**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题目: 实验一 进程创建与管理**

**学 院 计算机学院**

**班 级 2021211304**

**学 号 2021212171、2021212484**

**姓 名 杨晨、张梓良**

**2023年 11月**

##### 1 实验内容

在 Linux 环境下，采用 C/C++/Java（或其它语言）编程，完成以下实验：

1. 参照 Linux 内核源码结构，阅读 Linux 内核源码，分析 Linux 进程的组成，观察 Linxu 进

程的 task\_struc 等进程管理数据结构；

2. 参照相关示例程序，查阅 Linux 系统调用等相关资料，设计父进程和子进程的业务处理

逻辑；利用 C++等高级程序设计语言和 fork()、clone、exec()、wait()、exit()、kill()、getpid()

等系统调用，创建管理进程，观察父子进程/线程的结构和并发行为，掌握等待、撤销等进

程控制方法。

要求：

* 至少用到 fork()、clone、exec、wait、exit、kill、getpid 等 6 个系统调用；
* 所创建的父子进程各自具有不同的业务处理逻辑，父进程创建子进程后，子进程通过exec()调入自身执行代码；
* 进程可以自己通过 exit()主动结束，也可以被父进程执行 kill()命令来结束；
* 观察对比通过 fork()和 clone()创建的子任务/进程/线程的差异，分析 clone()系统调用中设置与不设置 CLONE\_FS、CLONE\_VFORK、CLONE\_FILES、CLONE\_FS、CLONE\_PID等参数对所创建的子进程/线程的影响；
* 在创建的父子进程/线程代码中的不同位置处增加随机延迟，使得进程执行横跨多个时间片，如通过增加数十到数百毫秒的延迟，保证进程执行时间不少于 3 个时间片。

3. 掌握 ps、top、pstree –h、vmstat、strace、ltrace、sleep x、kill、jobs 等命令的功能和使用

方式，利用这些命令观察进程的行为和特征。

##### 2 实验 1-1. 进程观察

# **2.1 Linux进程组成**

Linux进程由程序代码、数据和一些元数据组成。以下是对Linux进程的组成部分的解释：

1. 程序代码：也称为“可执行文件”，是由编译器将源代码编译后生成的二进制文件。程序代码包含了程序的指令、常量、变量等信息。当进程创建时，它的程序代码会被加载到内存中，并被操作系统调度执行。
2. 数据：也称为“数据段”，包括已初始化（BSS）和未初始化（DATA）两部分。已初始化数据包含了程序中明确赋值的全局变量和静态变量等，未初始化数据则包含了未初始化的全局变量和静态变量等。数据在进程创建时就被分配到内存中，用于存储程序运行过程中的­­­数据。
3. 堆：堆是由程序动态分配的内存空间，用于存储程序运行期间需要的数据结构等。堆是由操作系统在进程启动时为其分配的，可以通过malloc等函数动态申请和释放。
4. 栈：栈是用于存储函数调用和局部变量等的内存空间。每当一个函数被调用时，栈就会向下增长，当函数返回时，栈就会恢复其上一个状态。栈的大小是固定的，而且是由操作系统在进程创建时为其分配的。
5. 文件描述符表：每个进程都有自己的文件描述符表，用于描述与其相关的文件和I/O设备等。当进程需要访问一个文件时，它会打开该文件并在文件描述符表中创建一个对应的记录。文件描述符表是通过操作系统进行管理和分配的。
6. 进程控制块（PCB）：每个进程都有自己的进程控制块，用于存储进程的各种元数据和状态信息，包括进程ID、进程状态、进程优先级、寄存器值、程序计数器等。PCB是操作系统用来管理进程的重要数据结构，用于存储和管理进程的状态信息。
7. 程序计数器（PC）：是一个特殊的寄存器，用于存储当前正在执行的指令的地址。在运行过程中，CPU会不断地从程序计数器中读取指令，并按顺序执行。当一条指令执行完毕后，程序计数器会自动递增，指向下一条要执行的指令的地址。程序计数器在进程切换时非常重要，它记录了下一条要执行的指令的位置。
8. 寄存器：是位于CPU内部的高速存储器，用于存储临时数据、地址以及操作数等。不同的寄存器拥有不同的功能和用途，如通用寄存器、标志寄存器、累加器、地址寄存器等。寄存器的使用可以提高程序的执行效率，因为它们比内存访问更快。

