八、進階I/O

Nonblocking I/O

CODE. 8-1

- 有些I/O操作(也就是之前提到的slow system call)會永久block住程式的執行，必須等到該I/O操作完成後才可以繼續往下執行，例如讀取socket。因此如果我們希望無論如何該I/O操作都必須馬上返回，此時就可以使用nonblocking I/O
- 指定file descriptor為nonblocking有下列三種方法：
  - 在呼叫open函式時指定O_NONBLOCK的flag
  - 如果檔案已經開啟，可以用fcntl函式來改變file status flag，可以使用下列指令：fcntl(fd,F_SETFL,O_NONBLOCK);
  - 如果檔案已經開啟，可以用ioctl函式將file descriptor設為nonblocking，如下：

```
int value=1;
ioctl(fd,FIONBIO,&value);
```

- 當I/O操作設定為nonblocking時，如果該I/O操作無法馬上實現，就會返回錯誤，其errno會設定為EAGAIN

鎖定檔案(1/5)

- 如果要避免檔案同時被兩個以上的行程存取，我們可以使用lock來鎖定檔案，基本上lock根據讀寫的性質可以分成兩類：
  - shared lock (也稱為read lock)
    一個檔案可以有多個shared lock，擁有該檔shared lock的行程都可以讀取其內容，但是沒有行程可以做寫入的動作(當檔案有shared lock存在時，就無法取得exclusive lock)
  - exclusive lock(也稱為write lock)
    一個檔案只能有一個exclusive lock，只有擁有該檔exclsuive lock的行程可以對其做讀寫的動作，同時間其它行程無法取得shared lock
- 如果依照lock的強制性來分，可以分成下列兩類：
  - advisory lock(也稱為cooperative lock)
    這類的lock沒有強制性，必須程式設計師配合使用才行。換言之，程式設計師可以不理會這類lock而直接對檔案作存取
  - mandatory lock
    這類的lock由kernel保證其強制性，所以使用這類lock一定可以保證其作用，其缺點是read，write等呼叫會較慢，因為每次呼叫都必須額外檢查是否有lock存在
- 如果依照lock鎖定的範圍，可以分成下列兩類
  - 整個檔案
  - 部分檔案(又稱為record lock)

鎖定檔案(2/5)：flock函式

```
#include <sys/file.h>

    int flock(int fd, int operation);
```

- flock函式只能用來設定advisory lock，而且其lock的範圍為整個檔案，不能只lock部分檔案內容(也就是不能做record lock)
- flock函式的operation參數有下列可能值：
  - LOCK_SH 取得shared lock
  - LOCK_EX 取得exclusive lock
  - LOCK_UN 釋放lock
- 如果無法馬上取得lock時，flock函式會block住，直到取得lock為止，如果不想flock函式被block住，可以將LOCK_NB與原來的operation or起來一起使用
- flock函式的鎖定狀態會記錄在file table中，因此指向同一個file table entry的file descriptor是共用同一個lock(也就是fork或dup後的file descriptor都擁有同一個lock)。因此如果要解除lock，只要向任一個指向該file table entry的file descriptor做LOCK_UN或是關閉所有指向該file table entry的file descriptor就可以

鎖定檔案(3/5)：fcntl函式

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock *lock);
```

```
struct flock {
        short l_type;       /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence;     /* SEEK_SET, SEEK_CUR, SEEK_END */
        off_t l_start;      /* Starting offset for lock */
        off_t l_len;        /* Number of bytes to lock */
        pid_t l_pid;        /* PID of process blocking our lock (F_GETLK only) */
};
```

- fcntl函式可以用來設定record lock，而這些lock可以是advisory lock或是mandatory lock，如果不特別設定，預設是advisory lock
- fcntl函式與lock相關的cmd如下:
  - F_SETLK 取得lock，如果無法取得則傳回-1，errno設為EACCES
  - F_SETLKW 取得lock，如果無法取得則一直等到能夠取得為止，如果在等待的其間擷取到訊號，則此函式會被中斷(傳回-1)，errno會設為EINTR
  - F_GETLK 測試lock參數是否會與其它lock產生衝突
- 如果要lock整個檔案，可以將flock結構中的l_len與l_start都設為0
- fcntl函式的lock不會被fork繼承(因為PID不同)，而execve會繼承此種lock(因為PID相同)
- 如果產生deadlock，其中一個fcntl函式會傳回錯誤，errno會設為EDEADLK
- 指定使用mandatory lock的方式有下列兩種
  - 在mount指令加入-o mand
  - 將檔案的set-group-id設起來，並將group的執行權限取消

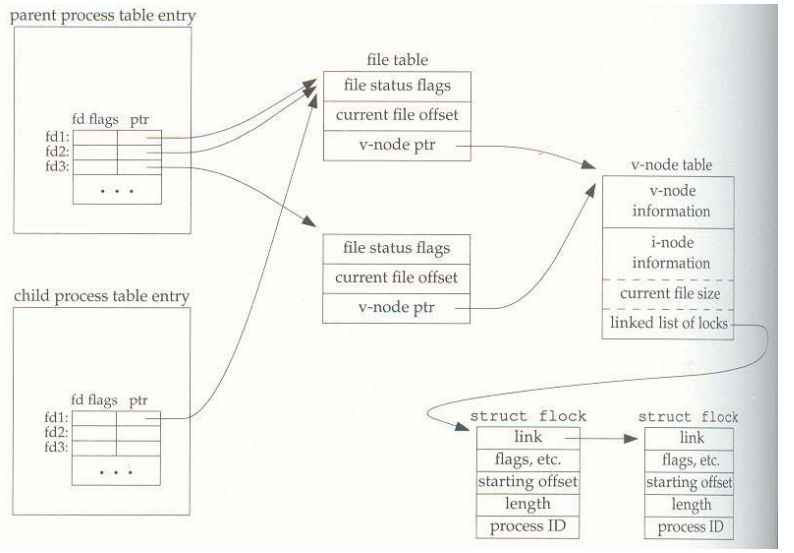
鎖定檔案(4/5)：lockf函式

