创建了大量的子进程后，便开始了自己的无限循环“工作”，而那些子进程实际上并没有做任何事情就直接退出了，但是由于母进程故意“迟迟”不对任何的僵尸进程进行清除而导致掌握在这大批量的僵尸进程手中的资源一直得不到回收释放，而最终导致系统资源的耗尽。本文为了解决这种安全隐患，改变了依赖于母进程的等待操作来进行僵尸进程资源回收的方式，而是利用调度器来代替母进程完成资源回收。这么做的好处是：调度器是系统进程，安全可靠，最主要是它必然会在每一次发生进程退出时自动介入运行，所以，只要进程一旦退出成为僵尸进程后，立马就会被调度器自动进行资源回收。接下来，为了验证本文所实现的方法确实解决了 Linux

处理僵尸进程流程中存在的安全问题，本文编写了一个小程序，Zattack.c，用于模拟恶意用户进程的攻击。

5.3.1 工具实现

进程表是 Linux 系统里用于存放所有进程控制块的一个数据结构，内核每创建一个进程，都需要从这个数据结构里申请一个空位置 (slot)，用来存放该进程的各种状态信息。所以，如果进程表满了的话，内核便因无法申请到一个空的 slot 来存放新进程的信息而导致创建新进程的操作失败，而无法创建新进程就意味着无法执行新任务。为了方便实验，本文将进程表的最大容量值 MAXPROC 设置为 100。在 Zattack.c 中，程序首先创建了一个进程用于运行 Zattack() 函数，该函数的核心工作是模拟一个恶意进程通过短期内产生大量的僵尸进程来试图耗光进程表的资源。Zattack 进程试图创建 100 个子进程，这些子进程在创建后并没有执行任何任务而是直接退出成为了僵尸进程。而 Zattack 进程本身作为母进程却没有调用 wait() 函数对任何的僵尸进程进行资源回收，而是开始自身的“毫无意义”的无限循环。在 Zattack 进程启动后，Zattack.c 程序尝试创建新的进程以执行新的任务，以验证系统是否受到恶意进程 Zattack 的影响。为了提高测试的准确性，程序每隔两秒钟进行一次继续创建新进程的尝试，并且总共尝试十次。Zattack.c 程序将先后运行于未经本文改变前的 Linux 系统内以及经本文重新设计后的 Linux 系统内，以通过输出结果的区别来验证本文所实现方法能成功提高系统的安全性，Zattack.c 具体实现参见附录 A.3。

5.3.2 输出结果分析对比

将 Zattack.c 程序运行于未经本文改变前的 Linux 系统内得到的输出结果如图 5-8 所示。

而运行于经过本文重新设计后的 Linux 系统内得出的输出结果则如图 5-9 所示。从图 5-8 可以看出，虽然 Zattack 进程试图创建 100 个子进程，但是最终在创建完第 96 个子进程后便无法再继续创建子进程了，原因是进程表的最多只能容纳 100 个进程，而这其中包括了 init 进程、shell 进程、运行 Zattack.c 的进程以及

运行 Zattack() 函数的进程，而除去这四个进程所占的位置后，进程表里就只剩下 96（$100-4=96$）个空位置了。在创建完 96 个子进程后，进程 Zattack 便开始进入了无限循环，此后，系统了开始以每隔两秒终创建一个新进程的尝试，并且总共尝试了 10 次，但最后都是以失败告终。原因就是 Zattack 进程创建的僵尸进程一直没有得到清除而导致进程表一直处于“被占满”状态从而导致了所有创建新进程的操作都失败。这种情况带来的严重后果就是：系统形同瘫痪，无法再执行任何的新任务。

从图 5-9 可以看出：一开始的 Zattack 进程试图创建 100 个子进程的时候，却没有因为进程表里只有 96 个空位置而受限，原因就是，每一个被 Zattack 进程创建的子进程在被创建完成后立马就退出了，而经过本文的重新设计后，进程退出后几乎是立马就被资源回收了（这一点在上一节已经被实验验证过了）。同样情况，虽然 Zattack 进程试图通过创建 100 个僵尸进程来绝对占满整个进程表的空间，以阻止系统继续创建新进程，但是事实却是：系统在 Zattack 进程启动后，做了 10 次尝试创建新进程的举动，而每一次尝试的结果都是成功的。这就表明了，经过重新设计与实现后，系统具备了避免 Zattack 进程攻击的能力。这也验证了本文所实现方法的优点：可成功避免原 Linux 系统处理僵尸进程的过程中存在的安全隐患，因此大大提高了系统的安全性。

