Lab2-Linux进程控制_wait()与exit()

学号	姓名	专业班级	课程名称	学期	任课教师	完成日期
23020011046	翟一航	软件工程23	操作系统	2025秋	李晓慧	12.29

一、实验目的及要求

- 了解进程与程序的区别，加深对进程概念的理解；
- 进一步认识进程并发执行的原理，理解进程并发执行的特点，区别进程并发执行与顺序执行；
- 分析进程争用临界资源的现象，学习解决进程互斥的方法；
- 学习wait()和waitpid()的函数使用原理和范例。
- 学习exit()函数使用原理和范例。

二、实验内容与源码

实验内容一：wait()

1. `wait` （等待子进程中断或结束）：

   进程一旦调用了 `wait` ，就立即阻塞自己，由 `wait` 自动分析是否当前进程的某个子进程已经退出

   - 如果找到了一个僵尸子进程， `wait` 就会收集这个子进程的信息，并把它彻底销毁后返回；
   - 如果没有找到， `wait` 就会一直阻塞。

2. 表头文件及定义函数

```c
# include<sys/types.h>/* 提供类型pid_t的定义 */
# include<sys/wait.h>

  pid_t wait (int * status);
```

3. 函数说明

   `wait ()` 会暂时停止目前进程的执行，直到有信号来到或子进程结束。

   - 如果在调用 `wait ()` 时子进程已经结束，则 `wait ()` 会立即返回子进程结束状态值。
     - 子进程的结束状态值会由参数 `status` 返回，而子进程的进程识别码也会同时返回。
     - 参数 `status` 用来保存并收集进程退出时的一些状态，它是一个指向 `int` 类型的指针。
     - 如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，我们就可以设定这个参数为 `NULL` ， `pid = wait(NULL)` 。返回值如果执行成功 `wait` 会返回被收集的子进程的进程 `ID` 。
   - 如果调用进程没有子进程，调用就会失败，此时 `wait` 返回 `-1` ，同时 `errno` 被置为 `ECHILD` 。

示例一：简单使用 wait() 函数处理僵尸进程

源码（修改程序中的小错误，并在sleep前后添加一个计时器，输出实际睡眠时间）

```c
/* wait1.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main()
{
    pid_t pc, pr;

    pc = fork();
    if(pc < 0) {       /* 如果出错 */
        printf("error ocurred!\n");
        exit(1);
    }
    else if(pc == 0){    /* 如果是子进程 */
        printf("This is child process with pid of %d\n", getpid());

        /* 在sleep前后添加计时器 */
        time_t start, end;
        double elapsed;

        time(&start);  // 记录sleep开始时间
        printf("Child: Start sleeping at %s", ctime(&start));

        sleep(10);  /* 睡眠10秒钟 */

        time(&end);    // 记录sleep结束时间
        elapsed = difftime(end, start);

        printf("Child: Finished sleeping at %s", ctime(&end));
        printf("Child: Actually slept for %.2f seconds\n", elapsed);

        exit(0);  // 子进程结束
    }
    else{            /* 如果是父进程 */
        pr = wait(NULL);  /* 在这里等待 */
        printf("I catched a child process with pid of %d\n", pr);
    }

    return 0;
}
```

实例二：使用 wait() 提取子进程退出状态

status参数

- WIFEXITED(status)
  - 这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值。
  - 虽然名字一样，这里的参数status并不同于wait唯一的参数--指向整数的指针status，而是那个指针所指向的整数。
- WEXITSTATUS(status)
  - 当WIFEXITED返回非零值时，我们可以用这个宏来提取子进程的返回值，如果子进程调用exit(5)退出，WEXITSTATUS(status) 就会返回5；如果子进程调用exit(7)，WEXITSTATUS(status)就会返回7。
  - 请注意，如果进程不是正常退出的，也就是说，WIFEXITED返回0，这个值就毫无意义