```
#include <sys/file.h>

    int lockf(int fd, int cmd, off_t len);
```

- 在linux中，lockf函式只是fcntl函式的使用介面，因此fcntl所支援的lockf也都支援(像是advisory lock與mandatory lock)，不過它只能用來取得exclusive lock
- lockf函式從目前的檔案讀寫位置起算，將len長度的區域給鎖定起來，其中len可以為正數，也可以為負數
- lockf的cmd可以是下列幾種
  - F_LOCK 取得exclusive lock，直到取得才返回
  - F_TLOCK 與F_LOCK相似，但是不會被block住
  - F_ULOCK 釋放lock
  - F_TEST 測試是否已被其它行程給lock住

鎖定檔案(5/5)



I/O multiplexing(1/4)

- 當一個行程同時要存取兩個以上的file descriptor時，因為file descriptor的I/O操作預設會block，因此讀取順序將會影響程式的執行。例如要讀取a與b兩個file descriptor，假設目前b有資料讀(讀b不會被block)，但是a沒有資料讀(讀a會被block)，此時如果先讀a，程式就會被block住，而無法讀取b的資料(雖然b有資料可以讀)
- 對於上述問題有下列解決方式：
  - 分兩個執行緒或是fork出一個行程，也就是一個行程負責存取一個
  - 將所有要存取的file descriptor設為nonblocking，然後用一個迴圈來存取
  - 使用I/O multiplexing
  - 使用asynchronous I/O

I/O multiplexing(2/4)

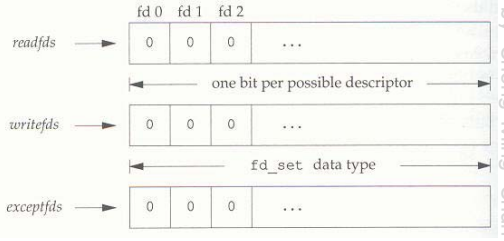
```
#include <sys/select.h>

    int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
        struct timeval *timeout);
    int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set
        *exceptfds, const struct timespec *timeout, const sigset_t *sigmask);
```

- select函式是用來做I/O multiplexing用，它的用法是將感興趣的狀態當參數，包括readfds，writefds，與exceptfds，分別代表可讀，可寫，例外狀態發生。當這些狀態沒有改變時，select函式會block住，直到timeout時間到了或是感興趣的狀態發生改變。
- timeval結構有兩個成員，分別是tv_sec(second)與tv_usec(microsecond)，總共有下列三種可能
  - 如果timouet為null pointer，select函式會等到感興趣的狀態改變才會返回
  - 如果tv_sec==0且tv_usec==0，select函式為nonblockgin模式
  - 如果tv_sec!=0或tv_usec!=0，select函式會等到感興趣的狀態改變，或是timeout
- select函式有三種回傳值
  - <0 代表有錯誤發生，例如被訊號中斷
  - ==0 代表沒有任何感興趣的狀態發生改變，是因為timeout才返回
  - >0 代表感興趣狀態發生改變的個數
- 無論監控的file descriptor是否為nonblocking都不影響select函式的運作
- 可以用getdtablesize這個函式取得該行程的最大開檔數
- pselect函式基本上與select函式功用相同，但是它多了一個參數sigmask，避免訊號在pselect函式呼叫後產生，而由signal handler處理

I/O multiplexing(3/4)

- 如何設定感興趣的狀態，總共用三類狀態 CODE. 8-3
  - 當file descriptor可讀(也就是呼叫read不會被block)，這類狀態記錄在readfds中
  - 當file descriptor可寫(也就是呼叫write不會被block)，這類狀態記錄在writefds中
  - 當file descriptor讀到Out of bound(OOB)時(因此這類狀態只能用在socket)，這類狀態記錄在exceptfds中
- readfds，writefds，exceptfds皆為一集合(以fd_set結構來實作)，可以使用下列四個巨集來設定或讀取：
  - FD_ZERO(fd_set *fdset)
    將fdset集合清空
  - FD_SET(int fd, fd_set *fdset)
    將fd加入fdset這個集合
  - FD_CLR(int fd, fd_set *fdset)
    將fd從fdset集合移除
  - FD_ISSET(int fd, fd_set *fdset)
    測試fd是否在fdset集合中



I/O multiplexing(4/4)