至此，本文通过三个测试验证了本文所实现的方法与原 Linux 的处理方法相比，具有三个优点：

1. 无需依赖于母进程等待即可实现对僵尸进程的处理；
2. 能够更快速回收僵尸进程的资源；
3. 能够避免原 Linux 处理流程中存在的安全隐患，安全性更高。

```
Machine  View
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$
$
$ Zattack
Proc Zattack running:
......
Forked 96 procs
Parent in Zattack infinite looping
Trying to fork new procs every 2 seconds after Zattack proc
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
Process table full,Fork failed
$ _
```

图 5-8　Zattack.c 在未经改变前的系统内运行的输出结果

Fig. 5-8　The system output before modifying Zattack.c

```
Machine  View
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ Zattack
$ Zattack running:
......
Fork claimes to work 100 times
Parent in Zattack infinite looping
Trying to fork new procs every 2 seconds after Zattack proc
Forked new proc 105
Forked new proc 106
Forked new proc 107
Forked new proc 108
Forked new proc 109
Forked new proc 110
Forked new proc 111
Forked new proc 112
Forked new proc 113
Forked new proc 114

$ _
```

图 5-9　Zattack.c 在重新设计后的系统内运行的输出结果

Fig. 5-9　The system output after redesigning Zattack.c

6 总结与展望

僵尸进程是 Linux 系统处理进程退出事务的过程中产生的一种处于特殊状态的进程，僵尸进程虽然不再运行，但是仍然占用着各种内存资源。僵尸进程产生后便一直处于系统中，直到被系统清除处理后才会消失。为了提高系统系统对内存资源的回收利用率，必须尽早处理僵尸进程。本文关注的是 Linux 处理进程退出过程中对僵尸进程资源的回收问题，针对 Linux 处理过程中存在的必须依赖于母进程进程的等待来回收僵尸进程的资源而造成的资源回收速度不够的快的问题以及存在安全隐患的问题，本文提出了采用调度器来替代母进程进行僵尸进程资源回收的新方法。总的来说，本文是从以下几个方面来完成论文的研究：

1. 提取 Linux 系统处理进程退出事务的过程模型，从本质上分析了僵尸进程产生的根源以及 Linux 对此的处理过程，并指出了 Linux 处理流程中潜在的必须依赖于母进程的等待操作、回收僵尸进程资源速度过慢、以及存在安全隐患的三个问题。
2. 面对 Linux 处理流程中存在的三个问题，本文提出了一种能够无需依赖于母进程的等待操作即可快速、安全可靠地回收僵尸进程资源的新方法，即采用调度器来替代母进程进行僵尸进程资源回收的方法。本文对所提的方法进行了方案设计，首先从局部开始，一一解决 Linux 流程中存在的这三个问题，然后在基于局部设计的基础上，进行通盘考虑，最终设计出一个能够同时解决这三个问题的全局解决方案，即通过采用 `exit()` 函数负责大部分的进程资源回收工作而调度器只负责少部分无法由 `exit()` 完成的回收工作的共同合作的方式来完美地达到同时解决这三个问题的目标。
3. 本文对所设计的方案进行了详细的编程实现，并通过实验分别从三个方面测试验证了本文所实现的方法与 Linux 原来的处理方法相比具有无需依赖于母

进程、能够更快速地回收僵尸进程资源、能够避免安全隐患这三个优点，即本文成功做到了能够同时解决 Linux 处理流程中存在的三个问题。

在将来的学习和工作中，本人将从以下两个方面展开后续的研究：

- 目前本文只是在虚拟机（QEMU）里进行测试验证了本文所实现的方法的正确性与有效性，但是尚未完成把经过本文重新设计后的内核用于实际机器上运行测试这一步，因此，暂时无法预测本文所实现的方法是否会对系统运行的稳定性产生影响。
- 本文在重新设计回收僵尸进程的过程中，做到了无需依赖于母进程的等待操作这一点，但是却尚未考虑到取消母进程的等待操作是否会影响母进程对子进程的运行信息的收集这一点。今后，本人将通过不懈努力，做到这一点。

参考文献

[1] HALDAR S, ARAVIND A A. Operating Systems[M]. Pearson, 2010.