源码

```c
# include <sys/wait.h>
# include <unistd.h>
# include <stdlib.h>
# include <stdio.h>
int main()
{
    int status;
    pid_t pc,pr;
    pc=fork();
    if(pc<0)    /* 如果出错 */
        printf("error ocurred!\n");
    else if(pc==0){ /* 子进程 */
        printf("This is child process with pid of %d.\n",getpid());
        exit(3);    /* 子进程返回3 */
    }
    else{       /* 父进程 */
        pr=wait(&status);
        if(WIFEXITED(status)){  /* 如果WIFEXITED返回非零值 */
            printf("the child process %d exit normally.\n",pr);
            printf("the return code is %d.\n",WEXITSTATUS(status));
        }else           /* 如果WIFEXITED返回零 */
            printf("the child process %d exit abnormally.\n",pr);
    }
}
```

Task—练习

源码

```c
# include <stdlib.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <stdio.h>
# include <stdlib.h>
int main()
```

```c
{
  pid_t pid;
  int status,i;
  if(fork()==0)
  {
    printf("This is the child process and pid =%d\n", getpid());
    exit(5);
  }
  else{
    sleep(1);
    printf("This is the parent process ,wait for child...\n");
    pid = wait(&status);
    i = WEXITSTATUS(status);
    printf("child's pid =%d exit and status=%d\n",pid,i);
  }
}
```

实验内容二：waitpid()

1. 表头文件及定义函数

```c
# include<sys/types.h>/* 提供类型pid_t的定义 */
# include<sys/wait.h>

  pid_t waitpid(pid_t pid,int * status,int options);
```

2. 函数说明
   - 从本质上讲，系统调用 `waitpid` 和 `wait` 的作用是完全相同的
   - `waitpid` 多了两个可由用户控制的参数 `pid` 和 `options`
     - `pid`
       - `pid>0` 时，只等待进程 `ID` 等于 `pid` 的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束， `waitpid` 就会一直等下去
       - `pid=-1` 时，等待任何一个子进程退出，没有任何限制，此时 `waitpid` 和 `wait` 的作用一模一样
       - `pid=0` 时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，`waitpid` 不会对它做任何理睬。
       - `pid<-1` 时，等待一个指定进程组中的任何子进程，这个进程组的 `ID` 等于 `pid` 的绝对值
     - `options` ：可以为0 或 OR 组合

Task—练习

源码

```c
/* waitpid.c */
```

```c
# include <sys/types.h>
# include <sys/wait.h>
# include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    pid_t pc, pr;

    pc=fork();
    if(pc<0)        /* 如果fork出错 */
        printf("Error occured on forking.\n");
    else if(pc==0){     /* 如果是子进程 */
        sleep(10);  /* 睡眠10秒 */
        exit(0);
    }
    /* 如果是父进程 */
    do{
        pr=waitpid(pc, NULL, WNOHANG);  /* 使用了WNOHANG参数，waitpid不会在这里等待 */
        if(pr==0){          /* 如果没有收集到子进程 */
            printf("No child exited\n");
            sleep(1);
        }
    }while(pr==0);              /* 没有收集到子进程，就回去继续尝试 */
    if(pr==pc)
        printf("successfully get child %d\n", pr);
    else
        printf("some error occured\n");
}
```

实验内容三：exit()

1. 表头文件及函数定义

```c
# include<stdlib.h>

  void exit(int status);
```

2. 函数说明

   `exit` 系统调用带有一个整数类型的参数 `status` ，可利用这个参数传递进程结束时的状态

   - 0 表示正常结束；
   - 其他的数值表示出现了错误，进程非正常结束。
   - 理论上， `exit` 可以返回小于 `256` 的任何整数。
   - 返回的不同数值给调用者作相应处理。
     - 单进程返回给操作系统。
     - 多进程返回给父进程。父进程里面调用 `waitpid()` 等函数得到子进程退出的状态，以便作不同处

理。

- 总的说来，`exit ()` 就是当前进程把控制权返回给调用该程序的程序，括号里的是返回值，告诉调用程序该程序的运行状态。在实际编程时，可以用 `wait` 系统调用接收子进程的返回值。

3. 示例

```c
/* exit_test1.c */
# include<stdio.h>
# include<stdlib.h>
int main()
{
    printf("this process will exit!\n");
    exit(0);
    printf("never be displayed!\n");
}
```

运行结果

```
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./exit_test1
this process will exit!
```

程序运行后，打印出 "This process will exit!"，并终止进程，未显示后续的 "This line will never be displayed."。