```
#include <sys/poll.h>

    int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

```
struct pollfd {
        int fd;         /* file descriptor */
        short events;   /* requested events */
        short revents;  /* returned events */
};
```

- poll函式的功用與select函式相同，但使用介面(參數)不一樣，它每一個要監控的file descriptor都用一個pollfd的結構來代表
- 在pollfd結構中，我們必須設定前兩個成員，分別為fd與events，其中fd代表要監控的file descriptor，而感興趣的狀態改變則設定在events中。revents為狀態改變的結果，由poll函式改變它的值
  - #define POLLIN 0x0001 /* There is data to read */
  - #define POLLPRI 0x0002 /* There is urgent data to read */
  - #define POLLOUT 0x0004 /* Writing now will not block */
  - #define POLLERR 0x0008 /* Error condition */
  - #define POLLHUP 0x0010 /* Hung up */
  - #define POLLNVAL 0x0020 /* Invalid request: fd not open */

非同步(asynchronous)I/O

- 之前的select與poll函式為同步(synchronous)的形式，也就是當我們呼叫select與poll函式時才能得知結果(哪些狀態發生改變)。如果是主動通知的方式，就為非同步(asynchronous)的形式，像訊號就為後者
- 非同步I/O的意思是當感興趣的狀態發生改變時，kernel會產生一個SIGIO或SIGURG的訊號。它基本的使用方式如下：
  - 用signal或sigaction函式註冊訊號的signal handler
  - 用fcntl(F_SETOWN)函式設定該file descriptor產生的SIGIO要送到哪一個process或是process group
  - 用fcntl(F_SETFL)函式設定該file descriptor為O_ASYNC
  - 如果只是要擷取SIGURG訊號，則第三個步驟不需要做
- 此種非同步I/O的缺點
  - 只能用在terminal與socket上
  - 每一個行程只有一個SIGIO可用，如果同時要處理多個檔案，當發生SIGIO時就不知道是哪一個檔案

使用非連續buffer(1/2)

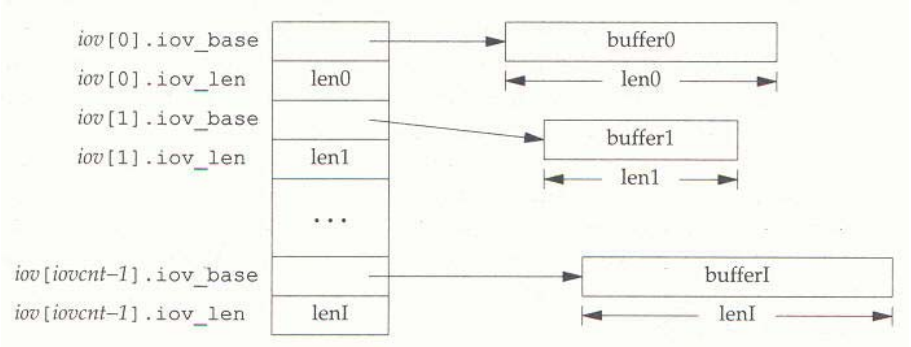
```
#include <sys/uio.h>

      ssize_t readv(int fd, const struct iovec *vector, int count);
      ssize_t writev(int fd, const struct iovec *vector, int count);
```

```
struct iovec {
        void *iov_base;    /* Starting address */
        size_t iov_len;    /* Length in bytes  */
};
```