[2] Silberschatz, Galvin, Gagne. Operating System Concepts Essentials[M]. John Wiley & Sons, 2011.

[3] TANENBAUM A S. Modern Operating Systems[M]. 3rd ed. Prentice Hall Press, 2007.

[4] AIVAZIAN T. Linux Kernel 2.4 Internals[Z]. 2002.

[5] BHATTACHARJEE A, LUSTIG D. Architectural and Operating System Support for Virtual Memory[M]. Morgan & Claypool, 2017.

[6] MCCARTHY J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I[J]. Commun. ACM, 1960, 3(4): 184-195.

[7] Wikipedia contributors. Garbage collection (computer science) — Wikipedia, The Free Encyclopedia[Z]. 2018.

[8] MCCARTHY J. A Micro-manual for LISP - Not the Whole Truth[J]. SIGPLAN Not., 1978, 13(8): 215-216.

[9] Wikipedia contributors. Edsger W. Dijkstra — Wikipedia, The Free Encyclopedia[Z]. 2018.

[10] JOHNSON R E. Reducing the Latency of a Real-time Garbage Collector[J]. ACM Lett. Program. Lang. Syst., 1992, 1(1): 46-58.

[11] BOVET D, CESATI M. Understanding The Linux Kernel[M]. 3rd ed. OReilly, 2005.

[12] Wikipedia contributors. Zombie process — Wikipedia, The Free Encyclopedia[Z]. 2017.

[13] LOVE R. Linux Kernel Development[M]. Addison-Wesley, 2010.

[14] William, Eric, Harshadrai. Process management[Z]. 2006.

[15] BACH M. The design of the UNIX operating system[M]. Prentice-Hall, 1986.

[16] O'DONNELL M D. Orphan computer process identification[Z]. 1998.

[17] MACPHAIL M G. System and method for handling orphaned cause and effect objects[Z]. 2004.

[18] KROAH-HARTMAN G. Linux Kernel in a Nutshell: A Desktop Quick Reference[M]. O'Reilly, 2007.

[19] BROWNING L M. System and method to improve harvesting of zombie processes in an operating system[Z]. 2002.

[20] FAULKNER R A. Method and apparatus for non-damaging process debugging via an agent thread[Z]. 1997.

[21] PRICE D. Methods and apparatus for managing defunct processes[Z]. 2001.

[22] STALLINGS W. Operating Systems: Internals and Design Principles[M]. 7th ed. Prentice Hall, 2011.

[23] HORWOOD E. IBM Knowledge Center - What is an address space?[Z]. 2013.

[24] SOURCE K. The Linux Boot Protocol[Z]. 2005.

[25] DIKE J. A user-mode port of the Linux kernel.[C]. in: Annual Linux Showcase & Conference. 2000.

[26] FORD B, BACK G, BENSON G, et al. The Flux OSKit: A substrate for kernel and language research[C]. in: ACM SIGOPS Operating Systems Review: vol. 31: 5. 1997: 38-51.

[27] MAUERER W. Professional Linux Kernel Architecture[M]. John Wiley & Sons, 2008.

[28] Wikipedia contributors. Context switch — Wikipedia, The Free Encyclopedia[Z]. 2018.

[29] RUSLING D A. The Linux Kernel[M]. Linux Documentation Project, 1999.

[30] Wikipedia contributors. O(1) scheduler — Wikipedia, The Free Encyclopedia[Z]. 2018.

[31] Wikipedia contributors. Completely Fair Scheduler — Wikipedia, The Free Encyclopedia[Z]. 2018.

附录 A　主要代码清单

A.1　处理方式对比

A.1.1　waytest.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

void waytest1()
{
    int pid,pid1;
    pid=fork();
    if(pid==0)
        {
            printf(1,"child %d working\n......\n",getpid());
            sleep(200);
            printf(1,"child %d is exiting:\n",getpid());
            exit();
        }
    printf(1,"parent waiting for child to complete ......\n");
    pid1=wait();
    if(pid1>0)
        {
            printf(1,"waited child %d\n",pid1);
        }
    lszombies();
    printf(1,"parent doing its own work:\n......\n");
    sleep(300);
    printf(1,"parent's work done\n");