分析

`exit(0)` 终止当前进程，清理内存和 PCB 数据结构，返回状态码 `0` 表示正常退出。调用 `exit(0)` 后，未执行的代码不会继续运行。通过 `exit(0)`，进程停止并返回控制权给父进程或操作系统。"This line will never be displayed." 未打印，表明代码在 `exit()` 处终止。

Task—练习1

- 比较 `exit()` 与 `_exit()` 函数的区别
- 搜集资料，整理形成实验报告

```c
/* exit2.c */
# include <stdio.h>
# include <stdlib.h>
int main()
{
    printf("output begin\n");
    printf("content in buffer");
    exit(0);
}
```

```c
/* _exit1.c */
# include <stdio.h>
# include <unistd.h>
int main()
{
    printf("output begin\n");
    printf("content in buffer");
    _exit(0);
}
```

Task—练习2

- 比较 `exit()` 与 `return 0` 的区别
- 搜集资料，整理形成实验报告

三、实验结果与分析

实验内容一：wait()

示例一：简单使用 wait() 函数处理僵尸进程

运行结果

```
Documents/OS/Lab2
❯ vim wait1.c
❯
Documents/OS/Lab2 via C v17.0.0-clang took 12s
❯ gcc wait1.c -o wait1
❯
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./wait1
This is child process with pid of 29756
Child: Start sleeping at Mon Dec 29 09:18:17 2025
Child: Finished sleeping at Mon Dec 29 09:18:27 2025
Child: Actually slept for 10.00 seconds
I catched a child process with pid of 29756
```

分析

根据计时器的输出可以注意到，在第2行结果打印出来前有10 秒钟的等待时间，这就是我们设定的让子进程睡眠的时间，只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。其实这里我们不管设定子进程睡眠的时间有多长，父进程都会一直等待下去子进程在运行时打印了自身的 PID 并进入休眠状态。父进程通过调用 `wait()` 阻塞自身，直到子进程退出。子进程结束后，父进程打印捕获到的子进程 PID。

示例二：使用 wait() 提取子进程退出状态

运行结果

```
Documents/OS/Lab2 via C v17.0.0-clang took 10s
❯ vim wait2.c

Documents/OS/Lab2 via C v17.0.0-clang took 3s
❯ gcc wait2.c -o wait2

Documents/OS/Lab2 via C v17.0.0-clang
❯ ./wait2
This is child process with pid of 29861.
the child process 29861 exit normally.
the return code is 3.
```

分析

父进程通过 `WIFEXITED` 确认子进程是否正常退出。

子进程的退出状态码（3）被成功提取并打印。

Task—练习

运行结果

```
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./task1
This is the child process and pid =30307
This is the parent process ,wait for child...
child's pid =30307 exit and status=5
```

可以发现，子进程打印自身的 PID 后退出，返回值为 5。父进程通过 `wait()` 获取子进程的退出状态，并正确提取返回值。

分析

子进程先运行并退出，父进程通过调用 `wait()` 同步执行，避免了进程竞争或僵尸进程问题。父进程从 `status` 中提取的退出码 `5`，与子进程调用 `exit(5)` 返回的值一致。如果父进程调用 `wait()` 时无子进程，则返回值为 `-1`，并设置 `errno` 为 `ECHILD`，在我们的程序中，子进程成功结束，返回值为5，因此使用 `wait()` 可以得到子进程的 PID。这种设计提高了函数的鲁棒性，能够有效处理异常情况。在复杂系统中，多个子进程可能并行执行，父进程通过 `wait()` 管理子进程的生命周期，确保系统稳定运行。 `wait()` 的阻塞特性避免了父进程继续执行可能导致的错误逻辑。

实验内容二：waitpid()

Task—练习

运行结果

```
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./task2
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 30527
```

可以观察到，开始时由于子进程睡眠，父进程调用 `waitpid()` 未收集到子进程，返回值为 0，并打印 `"No child exited"` 。随后父进程进入短暂休眠（1 秒），继续尝试收集子进程。

