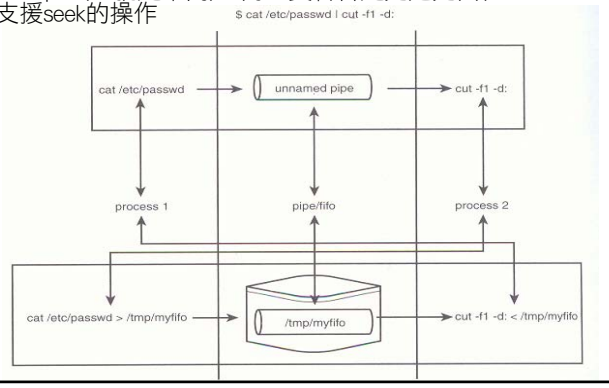
九、行程通訊

行程通訊(IPC)

- 在沒有IPC之前，兩個行程間只能透過檔案來交換資訊，這種方式不僅麻煩而且效率不好，因此本章節將介紹各種行程通訊的使用方式。在這一個章節中只介紹在同一台機器(host)上行程間的通訊，在不同機器間的行程通訊請參考下一章"Socket程式設計"。這一章將介紹下列內容
  - Pipes (unnamed pipes)
  - FIFOs (named pipes)
  - System V IPC
    - Message queue
    - Semaphore
    - Shared memory

管線(Pipes)

- Pipes可以分成兩種
  - Unnamed pipes (Pipes)
    由於這種pipes不存在檔案系統中，因此稱為unnamed pipes，這類的pipes只能用在有親屬關係的行程間。例如指令 ls | more
  - Named pipes (FIFOs)
    由於這種pipes在檔案系統中會有一代表該pipes的檔案，因此可以直接透過存取該檔案來對此類pipes做讀寫的動作，這類的pipes可以用在任一行程間的通訊。
- 這兩種pipes皆為half duplex(也就是單向)，而且資料皆是先進先出(First in and First out)，此外也都不支援seek的操作



建立pipes：unnamed pipes (1/2)

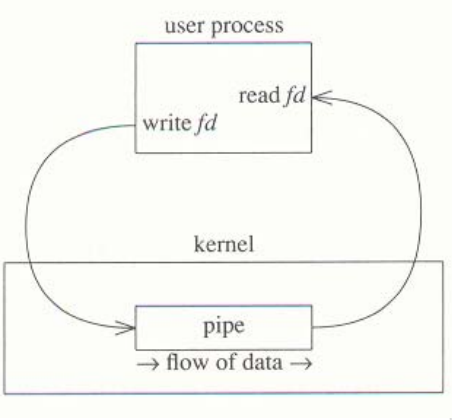
```
#include <unistd.h>

    int pipe(int fd[2]);
```

- pipe這個函式用來在核心中建立一管線(pipe)，這個管線有兩端，其中一端用來寫，另外一端用來讀，從管線寫入端寫入的資料可以從管線讀出端讀出。呼叫pipe函式之後，fd參數會被填入兩個file descriptor，其中fd[0]為讀出端(O_RDONLY)，而fd[1]為寫入端(O_WRONLY)，因此只要是寫入filedes[1]的資料都可以從fd[0]讀出
- 特別注意的是pipe為單向的(half-duplex)，也就是只能寫入fd[1]而從fd[0]讀出，不能寫入fd[0]而從fd[1]讀出
- 上述所說的兩個file descriptors可以用之前所講操作file descriptors的函式來操作，例如read, write, close, fstat等等，但不能使用lseek來移動檔案讀寫指標
- Pipes不存在檔案系統中，而是kernel中的一個inode而已，所以它不會用的disk I/O，因此效率會較好
- 當pipe用完之後，可以用close函式將兩邊的端點關閉

建立pipes：unnamed pipes (2/2)

```
int main(void){
        int fd[2];
        char buf[128];

        pipe(fd);       //建立pipe

        write(fd[1],"test\n\0",6);
        read(fd[0],buf,128);
        printf("%s",buf);
}
```