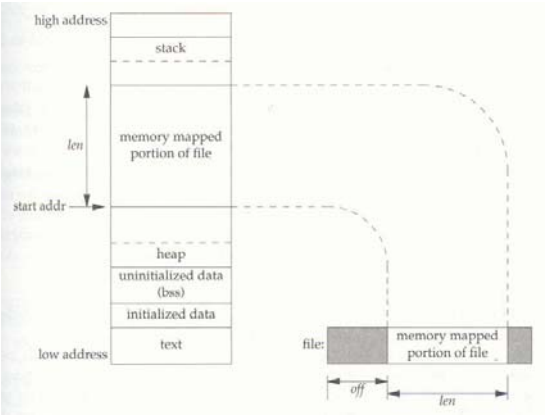
- readv可以將資料讀到一個以上的buffer中(buffer與buffer間為非連續空間)，它會先將第一個buffer填滿，然後接著填第二個buffer，依此類推
- writev可以將多個buffer的資料依序寫到檔案中
- 當處理非連續空間時，這兩個函式可以增加效率(當然也可以使用多個read或write達到相同的目的，但是比較慢)

使用非連續buffer(2/2)



記憶體映對(Memory Mapped) I/O

- 記憶體映對I/O可以將磁碟檔案的內容對應到記憶體中，如此就可以直接對記憶體來操作
- 記憶體映對I/O的優點
  - 效率較高
  - 使用起來較簡單



將磁碟檔案的內容對應到記憶體中

```
#include <sys/mman.h>

    #ifdef _POSIX_MAPPED_FILES
    void  *  mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
    int munmap(void *start, size_t length);
    #endif
```

- mmap函式將已開啟的file descriptor fd，從位置offset開始的地方，讀取length長度到start所指的記憶體位置上。其中prot參數用來指定存取的權限，flags參數用來指定運作模式
- mmap函式中的參數start只是參考值，通常是填NULL，也就是讓kernel自行決定要對應到哪裡
- prot參數(可以or起來使用)
  - PROT_NONE: 映對的記憶體空間不能存取
  - PROT_READ:映對的記憶體空間可以讀
  - PROT_WRITE：映對的記憶體空間可以寫
  - PROT_EXEC:映對的記憶體空間可以執行
- flags參數(可以or起來使用)
  - MAP_FIXED:強制要對映到參數start所指的位置
  - MAP_PRIVATE:對記憶體的修改不會反應到磁碟檔案上
  - MAP_SHARED:對記憶體的修改就有如修改磁碟檔案一般
- munmap函式用來將映對取消

CODE. 8-4

將記憶體中的映對寫回磁碟檔案

```
#include <unistd.h>
#include <sys/mman.h>

    #ifdef _POSIX_MAPPED_FILES
    #ifdef _POSIX_SYNCHRONIZED_IO
    int msync(const void *start, size_t length, int flags);
    #endif
    #endif
```

- 修改由mmap函式所映對的記憶體內容相當於改變檔案本身的內容，但是在呼叫munmap之前，對記憶體所做的修改並不能保證已經寫回檔案(效率考量)。如果要保證記憶體所做的修改已經寫回檔案，可以呼叫msync這個函式
- flags參數可以有下列值(可以or起來，但是MS_SYNC不能與MS_ASYNC同時使用)
  - MS_SYNC
    msync函式會將記憶體中的修改寫回檔案，完成後msync函式才會返回
  - MS_ASYNC
    msync函式會將寫回檔案的動作加入排程，msync在呼叫後會馬上返回，也就是不保證檔案內容已經更新
  - MS_INVALIDATE
    會將該檔案的其它記憶體映對設定成"失效"，因此這些記憶體映對會重讀該檔案內容，也就是會讀到更新的值，而不是舊的值

修改映對記憶體存取權限

```
#include <sys/mman.h>

    int mprotect(const void *addr, size_t len, int prot);
```

- mprotect函式用改變記憶體的存取權限，其中prot的值如下(可以OR起來使用)
  - PROT_NONE 不能存取
  - PROT_READ 能讀
  - PROT_WRITE 能寫(在linux下能寫就隱涵能讀)
  - PROT_EXEC 可以執行(在linux下能執行就隱涵能讀)

避免記憶體被置換(swap)出去

```
#include <sys/mman.h>

    int mlock(const void *addr, size_t len);
    int munlock(const void *addr, size_t len);
    int mlockall(int flags);
    int munlockall(void);
```

- 如果要避免page從記憶體被置換出去，可以使用mlock或mlockall函式來鎖定，被鎖定的記憶體就會一直保留在記憶體中。mlock函式用來鎖定特定區域的記憶體，而mlockall函式用來鎖定整個行程的記憶體，將page鎖定到記憶體有兩個原因
  - Real time的考量
  - Security 的考量
- munlock函式用來解除特定記憶體區域的鎖定，munlockall函式用來解除整個行程對記憶體的鎖定
- mlockall函式有一個參數flags，可以有下列兩個值
  - MCL_CURRENT 將該行程目前在記憶體的page鎖定在記憶體中
  - MCL_FUTURE 將該行程所有的記憶體鎖定在記憶體中，包含未來切換到記憶體的pages