}

void waytest2()
{
    int pid;
    pid=fork();
    if(pid==0)
    {
      printf(1,"child %d exiting:\n",getpid());
      exit();
    }
```

```
38      lszombies();
39      printf(1,"parent doing its own work:\n......\n");
40      sleep(300);
41      printf(1,"parent's work done, calling wait:\n......\n");
42      wait();
43      lszombies();
44  }
45
46  int main(void)
47  {
48    printf(1,"Test1: parent waits zombies before doing its job\n");
49    waytest1();
50    printf(1,"\n_____________________________________________\n\n");
51    printf(1,"Test2: parent waits zombies after its job done:\n");
52    waytest2();
53    exit();
54  }
```

A.1.2 waytest1.c

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4
5   int main(void)
6   {
7     int pid;
8     pid=fork();
9     if(pid==0)
10    {
11      printf(1,"child %d is working\n......\n",getpid());
12      sleep(200);
13      printf(1,"child %d exiting!\n",getpid());
14      exit();
15    }
16    printf(1,"parent doing its own work\n......\n");
17    sleep(300);
18    printf(1,"parent's work is done\n");
19    lszombies();
20    exit();
21  }
```

A.2 资源回收速度对比

A.2.1 ztime.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

int test1(void)
{
    int pid;
    printf(1,"%d freepages in system before forked \n",getfreepages());
    pid=fork();
    if(pid==0)
        {
            printf(1,"%d freepages in system after
            ↪  forked\n",getfreepages());
            exit();
        }
    sleep(2);
    printf(1,"%d freepages in system after exited\n",getfreepages());
    wait();
    printf(1,"%d freepages in system after waited\n",getfreepages());
    return 1;
}

int test2(void)
{
    int pid,i,pid1;

    pid=fork();
    if(pid==0)
        {
            sleep(200);
            exit();
        }
    for(i=1;i<=10;i++)
        {
            pid1=fork();
            if(pid1==0){
                while(1);}
        }
    wait();
    return 1;
}

int main(void)
{
    int i;
    printf(1,"\nTest1: With only 1 proc in ready queue when child
    ↪  exited\n");
    test1();
    for(i=1;i<=10;i++){
```

```
48          printf(1,"\nTest2: With %d procs in ready queue when child
            ↪  exited\n",i*10);
49      test2();}
50      exit();
51  }
```

A.2.2 ztime1.c

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4
5   int test1(void)
6   {
7       int pid;
8       printf(1,"%d freepages in system before forked \n",getfreepages());
9       pid=fork();
10      if(pid==0)
11          {
12              printf(1,"%d freepages in system after
                ↪  forked\n",getfreepages());
13              exit();
14          }
15      sleep(2);
16      printf(1,"%d freepages in system after exited\n",getfreepages());
17    return 1;
18  }
19
20  int test2(void)
21  {
22      int pid,i,pid1;
23      pid=fork();
24      if(pid==0)
25    {
26      sleep(200);
27      exit();
28    }
29      for(i=1;i<=10;i++)
30          {
31              pid1=fork();
32              if(pid1==0){
33                  while(1);}
34          }
35    sleep(300);
36    return 1;
37  }
38
39  int main(void)
40  {
41    int i;
42      printf(1,"\nTest1: With only 1 proc in ready queue when child
        ↪  exited\n");
43      test1();
```

```
44    for(i=1;i<=10;i++){
45      printf(1,"\nTest2: With %d procs in ready queue when child
        ↪  exited\n",i*10);
46      test2();
47    }
48    exit();
49  }
```

A.3 安全性对比

Zattack.c 程序的代码:

```
1   #include"types.h"
2   #include"stat.h"
3   #include"user.h"
4
5   int Zattack(void)
6   {
7       int pid1,i;
8       for(i=1;i<=100;i++)
9           {
10              pid1=fork();
11              if(pid1<0)
12                  {   printf(1,"Forked %d procs\n",i);
13                      break;
14                  }
15              if(pid1==0)
16                  {
17                      exit();
18                  }
19          }
20      if(i>=100)
21          printf(1,"Fork claimes to work 100 times\n");
22      printf(1,"Parent in Zattack infinite looping\n");
23      while(1);
24      return 1;
25  }
26
27  int main(void)
28  {
29      int pid,pid1,i;
30      pid1=fork();
31      if(pid1==0)
32          {
33              printf(1,"Proc Zattack running:\n......\n");
34              Zattack();
35          }
36      sleep(500);
37      printf(1,"Trying to fork new procs every 2 seconds after Zattack
        ↪  proc\n");
38      for(i=1;i<=10;i++)
39          {
```

```
40              pid=fork();
41              if(pid<0)
42                  {
43                      printf(1,"Process table full,Fork failed\n");
44                  }
45              else if(pid==0)
46                  exit();
47              else
48                  printf(1,"Forked new proc %d\n",pid);
49              sleep(200);
50          }
51      exit();
52  }
```

