

进程通信与内存管理

进程的软中断通信

1.man命令

problem:

1.直接使用man命令，出现错误 no manual entry for kill in section 2

```
[root@localhost ~]# man 2 kill
No manual entry for kill in section 2
[root@localhost ~]# man 3 kill
No manual entry for kill in section 3
[root@localhost ~]# man 2 read
No manual entry for read in section 2
[root@localhost ~]# man 3 kill
No manual entry for kill in section 3
[root@localhost ~]# man 3 fork
No manual entry for fork in section 3
```

发现是因为未下载和安装man的安装包；调用 `yum install man` 和 `yum install man-pages` 命令进行man的下载和安装。

```
[root@kp-test01 test]# yum install man
Last metadata expiration check: 0:02:56 ago on Wed 18 Oct 2023 06:17:00 PM CST.
Package man-db-2.8.7-5.oe1.aarch64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
```

```
[root@kp-test01 test]# yum install man-pages
Last metadata expiration check: 0:03:16 ago on Wed 18 Oct 2023 06:17:00 PM CST.
Dependencies resolved.
===== Package
Architecture      Version           Repository        Size
=====
man-pages          noarch           5.02-5.oe1        update 3.3 M
=====Installing:
Transaction Summary
=====Install 1 Package
Total download size: 3.3 M
Installed size: 3.0 M
Is this ok [y/N]: y
Downloading Packages:
man-pages-5.02-5.oe1.noarch.rpm                               17 MB/s | 3.3 MB  00:00
-----Total
17 MB/s | 3.3 MB  00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing                : 1/1
  Running scriptlet: man-pages-5.02-5.oe1.noarch : 1/1
  Installing           : man-pages-5.02-5.oe1.noarch : 1/1
  Running scriptlet: man-pages-5.02-5.oe1.noarch : 1/1
  Verifying            : man-pages-5.02-5.oe1.noarch : 1/1
Installed:
  man-pages-5.02-5.oe1.noarch
Complete!
```

之后显示已经complete；在调用man命令就无误了；

2.查阅man 后面的数字的含义，发现为2时是题目所需的系统调用。而sleep在section2没有说明，在section3中才有。

```
[root@kp-test01 test]# man 2 sleep
No manual entry for sleep in section 2
```

- 1 用户命令，可由任何人启动的。
 - 2 系统调用，即由内核提供的函数。
 - 3 例程，即库函数，比如标准C库libc。
 - 4 设备，即/dev目录下的特殊文件。
 - 5 文件格式描述，例如/etc/passwd。
 - 6 游戏，不用解释啦！
 - 7 杂项，例如宏命令包、惯例等。
 - 8 系统管理员工具，只能由root启动。
 - 9 其他（Linux 特定的），用来存放内核例行程序的文档。
- n 新文档，可能要移到更适合的领域。
- o 老文档，可能会在一段期限内保留。
- l 本地文档，与本特定系统有关的。

运行结果如下：

fork：

```
[root@kp-test01 test]# man 2 fork
```

```

FORK(2)                                Linux Programmer's Manual                                FORK(2)
NAME
    fork - create a child process
SYNOPSIS
    #include <sys/types.h>
    #include <unistd.h>

    pid_t fork(void);
DESCRIPTION
    fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

    The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

    The child process is an exact duplicate of the parent process except for the following points:

    * The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.
    * The child's parent process ID is the same as the parent's process ID.
    * The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).
    * Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.
    * The child's set of pending signals is initially empty (sigpending(2)).
    * The child does not inherit semaphore adjustments from its parent (semop(2)).
    * The child does not inherit process-associated record locks from its parent (fcntl(2)). (On the other hand, it does inherit fcntl(2) open file description locks and flock(2) locks from its parent.)
    * The child does not inherit timers from its parent (setitimer(2), alarm(2), timer_create(2)).
    * The child does not inherit outstanding asynchronous I/O operations from its parent (aio_read(3), aio_write(3)), nor does it inherit any asynchronous I/O contexts from its parent (see io_setup(2)).

Manual page fork(2) line 1 (press h for help or q to quit)
```

kill：

```
[root@kp-test01 test]# man 2 kill
```

```

KILL(2)                                     Linux Programmer's Manual                                     KILL(2)

NAME
    kill - send signal to a process

SYNOPSIS
    #include <sys/types.h>
    #include <signal.h>

    int kill(pid_t pid, int sig);

    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    kill(): _POSIX_C_SOURCE

DESCRIPTION
    The kill() system call can be used to send any signal to any process group or process.

    If pid is positive, then sig is sent to the process with the ID specified by pid.

    If pid equals 0, then sig is sent to every process in the process group of the calling process.

    If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.

    If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.

    If sig is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

    For a process to have permission to send a signal, it must either be privileged (under Linux: have the CAP_KILL capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

RETURN VALUE
    On success (at least one signal was sent), zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS
    EINVAL An invalid signal was specified.
Manual page kill(2) line 1 (press h for help or q to quit)