# **2.2 task\_struct**

task\_struct数据结构定义在/include/linux/sched.h文件中，部分代码截取如下：

1. struct task\_struct {
2. #ifdef CONFIG\_THREAD\_INFO\_IN\_TASK
3. */\**
4. \* For reasons of header soup (see current\_thread\_info()), this
5. \* must be the first element of task\_struct.
6. \*/
7. struct thread\_info  thread\_info;
8. #endif
9. unsigned int   \_\_state;
11. #ifdef CONFIG\_PREEMPT\_RT
12. */\* saved state for "spinlock sleepers" \*/*
13. unsigned int   saved\_state;
14. #endif
16. */\**
17. \* This begins the randomizable portion of task\_struct. Only
18. \* scheduling-critical items should be added above here.
19. \*/
20. randomized\_struct\_fields\_start
22. void    \*stack;
23. refcount\_t   usage;
24. */\* Per task flags (PF\_\*), defined further below: \*/*
25. unsigned int   flags;
26. unsigned int   ptrace;

task\_struct结构体中一些重要成员的定义如下：

**进程状态**

1. */\* Used in tsk->\_\_state: \*/*
2. #define TASK\_RUNNING   0x00000000
3. #define TASK\_INTERRUPTIBLE  0x00000001
4. #define TASK\_UNINTERRUPTIBLE  0x00000002
5. #define \_\_TASK\_STOPPED   0x00000004
6. #define \_\_TASK\_TRACED   0x00000008
7. */\* Used in tsk->exit\_state: \*/*
8. #define EXIT\_DEAD   0x00000010
9. #define EXIT\_ZOMBIE   0x00000020
10. #define EXIT\_TRACE   (EXIT\_ZOMBIE | EXIT\_DEAD)
11. */\* Used in tsk->\_\_state again: \*/*
12. #define TASK\_PARKED   0x00000040
13. #define TASK\_DEAD   0x00000080
14. #define TASK\_WAKEKILL   0x00000100
15. #define TASK\_WAKING   0x00000200
16. #define TASK\_NOLOAD   0x00000400
17. #define TASK\_NEW   0x00000800
18. #define TASK\_RTLOCK\_WAIT  0x00001000
19. #define TASK\_FREEZABLE   0x00002000
20. #define \_\_TASK\_FREEZABLE\_UNSAFE        (0x00004000 \* IS\_ENABLED(CONFIG\_LOCKDEP))
21. #define TASK\_FROZEN   0x00008000
22. #define TASK\_STATE\_MAX   0x00010000
23. #define TASK\_ANY   (TASK\_STATE\_MAX-1)

系统中的每个进程都必然处于以上所列进程状态中的一种。

**进程标识符（PID）**

1. #define PID\_MAX\_DEFAULT (CONFIG\_BASE\_SMALL ? 0x1000 : 0x8000)
2. pid\_t    pid;
3. pid\_t    tgid;

在CONFIG\_BASE\_SMALL配置为0的情况下，PID的取值范围是0到32767，即系统中的进程数最大为32768个。

在Linux系统中，一个线程组中的所有线程使用和该线程组的领头线程（该组中的第一个轻量级进程）相同的PID，并被存放在tgid成员中。只有线程组的领头线程的pid成员才会被设置为与tgid相同的值。注意，**getpid()系统调用返回的是当前进程的tgid值而不是pid值**。

**进程内核栈**

1. void \*stack;