CODE. 9-1

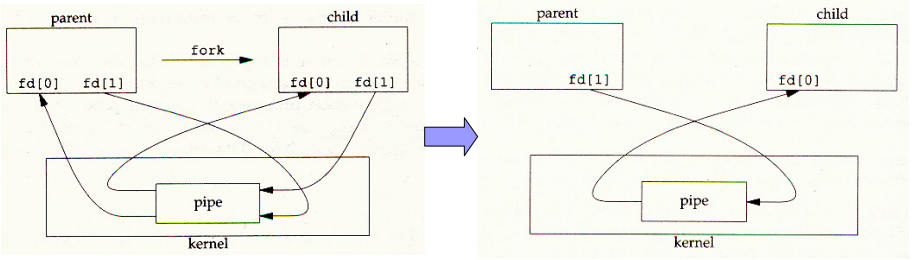


讀寫pipes

- 每一個pipes都有兩個端點，一個用來讀一個用來寫，通常都是一個writer對一個reader在使用，但是也有可能是多個writer對一個reader
- 每一個pipe的緩衝區長度定義在<limits.h>中的PIPE_BUF常數，linux中預設為4096 bytes
- 讀取pipe
  - 可以使用read函式來讀取pipe的讀取端，如果pipe中沒有資料，預設會block住，等到有資料才會返回
  - 如果另一端(也就是寫入端)已經沒有任何writer，則會傳回0，代表EOF的意思
- 寫入pipe
  - 可以使用write函式來寫入pipe的寫入端。假設要寫入n個位元組，如果pipe的緩衝區剩餘空間小於n，則write的動作會被block住，直到pipe的緩衝剩餘區空間大於等於n為止。另外不同行程間對同一個pipe的write動作不會交錯，除非單一write的大小大於PIPE_BUF
  - 如果寫入一個沒有reader的pipe，也就是讀取端已經被關閉，則會產生SIGPIPE的訊號，而write會傳回-1，errno會設為EPIPE

pipes 與 fork 的關係(1/2)

- 通常pipes不會在一個行程中單獨使用，而是作為兩個行程間溝通的橋樑。由於pipe函式呼叫後會產生兩個file descriptors，因此透過呼叫fork函式，這兩個file descriptors就會同時出現在父行程與子行程中(在父行程中已經開啟的檔案在子行程中也同樣是開啟的)
- pipe函式通常是在fork函式之前呼叫，如此才能同時在父行程與子行程中使用
- 特別注意pipe為單向的
  - 如果方向是從父行程到子行程，父行程要將fd[0]關閉，子行程要將fd[1]關閉
  - 如果方向是從子行程到父行程，父行程要將fd[1]關閉，子行程要將fd[0]關閉
  - 如果要雙向，只能建立兩個pipes
  - 不用的端點記得關閉，否則容易出問題



pipes 與 fork 的關係(2/2)

```
#include <unistd.h>

int main(void){
        int fd[2];
        int pid;

        pipe(fd);                       //create pipe

        if((pid=fork())==0){            //child
                int n;
                char buf[128];

                close(fd[1]);
                n=read(fd[0],buf,128);
                write(0,buf,n);
        }else{                          //parent
                close(fd[0]);
                write(fd[1],"hello world\n",12);
                wait(NULL);
        }
}
```

CODE. 9-2

pipes 與 dup 的關係(1/6)

- 在前面的例子中，我們都是直接對fd[0](讀)和fd[1](寫)做操作，如果有現存的兩個程式A和B，希望將A程式的結果當成B程式的輸入時該怎麼辦(也就是shell中執行 A | B)。此種情況就必須配合dup函式使用了
- 可以將fd[0]透過dup2函式複製到0，也就是標準輸入，此時讀取標準輸入就相當於讀取pipe
- 可以將fd[1]透過dup2函式複製到1，也就是標準輸出，此時寫入標準輸出就相當於寫入pipe

pipes 與 dup 的關係(2/6)

CODE. 9-3

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

int main(void){
        int fd[2],pid;
        pipe(fd);                       //create pipe
        if((pid=fork())==0){            //child
                char buf[128];
                close(fd[1]);
                dup2(fd[0],0);
                close(fd[0]);
                fgets(buf,128,stdin);
                printf("%s",buf);
        }else{                          //parent
                close(fd[0]);
                dup2(fd[1],1);
                close(fd[1]);
                printf("hello world\n");
                wait(NULL);
        }
}
```

因爲預設是fully buffered，printf不一定會真正write出去，所以這邊會出問題

pipes 與 dup 的關係(3/6)

CODE. 9-4

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

int main(void){
        int fd[2],pid;
        pipe(fd);                       //create pipe
        if((pid=fork())==0){            //child
                char buf[128];
                close(fd[1]);
                dup2(fd[0],0);
                close(fd[0]);
                fgets(buf,128,stdin);
                printf("%s",buf);
        }else{                          //parent
                close(fd[0]);
                dup2(fd[1],1);
                close(fd[1]);
                setvbuf(stdout, NULL, _IOLBF, 0);
                printf("hello world\n");
                wait(NULL);
        }
}
```