```

signal:

```
[root@kp-test01 test]# man 2 signal
```

```

SIGNAL(2)                                     Linux Programmer's Manual                                     SIGNAL(2)

NAME
    signal - ANSI C signal handling

SYNOPSIS
    #include <signal.h>

    typedef void (*sighandler_t)(int);

    sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
    The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use sigaction(2) instead. See Portability below.

    signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

    If the signal signum is delivered to the process, then one of the following happens:

    * If the disposition is set to SIG_IGN, then the signal is ignored.

    * If the disposition is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs.

    * If the disposition is set to a function, then first either the disposition is reset to SIG_DFL, or the signal is blocked (see Portability below), and then handler is called with argument signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

    The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE
    signal() returns the previous value of the signal handler, or SIG_ERR on error. In the event of an error, errno is set to indicate the cause.

ERRORS
    EINVAL signum is invalid.

CONFORMING TO
    POSIX.1-2001, POSIX.1-2008, C89, C99.
Manual page signal(2) line 1 (press h for help or q to quit)

```

sleep:

```
[root@kp-test01 test]# man 3 sleep
```

```
SLEEP(3) Linux Programmer's Manual SLEEP(3)

NAME
    sleep - sleep for a specified number of seconds

SYNOPSIS
    #include <unistd.h>

    unsigned int sleep(unsigned int seconds);

DESCRIPTION
    sleep() causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.

RETURN VALUE
    Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

ATTRIBUTES
    For an explanation of the terms used in this section, see attributes(7).



| Interface | Attribute     | Value                       |
|-----------|---------------|-----------------------------|
| sleep()   | Thread safety | MT-Unsafe sig:SIGCHLD/Linux |



CONFORMING TO
    POSIX.1-2001, POSIX.1-2008.

NOTES
    On Linux, sleep() is implemented via nanosleep(2). See the nanosleep(2) man page for a discussion of the clock used.

    Portability notes
    On some systems, sleep() may be implemented using alarm(2) and SIGALRM (POSIX.1 permits this); mixing calls to alarm(2) and sleep() is a bad idea.

    Using longjmp(3) from a signal handler or modifying the handling of SIGALRM while sleeping will cause undefined results.

SEE ALSO
    sleep(1), alarm(2), nanosleep(2), signal(2), signal(7)
    Manual page sleep(3) line 1 (press h for help or q to quit)
```

exit:

```
[root@kp-test01 test]# man 2 exit
```

```
_EXIT(2) Linux Programmer's Manual _EXIT(2)

NAME
    _exit, _Exit - terminate the calling process

SYNOPSIS
    #include <unistd.h>

    void _exit(int status);

    #include <stdlib.h>

    void _Exit(int status);

    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    _Exit():
        _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION
    The function _exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed. Any children of the process are inherited by init(1) (or by the nearest "subreaper" process as defined through the use of the prctl(2) PR_SET_CHILD_SUBREAPER operation). The process's parent is sent a SIGCHLD signal.

    The value status & 0377 is returned to the parent process as the process's exit status, and can be collected using one of the wait(2) family of calls.

    The function _Exit() is equivalent to _exit().

RETURN VALUE
    These functions do not return.

CONFORMING TO
    POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD. The function _Exit() was introduced by C99.

NOTES
    For a discussion on the effects of an exit, the transmission of exit status, zombie processes, signals sent, and so on, see exit(3).

    The function _exit() is like exit(3), but does not call any functions registered with atexit(3) or on_exit(3). Open stdio(3) streams are not flushed. On the other hand, _exit() does close open file descriptors, and this may cause an unknown delay, waiting for pending output.

    Manual page exit(2) line 1 (press h for help or q to quit)
```

2.软中断通信

problem:

1. 在父进程中调用kill后向子进程发出信号后未终止，导致父进程一直卡在wait处；

```
// Child process 2|
signal(17, SIG_IGN);
while (1) {
    pause();
}
printf("Child process2 is killed by parent!!\n");
return 0;
```