个人简介

罗志兵（1989 ），女，广西贵港人，本科毕业于西南林业大学计算机与信息科学学院，现为在读硕士研究生，就读于西南林业大学大数据与智能工程学院，所学专业为林业信息工程。曾发表的论文有《基于动态规划的基因双序列比对研究》。

导师简介

赵家刚，硕士研究生、副教授，硕士研究生导师，云南省科技项目评审专家，云南省计算机协会理事，主要研究智能数据处理、信息系统开发。主持了多个项目，主编教材 3 部。

主讲的课程有《C 语言程序设计》、《面向对象程序设计》、《数据结构》、《图形图像程序设计》、《人工智能》、《人工神经网络》、《组件技术》、《离散数学》、《计算机编程导论》。

- 主要学习、工作经历：

  1978.9 - 1980.7，在腾冲师范读书；

  1980.8 - 1983.8，在腾冲固东中学任教；

  1983.9 - 1987.7，在云南师大数学系读本科，获理学士学位；

  1987.6 - 1998.8，在保山师专任计算机教学工作；

  1998.9 - 2001.6，在云南师大计科系读研究生，获理学硕士学位；

  2001.7 - 至今，在西南林学院计科系工作。主要从事软件开发的教学工作，培养了众多编程高手。

- 主要论著和文章：

[1] 赵家刚, 徐声远. ENT 环境下远程处理的教学方法探索 [J]. 西南林学院学报增刊, 2002.

[2] 赵家刚, 林毓材. 数据挖掘的关联规则研究 [J]. 计算机科学专刊, 2003.

[3] 赵家刚, 王红菘. 大学计算机程序设计 [M]. 中国科学技术出版社, 2007.

[4] ZHAO J, LI S. Prediction Model For Postfire Mortality of Pinus Yunnanensis in central Yunnan Province[J]. ICCAE 国际会议, 2009.

[5]　赵家刚. 枚举组合算法的研究与应用 [J]. 科技信息, 2009: pp430-431.

[6]　赵家刚. C 语言函数教学法探讨 [J]. 中国教育研究论, 2009: pp61-65.

[7]　赵家刚. 指针的学习与应用 [J]. 电脑编程技巧与维护, 2009: pp114-126.

[8]　赵家刚, 李俊𠀋. C 语言程序设计 [M]. 西南交通大学出版社, 2010.

[9]　赵家刚. C 语言指针教学中的知识点分析与总结 [J]. 计算机教育, 2011: pp55-61.

[10]　赵家刚, 狄光智, 吕丹桔. 计算机编程导论 – Python 程序设计 [M]. 人民邮电出版社, 2013.

- 主持的项目：
  1. 2003.6 - 2004.5，西南林学院，“数据结构教学改革研究”；
  2. 2007.5 - 2008.12，云南海诚集团，“项目建设工程数据库管理软件”，具有工作流；
  3. 2008.6 - 2009.6，昆明市科计局科技项目审批软件，具有工作流；
  4. 2010.10 - 2013.10，科学出版社大型教育类网站研发；
  5. 2013.10 - 2016.10，云南省委组织部办公邮件系统。

获得成果目录

[1] 罗志兵. 基于动态规划的基因双序列比对研究 [J]. 现代计算机, 2017(11): 28-33.

致　谢

本课题和论文的完成得到了许多老师和同学的支持和帮助，在这里，我将给予他们最衷心的感谢。在这里，感谢我的导师，赵家刚以及代飞老师的悉心指导和帮助，使得该论文能如期完成。从选择研究方向到确定具体的实施方案,从论文的初稿到最后的审定，无不花费了他大量的心血和精力。 赵老师和代老师的博学多才，和他们在科研学术方面的认真、严谨、求实、创新的态度， 以及开拓进取的工作作风都让我收益匪浅。