將stdout設定爲line buffered，也就是遇到換行就會自動flush

pipes 與 dup 的關係(4/6)

CODE. 9-5

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

int main(void){
        int fd[2],pid;
        pipe(fd);                       //create pipe
        if((pid=fork())==0){            //child
                char buf[128];
                close(fd[1]);
                dup2(fd[0],0);
                close(fd[0]);
                fgets(buf,128,stdin);
                printf("%s",buf);
        }else{                          //parent
                close(fd[0]);
                dup2(fd[1],1);
                close(fd[1]);
                printf("hello world\n");
                fflush(stdout);
                wait(NULL);
        }
}
```

強制stdout做flush

pipes 與 dup 的關係(5/6)

CODE. 9-6

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

int main(void){
        int fd[2],pid;
        pipe(fd);                       //create pipe
        if((pid=fork())==0){            //child
                char buf[128];
                close(fd[1]);
                dup2(fd[0],0);
                close(fd[0]);
                fgets(buf,128,stdin);
                printf("%s",buf);
        }else{                          //parent
                close(fd[0]);
                dup2(fd[1],1);
                close(fd[1]);
                printf("hello world\n");
                close(1);
                wait(NULL);
        }
}
```

直接關閉file descriptor 1

pipes 與 dup 的關係(6/6)

CODE. 9-7

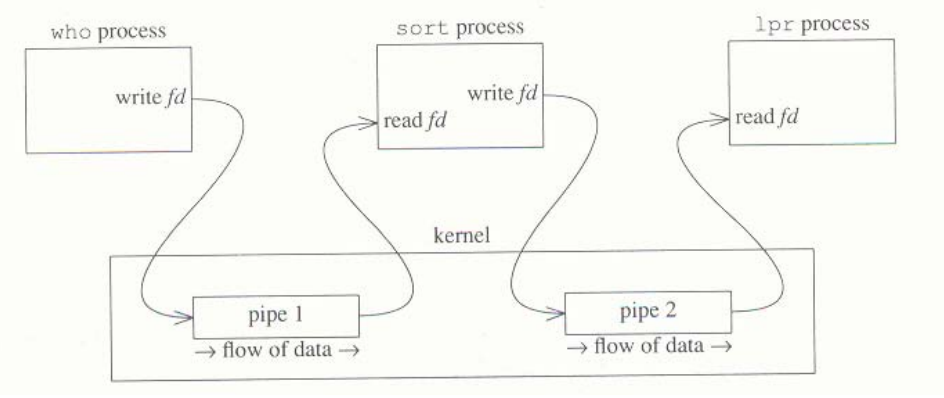
```
#include <unistd.h>
#include <stdio.h>

int main(void){
        int fd[2],pid;
        pipe(fd);                       //create pipe
        if((pid=fork())==0){            //child
                char buf[128];
                close(fd[1]);
                dup2(fd[0],0);
                close(fd[0]);
                fgets(buf,128,stdin);
                printf("%s",buf);
        }else{                          //parent
                close(fd[0]);
                dup2(fd[1],1);
                close(fd[1]);
                printf("hello world\n");
                fclose(stdout);
                wait(NULL);
        }
}
```

fclose也會自動作flush的動作

pipes 與 shell

■在shell中執行下列指令，其process與pipes的關係如下圖

$ who | sort | lpr



popen function(1/2)

```
#include <stdio.h>

    FILE *popen(const char *command, const char *type);
    int pclose(FILE *stream);
```