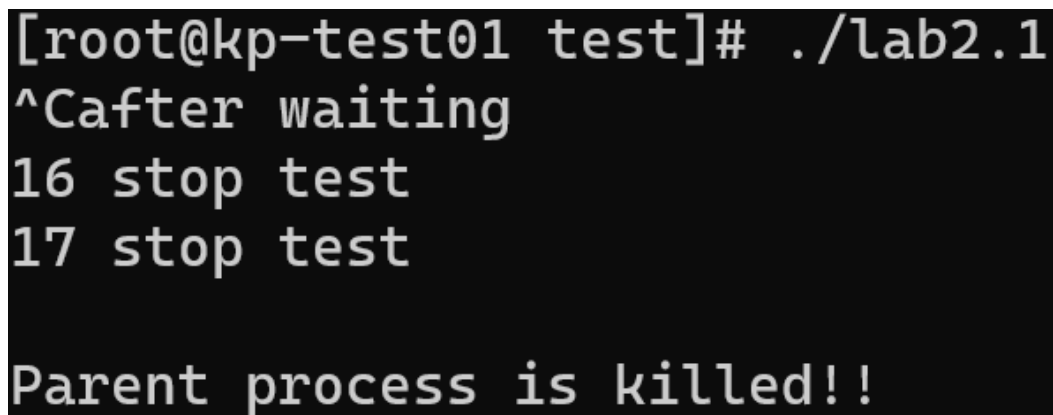
将pause处连带循环一起删去即可;

2. 没过5秒, 键盘没有产生中断, 子进程直接显示被kill;

子进程未收到信号就终止了

S:在子进程处也设置flag值即可, 收到信号才能向后运行。

3. 使用Ctrl c或者Ctrl \产生中断信号时, 父进程直接终止, 无子进程输出;



```
[root@kp-test01 test]# ./lab2.1
^Cafter waiting
16 stop test
17 stop test

Parent process is killed!!
```

将delete和quit中断信号放在main函数中, 移出父进程。

4.使用time头函数记录等待时间,start记录开始, end记录收到任一信号时的时间。

运行结果:

```
[root@kp-test01 test]# gcc -o lab2.1 lab2.1.c
[root@kp-test01 test]# ./lab2.1
5 stop test

16 stop test
17 stop test

Child process1 is killed by parent!!
Child process2 is killed by parent!!
Parent process is killed!!
[root@kp-test01 test]# ./lab2.1
^\\5 stop test

16 stop test
17 stop test

Child process1 is killed by parent!!
Child process2 is killed by parent!!
Parent process is killed!!
[root@kp-test01 test]# ./lab2.1
^\\2 stop test

16 stop test
17 stop test

Child process2 is killed by parent!!
Child process1 is killed by parent!!
Parent process is killed!!
```

理论分析:

首先输出了一个数字，表示父进程运行的时间。这个数字是由于在程序中设置了一个定时器，当定时器超时时会触发一个信号，从而让父进程退出。

接着输出了两个数字，分别为16和17。这两个数字表示父进程向子进程发送的信号，其中16表示SIGUSR1信号，17表示SIGUSR2信号。当子进程收到这些信号时，会触发相应的信号处理函数，从而让子进程退出。

然后输出了两行信息，分别表示两个子进程被父进程杀死了。这是由于子进程在收到相应的信号后，会触发相应的信号处理函数，从而让子进程退出。

最后输出了一行信息，表示父进程被杀死了。这是由于父进程在定时器超时后，会触发相应的信号处理函数，从而让父进程退出。

程序的原理是通过信号传递来控制子进程的运行和退出，其中父进程通过kill函数向子进程发送信号，子进程通过signal函数来注册信号处理函数。当子进程收到相应的信号时，会触发相应的信号处理函数，从而让子进程退出。

```

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>

int flag=0,flag1=0,flag2=0;
time_t end ;

void inter_handler() {
    flag = 1;
    end=time(NULL);
}

void c1_handler() {
    flag1 = 1;
}

void c2_handler() {
    flag2 = 1;
}

void waiting() {
    while(!flag){};
}

void waiting1() {
    while(!flag1){};
}

void waiting2() {
    while(!flag2){};
}

int main() {
    pid_t pid1=-1, pid2=-1;
    while (pid1 == -1) pid1=fork();
    signal(SIGINT,inter_handler);
    signal(SIGQUIT,inter_handler);
    if (pid1 > 0) {
        while (pid2 == -1) pid2=fork();
        if (pid2 > 0) {
            // Parent process
            signal(SIGALRM,inter_handler);

            time_t start = time(NULL);
            alarm(5);
            waiting();
            printf("%d stop test\n\n",end-start);
            kill(pid1, 16);
            printf("16 stop test\n");
            kill(pid2, 17);
            printf("17 stop test\n\n");
            wait(NULL);
            wait(NULL);
            printf("Parent process is killed!!\n");
        } else {
            // Child process 2
            signal(17, c2_handler);