进程通过alloc\_thread\_info函数分配它的内核栈，通过free\_thread\_info函数释放所分配的内核栈。

**进程调度**

1. #define MAX\_USER\_RT\_PRIO 100
2. #define MAX\_RT\_PRIO MAX\_USER\_RT\_PRIO
3. #define MAX\_PRIO (MAX\_RT\_PRIO + 40)
4. #define DEFAULT\_PRIO (MAX\_RT\_PRIO + 20)
5. int prio, static\_prio, normal\_prio;
6. unsigned int rt\_priority;
7. const struct sched\_class \*sched\_class;
8. struct sched\_entity se;
9. struct sched\_rt\_entity rt;
10. unsigned int policy;
11. cpumask\_t cpus\_allowed;

static\_prio用于保存静态优先级，可以通过nice系统调用来进行修改。rt\_priority用于保存实时优先级。normal\_prio的值取决于静态优先级和调度策略。prio用于保存动态优先级。实时优先级范围是0到MAX\_RT\_PRIO-1（即99），而普通进程的静态优先级范围是从MAX\_RT\_PRIO到MAX\_PRIO-1（即100到139）。值越大静态优先级越低。

policy表示进程的调度策略，目前主要有以下五种：

1. #define SCHED\_IDLE 5
2. #define SCHED\_NORMAL 0
3. #define SCHED\_FIFO 1
4. #define SCHED\_RR 2
5. #define SCHED\_BATCH 3

SCHED\_NORMAL用于普通进程，通过CFS调度器实现。SCHED\_BATCH用于非交互的处理器消耗型进程。SCHED\_IDLE是在系统负载很低时使用。SCHED\_FIFO（先入先出调度算法）和SCHED\_RR（轮流调度算法）都是实时调度策略。

**进程地址空间**

1. struct mm\_struct \*mm, \*active\_mm;
2. #ifdef CONFIG\_COMPAT\_BRK
3. unsigned brk\_randomized:1;
4. #endif
5. #if defined(SPLIT\_RSS\_COUNTING)
6. struct task\_rss\_stat rss\_stat;
7. #endif

mm指向进程所拥有的内存描述符，而active\_mm指向进程运行时所使用的内存描述符。对于普通进程而言，这两个指针变量的值相同。但是，内核线程不拥有任何内存描述符，所以它们的mm成员总是为NULL。当内核线程得以运行时，它的active\_mm成员被初始化为前一个运行进程的active\_mm值。brk\_randomized用来确定对随机堆内存的探测。rss\_stat用来记录缓冲信息。

**时间**

1. cputime\_t utime, stime, utimescaled, stimescaled;
2. cputime\_t gtime;
3. *#ifndef CONFIG\_VIRT\_CPU\_ACCOUNTING*
4. cputime\_t prev\_utime, prev\_stime;
5. *#endif*
6. unsigned long nvcsw, nivcsw; /\* context switch counts /
7. struct timespec start\_time; / monotonic time /
8. struct timespec real\_start\_time; / boot based time /
9. struct task\_cputime cputime\_expires;
10. struct list\_head cpu\_timers[3];
11. *#ifdef CONFIG\_DETECT\_HUNG\_TASK*
12. / hung task detection /
13. unsigned long last\_switch\_count;
14. *#endif*
15. cputime\_t utime, stime, utimescaled, stimescaled;
16. cputime\_t gtime;
17. *#ifndef CONFIG\_VIRT\_CPU\_ACCOUNTING*
18. cputime\_t prev\_utime, prev\_stime;
19. *#endif*
20. unsigned long nvcsw, nivcsw; / context switch counts /
21. struct timespec start\_time; / monotonic time /
22. struct timespec real\_start\_time; / boot based time /
23. struct task\_cputime cputime\_expires;
24. struct list\_head cpu\_timers[3];
25. *#ifdef CONFIG\_DETECT\_HUNG\_TASK*
26. / hung task detection \*/
27. unsigned long last\_switch\_count;
28. *#endif*

utime/stime用于记录进程在用户态/内核态下所经过的节拍数（定时器）。prev\_utime/prev\_stime是先前的运行时间。utimescaled/stimescaled也是用于记录进程在用户态/内核态的运行时间，但它们以处理器的频率为刻度。gtime是以节拍计数的虚拟机运行时间（guest time）。nvcsw/nivcsw是自愿（voluntary）/非自愿（involuntary）上下文切换计数。last\_switch\_count是nvcsw和nivcsw的总和。

start\_time和real\_start\_time都是进程创建时间，real\_start\_time还包含了进程睡眠时间，常用于/proc/pid/stat。cputime\_expires用来统计进程或进程组被跟踪的处理器时间，其中的三个成员对应着cpu\_timers[3]的三个链表。