子进程睡眠 10 秒后退出，父进程成功捕获子进程的 PID，并打印 `"Successfully got child 30527"` 。

分析

使用 `WNOHANG` 参数后， `waitpid()` 即使未收集到子进程也会立即返回，不会阻塞父进程的运行。这种特性适用于父进程需要执行其他任务的场景，例如定时检查子进程状态。参数 `pid` 为 `pc` ，表示仅等待特定的子进程。这种方式比 `wait()` 更加灵活，适用于复杂的多子进程管理场景。通过设置 `pid` 参数（如 `-1` 或 `< -1` ），父进程可以控制对哪些子进程进行操作。

相比之下， `wait()` 只能阻塞式等待任意子进程退出，无法指定特定子进程，也不支持非阻塞模式。 `waitpid()` 提供了更多控制选项（ `pid` 和 `options` ），适用于更广泛的应用场景。

实验内容三：exit()

Task—练习1：比较 `exit()` 与 `_exit()` 函数的区别

运行结果

使用 `exit()`

```
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./exit2
output begin
content in buffer%
```

使用 `_exit()`

```
Documents/OS/Lab2 via C v17.0.0-clang
❯ ./_exit1
output begin
```

分析

`exit()` 刷新文件缓冲区，确保所有数据写入文件或终端后才退出。 `_exit()` 直接终止进程，不清理文件缓冲区，未写入的数据将丢失。 `printf` 使用缓冲区存储未立即输出的数据。在调用 `_exit()` 时，缓冲区未刷新，导致 "content in buffer" 未被打印。使用 `exit()` 时确保数据完整性，适合需要文件写入或终端输出的场景。使用 `_exit()` 时提升性能，适合对未完成数据无特殊要求的快速退出场景。

此外，对程序以及输出结果进行具体的分析，我们发现，使用 `exit()` 的程序最终输出的后面有一个 `%` ，这并不是程序的输出，而是 `shell` 的输出。第一个 `printf("output begin\n")` 包含换行符 `\n` ，在行缓冲模式下，遇到 `\n` 会立即刷新缓冲区，所以这个肯定会输出。但是，第二个 `printf("content in buffer")` 没有换行符，输出会留在缓冲区中，等待缓冲区刷新才会显示。

由于 `exit()` 会刷新缓冲区，因而"content in buffer"会被打印出来，但是由于没有换行符， `%` 提示符会被 `shell` 输出出来，紧跟在后面。

`_exit()` 不会刷新缓冲区，所以"content in buffer"就直接丢失了，由于"output begin\n"有换行符，所以 `%` 会在下一行被打印出来。

Task—练习2：比较 `exit()` 与 `return 0` 的区别

`exit()` ：

需要头文件 `#include <stdlib.h>` 。可以在程序的任何地方使用，在mian函数中使用时，它的作用等价于 `return 0` ，在其他函数中会终止整个程序。多进程环境下，子进程的普通函数中用 `exit()` 会立即终止该子进程。

`return 0` ：

不需要特殊的头文件。只能在函数返回时使用，在main函数中用于实现程序的正常结束，在其他函数中只返回调用者函数。多进程环境下，用 `return` 子进程还会继续执行后面的代码。

四、心得总结

实验中遇到的困难与解决方法

困难

在简单使用 `wait()` 函数处理僵尸进程时，验证 `wait()` 是否真的对子进程进行了等待无法量化

解决方法

在sleep前后添加了计时器，输出等待时间

收获与体会

通过本次实验，我对Linux进程控制机制有了更深入的理解，特别是对 `wait()` 、 `exit()` 等系统调用的使用有了更清晰的认识。在实践过程中，我学到了父进程通过 `wait()` 可以有效地同步子进程的执行，避免僵尸进程的产生，同时还能获取子进程的退出状态。 `waitpid()` 函数的非阻塞特性让我认识到进程管理的灵活性。通过分析 `exit()` 和 `_exit()` 的区别也让我明白了缓冲区的存在及其对程序输出的影响。

这次实验让我掌握了进程控制的基本操作，从简单的进程创建到复杂的进程间同步，每一步都需要仔细考虑资源的分配和状态的传递。特别是在多进程环境下， `exit()` 和 `return` 的行为差异让我意识到函数调用与进程终止的本质区别，这为今后编写更健壮、更高效的系统程序打下了坚实基础。通过此次实验，我对操作系统如何管理进程、实现进程间通信有了更直观的感受，这对于理解操作系统的核心原理有着重要意义。