```

```

        waiting2();
        printf("Child process2 is killed by parent!!\n");
        return 0;
    }
} else {
    // child process 1
    signal(16, c1_handler);
    waiting1();
    printf("Child process1 is killed by parent!!\n");
    return 0;
}
return 0;
}

```

进程的管道通信

problem:

1. 管道write函数，第二个参数不能直接使用单字符；

```

lab2.2.c: In function 'main':
lab2.2.c:17:16: warning: passing argument 2 of 'write' makes pointer from integer without a cast [-Wint-conversion]
    write(fd[1],c1,1);
                  ^~

In file included from lab2.2.c:1:0:
/usr/include/unistd.h:366:16: note: expected 'const void *' but argument is of type 'char'
extern ssize_t write (int __fd, const void *__buf, size_t __n) __wur;
                  ^~~~~~

lab2.2.c:28:17: warning: passing argument 2 of 'write' makes pointer from integer without a cast [-Wint-conversion]
    write(fd[1],c2,1);
                  ^~

In file included from lab2.2.c:1:0:
/usr/include/unistd.h:366:16: note: expected 'const void *' but argument is of type 'char'
extern ssize_t write (int __fd, const void *__buf, size_t __n) __wur;
                  ^~~~~~

```

使用字符指针即可；

运行结果：

有锁：

[illegible]

无锁：

[illegible]

理论分析：

以上程序加锁的作用是保证两个子进程写入管道时不会互相干扰，从而保证数据的完整性和正确性。如果不加锁，两个子进程同时写入管道时可能会出现数据混乱或者丢失的情况。

具体来说，加锁的方式是使用lockf函数对管道进行互斥锁操作。lockf函数的第一个参数是文件描述符，第二个参数是锁定方式，0表示解锁，1表示锁定，第三个参数是锁定的字节数。在本程序中，我们对写入管道的代码进行了加锁，即在写入之前先锁定管道，写完之后再解锁管道。

如果不加锁，程序输出的结果可能会出现以下两种情况：

1. 数据混乱：由于两个子进程同时写入管道，数据可能会交错出现，从而导致输出结果的顺序与预期不符。
2. 数据丢失：由于两个子进程同时写入管道，数据可能会被覆盖或者丢失，从而导致输出结果不完整或者错误。

因此，加锁是一种保证数据正确性的重要手段，特别是在多进程或者多线程的环境中，更需要注意加锁的问题。

源代码

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
```

```

#include <wait.h>

int main( ) {
    pid_t pid1,pid2;
    int fd[2];
    char InPipe[5000];
    char *c1="1", *c2="2";
    pipe(fd);
    while((pid1 = fork( )) == -1);
    if(pid1 == 0) {
        //lockf(fd[1],1,0);
        for(int i=0;i<2000;++i){
            write(fd[1],c1,1);
        }
        sleep(5);
        //lockf(fd[1],0,0);
        exit(0);
    }
    else {
        while((pid2 = fork()) == -1);
        if(pid2 == 0) {
            //lockf(fd[1],1,0);
            for(int i=0;i<2000;++i){
                write(fd[1],c2,1);
            }
            sleep(5);
            //lockf(fd[1],0,0);
            exit(0);
        }
        else {
            wait(0);
            wait(0);
            read(fd[0],InPipe,sizeof(InPipe));
            InPipe[4000]='\0';
            printf("%s\n",InPipe);
            exit(0);
        }
    }
    return 0;
}

```

页面置换

FIFO random:

```
[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
1
Select how to generate page references:
1. Random
2. Manual Input
1
Generated Page References: 5 7 2 5 5 2 0 2 3 2 0 5 2 5 3 2 5 3 6 6

FIFO Algorithm:
Page 5: 5 X X
Page 7: 5 7 X
Page 2: 5 7 2
Page 5: 5 7 2
Page 5: 5 7 2
Page 2: 5 7 2
Page 0: 0 7 2
Page 2: 0 7 2
Page 3: 0 3 2
Page 2: 0 3 2
Page 0: 0 3 2
Page 5: 0 3 5
Page 2: 2 3 5
Page 5: 2 3 5
Page 3: 2 3 5
Page 2: 2 3 5
Page 5: 2 3 5
Page 3: 2 3 5
Page 6: 2 6 5
Page 6: 2 6 5
FIFO Page Faults: 8
FIFO Page Fault Rate: 40.00%
```

FIFO manual input:

```
[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
1
Select how to generate page references:
1. Random
2. Manual Input
2
Enter Page References (separate with spaces): 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO Algorithm:
Page 7: 7 X X
Page 0: 7 0 X
Page 1: 7 0 1
Page 2: 2 0 1
Page 0: 2 0 1
Page 3: 2 3 1
Page 0: 2 3 0
Page 4: 4 3 0
Page 2: 4 2 0
Page 3: 4 2 3
Page 0: 0 2 3
Page 3: 0 2 3
Page 2: 0 2 3
Page 1: 0 1 3
Page 2: 0 1 2
Page 0: 0 1 2
Page 1: 0 1 2
Page 7: 7 1 2
Page 0: 7 0 2
Page 1: 7 0 1
FIFO Page Faults: 15
FIFO Page Fault Rate: 75.00%
```

LRU random:

```
[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
2
Select how to generate page references:
1. Random
2. Manual Input
1
Generated Page References: 4 5 2 6 1 6 7 5 5 3 1 2 1 2 5 2 5 0 3 7

LRU Algorithm:
Page 4: 4 X X
Page 5: 4 5 X
Page 2: 4 5 2
Page 6: 6 5 2
Page 1: 6 1 2
Page 6: 6 1 2
Page 7: 6 1 7
Page 5: 6 5 7
Page 5: 6 5 7
Page 3: 3 5 7
Page 1: 3 5 1
Page 2: 3 2 1
Page 1: 3 2 1
Page 2: 3 2 1
Page 5: 5 2 1
Page 2: 5 2 1
Page 5: 5 2 1
Page 0: 5 2 0
Page 3: 5 3 0
Page 7: 7 3 0
LRU Page Faults: 14
LRU Page Fault Rate: 70.00%
```

LRU manual input:

```

[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
2
Select how to generate page references:
1. Random
2. Manual Input
2
Enter Page References (separate with spaces): 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

LRU Algorithm:
Page 7: 7 X X
Page 0: 7 0 X
Page 1: 7 0 1
Page 2: 2 0 1
Page 0: 2 0 1
Page 3: 2 0 3
Page 0: 2 0 3
Page 4: 4 0 3
Page 2: 4 0 2
Page 3: 4 3 2
Page 0: 0 3 2
Page 3: 0 3 2
Page 2: 0 3 2
Page 1: 1 3 2
Page 2: 1 3 2
Page 0: 1 0 2
Page 1: 1 0 2
Page 7: 1 0 7
Page 0: 1 0 7
Page 1: 1 0 7
LRU Page Faults: 12
LRU Page Fault Rate: 60.00%

```

FIFO的Belady现象:

增加物理块的数量，缺页率有可能会增长；

以下是对比（页表数为6，引用序列都相同，数量为12）：

1 2 3 4 1 2 5 1 2 3 4 5

Frames=3：

```

[root@kp-test01 test]# gcc -o page page.c
[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
1
Select how to generate page references:
1. Random
2. Manual Input
2
Enter Page References (separate with spaces): 1 2 3 4 1 2 5 1 2 3 4 5

FIFO Algorithm:
Page 1: 1 X X
Page 2: 1 2 X
Page 3: 1 2 3
Page 4: 4 2 3
Page 1: 4 1 3
Page 2: 4 1 2
Page 5: 5 1 2
Page 1: 5 1 2
Page 2: 5 1 2
Page 3: 5 3 2
Page 4: 5 3 4
Page 5: 5 3 4
FIFO Page Faults: 9
FIFO Page Fault Rate: 75.00%

```

Frames=4:

```

[root@kp-test01 test]# ./page
Select the page replacement algorithm:
1. FIFO
2. LRU
1
Select how to generate page references:
1. Random
2. Manual Input
2
Enter Page References (separate with spaces): 1 2 3 4 1 2 5 1 2 3 4 5