**信号处理**

1. struct signal\_struct \*signal;
2. struct sighand\_struct \*sighand;
3. sigset\_t blocked, real\_blocked;
4. sigset\_t saved\_sigmask; */\* restored if set\_restore\_sigmask() was used \*/*
5. struct sigpending pending;
6. unsigned long sas\_ss\_sp;
7. size\_t sas\_ss\_size;
8. int (\*notifier)(void \*priv);
9. void \*notifier\_data;
10. sigset\_t \*notifier\_mask;

signal指向进程的信号描述符。sighand指向进程的信号处理程序描述符。blocked表示被阻塞信号的掩码，real\_blocked表示临时掩码。pending存放私有挂起信号的数据结构。sas\_ss\_sp是信号处理程序备用堆栈的地址，sas\_ss\_size表示堆栈的大小。设备驱动程序常用notifier指向的函数来阻塞进程的某些信号（notifier\_mask是这些信号的位掩码），notifier\_data指的是notifier所指向的函数可能使用的数据。

##### 3 实验 1-2. 进程创建与管理

# **3.1 进程管理系统调用**

**fork()**

fork()函数创建一个新进程。

返回值：正确返回时，等于0表示创建子进程，从子进程返回的ID值；大于0表示从父进程返回的子进程的进程ID值。错误返回时，等于-1表示创建失败。

**clone()**

clone()用于创建 Linux 线程。创建时可以通过多个参数，指定新的命名空间（namespace），有选择地继承父进程的资源（如虚拟内存空间），将创建的子进程指定为父进程的兄弟进程。当创建出的子进程与父进程共享同一虚拟内存空间时，子进程被认为是线程。

调用格式：int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg)

如果该调用执行成功，所创建的子进程的 threadID 将返回给调用者；如果该调用执行失败，’-1’将返回给调用者。

**exec()**

exec()函数用于将一个新的程序代码调入本进程所占的内存，并将其覆盖，产生新的内存进程映像，新的程序可以是可执行文件或Shell批命令。

调用格式：

int execl(path,argo,…,argn,(char\*)0)

char \*path,\*argo,…\*argn；

int execv(path,argy)

char \*path,\*argv[]；

int execle(path,arg0,…,argn,(char \*)0,envp)

char \*path,\*argo,…,\*argn,\*envp[]；

int execve(path,argv,envp)

char \*path,\*argv[],\*envp[]；

int execvp(file,argv)

char \*file,\*argv[]；

说明：exec()是一个系统调用族，包括 execl、execv、execle、execve、execvp 等。exec函数族的函数执行成功后不会返回，只有调用失败了，它们才会返回一个-1。

**execl：**

函数原型：int execl(const char \*path, const char \*arg0, ..., (char \*)NULL);

需要提供可执行文件的完整路径。

需要逐个指定命令行参数，以 NULL 结束。

不传递环境变量给新进程。

**execlp：**

函数原型：int execlp(const char \*file, const char \*arg0, ..., (char \*)NULL);

**在系统的环境变量 PATH 中搜索可执行文件。**

**只需提供可执行文件的名称，无需提供完整路径。**

需要逐个指定命令行参数，以 NULL 结束。

不传递环境变量给新进程。

**execle：**

函数原型：int execle(const char \*path, const char \*arg0, ..., (char \*)NULL, char \*const envp[]);

需要提供可执行文件的完整路径。

需要逐个指定命令行参数，以 NULL 结束。

需要通过 envp 参数传递环境变量给新进程。

**execv：**

函数原型：int execv(const char \*path, char \*const argv[]);

需要提供可执行文件的完整路径。

需要通过 argv 参数传递命令行参数给新进程。

不传递环境变量给新进程。

**execve：**

函数原型：int execve(const char \*path, char \*const argv[], char \*const envp[]);

需要提供可执行文件的完整路径。

需要通过 argv 参数传递命令行参数给新进程。

需要通过 envp 参数传递环境变量给新进程。

**execvp：**

函数原型：int execvp(const char \*file, char \*const argv[]);

**在系统的环境变量 PATH 中搜索可执行文件。**

**只需提供可执行文件的名称，无需提供完整路径。**

需要通过 argv 参数传递命令行参数给新进程。

不传递环境变量给新进程。

**wait()**

wait()用于控制父进程与子进程的同步。父进程调用wait()函数，则父进程被阻塞，进入等待队列，等待子进程结束。当子进程结束时，会产生一个终止状态字，系族会向父进程发出SIGCHLD信号。当接到信号后，父进程提取子进程的终止状态字，从wait()函数返回继续执行原程序。

调用格式：(pid t)wait(int \*statloc)；

返回值：正确返回时，大于0表示子进程的进程ID值；等于0表示其他。错误返回时，等于-1 表示调用失败。

**exit()**

exit()用于结束进程自身。在进程正常结束时，exit()函数返回进程结束状态。在主程序

main()函数中调用return，实际是调用exit()函数。

调用格式：void exit(int status)；其中，status 为进程结束状态。

**kill()**

kill()函数用来向进程发送一个信号。

调用格式：kill(int pid，int sig)；

如果参数pid是正数，那么该调用将信号sig发送到进程号为pid的进程。如果pid等于0，那么信号sig将发送给当前进程所属进程组里的所有进程。如果参数pid等于-1，信号sig将发送给除了进程1和自身以外的所有进程。如果参数pid小于- 1，信号sig将发送给属于进程组-pid的所有进程。如果参数sig为0，将不发送信号。该调用执行成功时，返回值为0；错误时，返回-1，并设置相应 的错误代码errno。

**signal()**

signal()函数支持进程间基于信号signal（也称为软中断）的通信机制，允许调用进程控制软中断信号的处理。

调用格式： void(\*func)(); signal(sig，function)；

**getpid()**

获取目前进程的进程标识 pid 。

调用格式：int getpid()；

# **3.2 fork, wait, exit, getpid系统调用实例**

**测试代码**

1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <unistd.h>
4. #include <sys/wait.h>
5. int main(int argc, char \*\*argv)
6. {
7. int child\_pid, parent\_pid, pid;
8. int status;
9. if ((pid = fork()) > 0)
10. {
11. *// 父进程处理过程*
12. child\_pid = wait(&status);
13. printf("I am the parent process.\n");
14. if (pid == child\_pid)
15. printf("My child process has finished, and its process id = %d, which is equal to fork() retrun value.\n", child\_pid);
16. printf("My child process's return code is %d.\n", status);
17. }
18. else if (pid == 0)
19. {
20. *// 子进程处理过程*
21. printf("I am the child process.\n");
22. child\_pid = getpid();
23. printf("My process id = %d.\n", child\_pid);
24. exit(0);
25. }
26. else
27. {
28. printf("fork error\n");
29. exit(1);
30. }
31. }

该程序，演示了fork()，wait()和getpid()系统调用的使用。这些系统调用分别用于创建新的进程，等待子进程完成，以及获取进程ID。

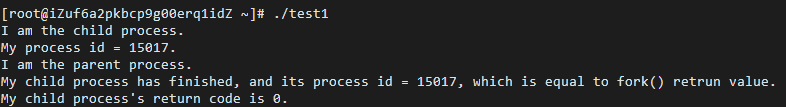
main()函数首先声明了几个整数变量：child\_pid，parent\_pid，pid和status。然后使用fork()系统调用创建一个新的进程。fork()的返回值存储在pid中。

如果fork()返回一个正值，这意味着当前进程是父进程。父进程然后使用wait()系统调用等待子进程完成，wait()返回子进程的进程ID和返回代码。父进程打印消息表明它是父进程，并检查fork()返回的进程ID是否与wait()返回的进程ID相同。如果相同，打印一条消息，表明子进程已经完成并显示其进程ID。同时也打印子进程的返回代码。