FIFO Algorithm:
Page 1: 1 X X X
Page 2: 1 2 X X
Page 3: 1 2 3 X
Page 4: 1 2 3 4
Page 1: 1 2 3 4
Page 2: 1 2 3 4
Page 5: 5 2 3 4
Page 1: 5 1 3 4
Page 2: 5 1 2 4
Page 3: 5 1 2 3
Page 4: 4 1 2 3
Page 5: 4 5 2 3
FIFO Page Faults: 10
FIFO Page Fault Rate: 83.33%

```

在FIFO算法下，随着页面帧数的增加，缺页率却可能会增加，而不是减少。这与直觉相反，因为通常认为增加内存帧数应该降低缺页率。这一现象的原因在于FIFO算法的工作方式。

此时，Belady现象出现了。尽管页面帧数增加了，但缺页率却没有减少，而是增加了。这是因为FIFO算法替换的不是"最近使用"最少的页面，而是"最早进入"的页面，而这些可能是在更大页面帧数下被淘汰的页面。因此，增加页面帧数可能会导致更多页面被淘汰，从而增加了缺页率。

这种情况表明，FIFO算法不是一个优秀的页面置换算法，特别是当系统中页面帧数较多时，它容易导致Bleady现象。

源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define MAX_FRAMES 3      //物理块的数量
#define MAX_PAGES 8      //页表的数量
#define MAX_STRING 20    //引用序列的长度

int frames[MAX_FRAMES];
int pageReference[MAX_STRING];

//initialize
void initializeFrames() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
    }
}

void printFrames() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] == -1) {
            printf("X ");
        } else {
            printf("%d ", frames[i]);
        }
    }
    printf("\n");
}

int isPageInFrames(int page) {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] == page) {
            return i;
        }
    }
    return -1;
}

void FIFO() {
```



```

int pageFaults = 0;
int currentIndex = 0;

for (int i = 0; i < MAX_STRING; i++) {
    int page = pageReference[i];
    printf("Page %d: ", page);

    if (isPageInFrames(page)==-1) {
        frames[currentIndex] = page;
        currentIndex = (currentIndex + 1) % MAX_FRAMES;
        pageFaults++;
    }
    printFrames();
}

printf("FIFO Page Faults: %d\n", pageFaults);
printf("FIFO Page Fault Rate: %.2f%%\n", (float)pageFaults / MAX_STRING *
100);
}

void LRU() {
    int pageFaults = MAX_FRAMES;
    int Counter[MAX_FRAMES] = {0};

    for(int i = 0; i < MAX_FRAMES; i++) {
        Counter[i]=MAX_FRAMES-i;
        frames[i]=pageReference[i];
        printf("Page %d: ",frames[i]);
        printFrames();
    }
    for (int i = MAX_FRAMES; i < MAX_STRING; i++) {
        int page = pageReference[i];
        printf("Page %d: ",page);

        if (isPageInFrames(page)==-1) {
            int replaceIndex = 0;

            for (int j = 1; j < MAX_FRAMES; j++) {
                if (Counter[j] > Counter[replaceIndex]) {
                    replaceIndex = j;
                }
            }

            frames[replaceIndex] = page;
            pageFaults++;
        }

        Counter[isPageInFrames(page)] = 0;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] != -1) {
                Counter[j]++;
            }
        }

        printFrames();
    }
}

```

```

    }

    printf("LRU Page Faults: %d\n", pageFaults);
    printf("LRU Page Fault Rate: %.2f%%\n", (float)pageFaults / MAX_STRING *
100);
}

int main() {
    initializeFrames();
    int choice, i;

    //select algorithm
    printf("Select the page replacement algorithm:\n");
    printf("1. FIFO\n");
    printf("2. LRU\n");
    scanf("%d", &choice);
    //invalid input
    if (choice != 1 && choice != 2) {
        printf("Invalid choice.\n");
        return 1;
    }

    //select way
    printf("Select how to generate page references:\n");
    printf("1. Random\n");
    printf("2. Manual Input\n");
    scanf("%d", &i);

    if (i != 1 && i != 2) {
        printf("Invalid choice.\n");
        return 1;
    }

    if (i == 1) {
        srand(time(NULL));
        printf("Generated Page References: ");
        for (i = 0; i < MAX_STRING; i++) {
            pageReference[i] = rand() % MAX_PAGES;
            printf("%d ", pageReference[i]);
        }
        printf("\n");
    } else {
        printf("Enter Page References (separate with spaces): ");
        for (i = 0; i < MAX_STRING; i++) {
            scanf("%d", &pageReference[i]);
        }
    }

    if (choice == 1) {
        printf("\nFIFO Algorithm:\n");
        FIFO();
    } else {
        printf("\nLRU Algorithm:\n");
        LRU();
    }
}

```

```
}  
  
    return 0;  
}
```