如果fork()返回0，这意味着当前进程是子进程。子进程打印消息表明它是子进程，并使用getpid()系统调用获取其进程ID，并打印其进程ID。然后它以返回代码0退出。

如果fork()返回一个负值，这表示在fork操作期间发生了错误。程序打印一个错误消息，并以返回代码1退出。

**运行结果**

****

# **3.3 exec系列系统调用实例**

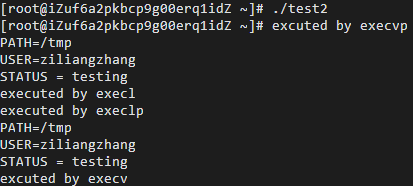
**测试代码**

1. #include <unistd.h>
2. int main(int arge, char \*argv[])
3. {
4. char \*envp[] = {"PATH=/tmp", "USER=ziliangzhang", "STATUS = testing", NULL};
5. char \*argv\_execv[] = {"echo", "excuted by execv", NULL};
6. char \*argv\_execvp[] = {"echo", "excuted by execvp", NULL};
7. char \*argv\_execve[] = {"env", NULL};
8. if (fork() == 0)
9. {
10. if (execl("/bin/echo", "echo", "executed by execl", NULL) < 0)
11. perror("Err on execl");
12. exit(0);
13. }
14. if (fork() == 0)
15. {
16. if (execlp("echo", "echo", "executed by execlp", NULL) < 0)
17. perror("Err on execlp");
18. exit(0);
19. }
20. if (fork() == 0)
21. {
22. if (execle("/usr/bin/env", "env", NULL, envp) < 0)
23. perror("Err on execle");
24. exit(0);
25. }
26. if (fork() == 0)
27. {
28. if (execv("/bin/echo", argv\_execv) < 0)
29. perror("Err on execv");
30. exit(0);
31. }
32. if (fork() == 0)
33. {
34. if (execvp("echo", argv\_execvp) < 0)
35. perror("Err on execvp");
36. exit(0);
37. }
38. if (fork() == 0)
39. {
40. if (execve("/usr/bin/env", argv\_execve, envp) < 0)
41. perror("Err on execve");
42. exit(0);
43. }
45. return 0;
46. }

该程序通过创建子进程和调用不同的 exec 函数，实现了以下操作：

* 使用 execl 函数执行 /bin/echo 程序，并传递参数。
* 使用 execlp 函数执行 echo 程序，并传递参数。
* 使用 execle 函数执行 /usr/bin/env 程序，并传递参数和自定义的环境变量。
* 使用 execv 函数执行 /bin/echo 程序，并传递参数数组。
* 使用 execvp 函数执行 echo 程序，并传递参数数组。
* 使用 execve 函数执行 /usr/bin/env 程序，并传递参数数组和自定义的环境变量。

**运行结果**

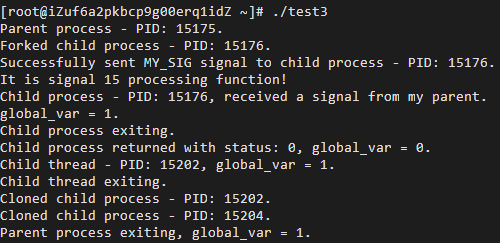
****

# **3.4 signal, kill, clone系统调用实例**

**测试代码**

1. #define \_GNU\_SOURCE
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <signal.h>
8. #include <sched.h>
9. #define MY\_SIG 15
10. int global\_var = 0;
11. void func()
12. {
13. printf("It is signal %d processing function!\n", MY\_SIG);
14. }
15. void childProcessLogic()
16. {
17. sleep(3); *// 增加延迟*
18. printf("Child process - PID: %d, received a signal from my parent.\n", getpid());
19. global\_var++;
20. printf("global\_var = %d.\n", global\_var);
21. printf("Child process exiting.\n");
22. exit(0);
23. }
24. void childThreadLogic()
25. {
26. global\_var++;
27. printf("Child thread - PID: %d, global\_var = %d.\n", getpid(), global\_var);
28. sleep(3); *// 增加延迟*
29. printf("Child thread exiting.\n");
30. exit(0);
31. }
32. int main()
33. {
34. pid\_t pid;
35. int status;
36. *// 注册信号处理函数*
37. signal(MY\_SIG, func);
38. *// 使用 fork() 创建子进程*
39. pid = fork();
40. if (pid == 0)
41. {
42. childProcessLogic();
43. }
44. else if (pid > 0)
45. {
46. printf("Parent process - PID: %d.\n", getpid());
47. printf("Forked child process - PID: %d.\n", pid);
48. *// 使用 kill() 命令结束子进程*
49. int result = kill(pid, MY\_SIG);
50. if (result == 0)
51. {
52. printf("Successfully sent MY\_SIG signal to child process - PID: %d.\n", pid);
53. }
54. else
55. {
56. perror("kill");
57. }
58. *// 使用 wait() 等待子进程结束*
59. wait(&status);
60. printf("Child process returned with status: %d, global\_var = %d.\n", WEXITSTATUS(status), global\_var);
61. *// 使用 clone() 创建子进程/线程*
62. *// mask 参数的取值：*
63. *// CLONE\_VM：子进程与父进程运行于相同的内存空间中*
64. *// CLONE\_VFORK：父进程被挂起*
65. *// CLONE\_CHILD\_SETTID：子进程与父进程共享相同的线程群*
66. int mask = CLONE\_VM | CLONE\_VFORK | CLONE\_CHILD\_SETTID;
67. pid = clone(childThreadLogic, malloc(4096) + 4096, mask, NULL);
68. if (pid == -1)
69. {
70. perror("clone");
71. }
72. else
73. {
74. printf("Cloned child process - PID: %d.\n", pid);
75. }
76. mask = CLONE\_VFORK | CLONE\_CHILD\_SETTID;
77. pid = clone(childThreadLogic, malloc(4096) + 4096, mask, NULL);
78. if (pid == -1)
79. {
80. perror("clone");
81. }
82. else
83. {
84. printf("Cloned child process - PID: %d.\n", pid);
85. }
86. waitpid(pid, NULL, 0);
87. printf("Parent process exiting, global\_var = %d.\n", global\_var);
88. }
89. else
90. {
91. printf("Fork failed.\n");
92. return 1;
93. }
94. return 0;
95. }

**运行结果**

****

通过运行结果可知：

clone 和 fork 的区别：

（1）clone 需要通过参数传入一个函数，作为子进程的执行体，在创建的子进程中执行；

（2）clone 不复制父进程的栈空间，而是通过第 2 个参数（void \*child\_stack）为创建的子进程分配一个新的栈空间，需要分配栈指针的空间大小。因此，clone 不再是简单地继承或者复制父进程；

（3）fork 函数创建的子进程完全拷贝父进程，clone 则是有选择性地拷贝。

同时clone()系统调用中flags参数的值对所创建的子进程/线程有所影响，例如设置从CLONE\_VM | CLONE\_VFORK | CLONE\_CHILD\_SETTID 变为CLONE\_VFORK | CLONE\_CHILD\_SETTID会导致子进程与父进程运行于不同的内存空间中，此时clone操作就类似于fork操作，这也是为什么第二次clone的子进程中对global\_var进行了修改但父进程中的global\_var的值不变，仍为1。

##### 4 实验 1-3. 使用ps、top等命令观察进程行为和特征

# **4.1 相应命令的功能和使用方式**

**ps 命令：**

功能：显示当前系统中运行的进程信息。

使用方式：ps [options]

常用选项：

-e：显示所有进程。

-f：显示完整格式的输出。

-l：使用长格式显示输出。

-u <username>：显示特定用户的进程。

**top 命令：**

功能：实时显示系统中进程的资源占用情况。

使用方式：top

常用操作：

q：退出 top。

k：发送信号给选定的进程。

r：修改进程的优先级。

1：切换到单核心模式。

**pstree -h 命令：**

功能：以树状结构显示进程之间的关系。

使用方式：pstree -h

**vmstat 命令：**

功能：显示虚拟内存、进程、CPU 和 I/O 等系统统计信息。

使用方式：vmstat [options] [delay [count]]

常用选项：

-a：显示活动和非活动内存的状态。

-d：显示磁盘活动统计信息。

-p：显示分区的虚拟内存统计信息。

**strace 命令：**

功能：跟踪并打印出进程执行的系统调用和接收的信号。

使用方式：strace [options] command [args]

常用选项：

-p <pid>：跟踪指定进程的系统调用。

-e <event>：指定要跟踪的系统调用事件。

**ltrace 命令：**

功能：跟踪并打印出进程执行的库函数调用。

使用方式：ltrace [options] command [args]

常用选项：

-p <pid>：跟踪指定进程的库函数调用。

-e <library>：指定要跟踪的库函数。

**sleep x 命令：**

功能：暂停当前进程的执行一段时间（x 表示秒数）。

使用方式：sleep x

**kill 命令：**

功能：向指定进程发送信号，用于终止、停止或控制进程。

使用方式：kill [options] <pid>

常用选项：

-9：强制终止进程。

**jobs 命令：**

功能：显示当前 shell 会话中的作业列表。

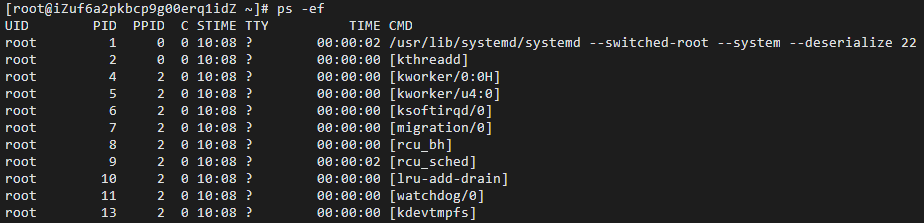
使用方式：jobs [options]

常用选项：

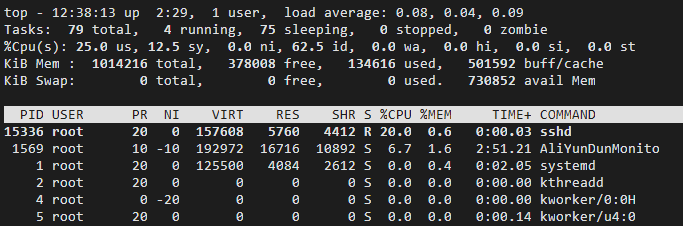
-l：显示作业的 PID 和状态。

# **4.2 使用实例**

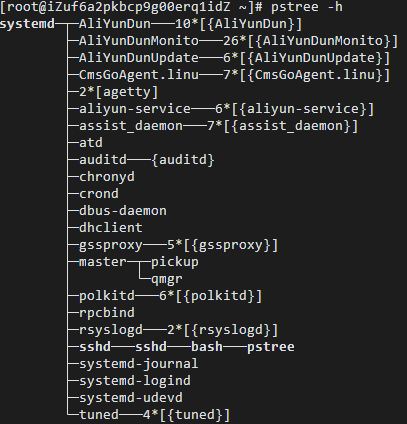
**ps**



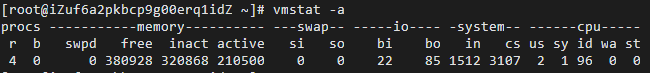
**top**

****

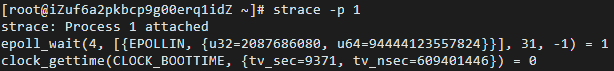
**pstree -h**

****

**vmstat**

****

**strace**

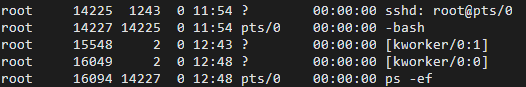
****

**sleep x**

****

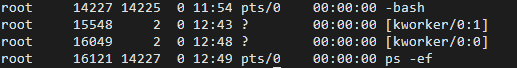
**kill**

kill前：

****

****

kill后：

****