- 如果只是要讀取某個行程的結果，或是寫入某個行程的輸入，就可以使用popen與pclose函式
- popen函式相當於建立pipe，fork出子行程，關閉pipe中不用的端點，最後呼叫execve函式來執行command。popen函式還有一個參數type，有兩種值:
  - "r":pipe方向為讀取子行程的結果
  - "w":pipe方向為寫入子行程的標準輸入
- pclose函式相當於關閉pipe中所有端點，然後wait子行程的結束，其傳回值即為子行程的結束狀態
- popen與system函式相同，都是呼叫/bin/sh –c 來執行command

popen function(2/2)

CODE. 9-8

```
#include <stdio.h>

int main(void){
        FILE *fp;
        char buf[256];
        int line=1;

        fp=popen("/bin/ls -a" ,"r");                    //read result from command “ls”

        while(fgets(buf,256,fp)!=NULL){
                printf("%d: %s",line++,buf);
        }

        pclose(fp);
}
```


建立FIFOs：named pipes

```
#include <sys/types.h>
#include <sys/stat.h>

      int mkfifo(const char *pathname, mode_t mode);
```

- 與pipe不同的是，FIFOs會存在檔案系統中，因此它可以用在任一行程間的通訊(必須擁有該FIFO的讀或寫權限)，不像pipe只能用在父子行程間。另一方面，建立出來的FIFOs會一直存在於檔案系統中，除非刪除(unlink)它，不像pipe在行程結束後就會自動消滅
- mkfifo函式用來建立FIFO，該函式會在檔案系統中建立pathname的檔案，其存取權限為mode(當然真正的建立權限為mode & ~umask)。注意mkfifo函式只是用來建立FIFO，因此其傳回值代表的是是否建立成功，而不是該FIFO的file descriptor
- 也可以使用mkfifo(1)指令來建立FIFO
- 通常FIFO是事先建立好的，行程間就可以透過這些FIFO來通訊

操作FIFO(1/2)

- 操作FIFO大致與操作檔案相同，首先要用open函式開啟它，開啟之後才能對它做read與write的動作，不過FIFO與普通檔案不同，必須等到兩端都開啟後才能做read與write的動作
  - 如果open時的flags為O_RDONLY，該open呼叫會等到有任一個行程開啟該FIFO為O_WRONLY時才返回
  - 如果open時的flags為O_WRONLY，該open呼叫回等到有任一個行程開啟該FIFO為O_RDONLY時才返回
  - 如果open時的flags為O_RDONLY |O_NONBLOCK，該open呼叫會馬上返回
  - 如果open時的flags為O_WRONLY |O_NONBLOCK，該open呼叫會馬上返回，但是如果該FIFO沒有任何一個行程以O_RDONLY開啟則會發生錯誤，其errno會設為ENXIO
- 與pipe相同，如果寫入一個沒有reader的FIFO會產生SIGPIPE的訊號，另外如果讀取一個沒有writer的FIFO則會讀到EOF
- 通常一個FIFO可能有許多的writers，與pipe相同的是，如果這些單一write的大小不超過PIPE_BUF時，這些write就為atomic，也就不會有交錯的情形出現
- 當行程不再需要該FIFO時就可以用close函式將之關閉，特別注意，行程結束時並不會刪除FIFO，但是FIFO中剩餘的資料將會被清除
- 如果某個FIFO已經沒有任何用途，可以用unlink函式將之刪除

操作FIFO(2/2)

CODE. 9-9

建立FIFO

```
int main(void){
        mkfifo("fifo",0660);                    //create fifo
}
```

writer

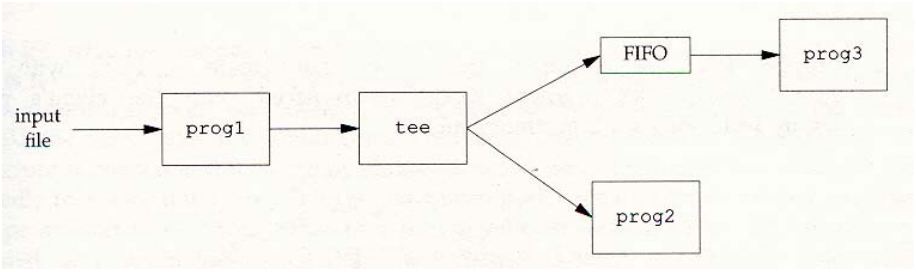
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void){
        int fd=open("fifo",O_WRONLY);           //write to fifo
        write(fd,"fifo test\n",10);
}
```

reader

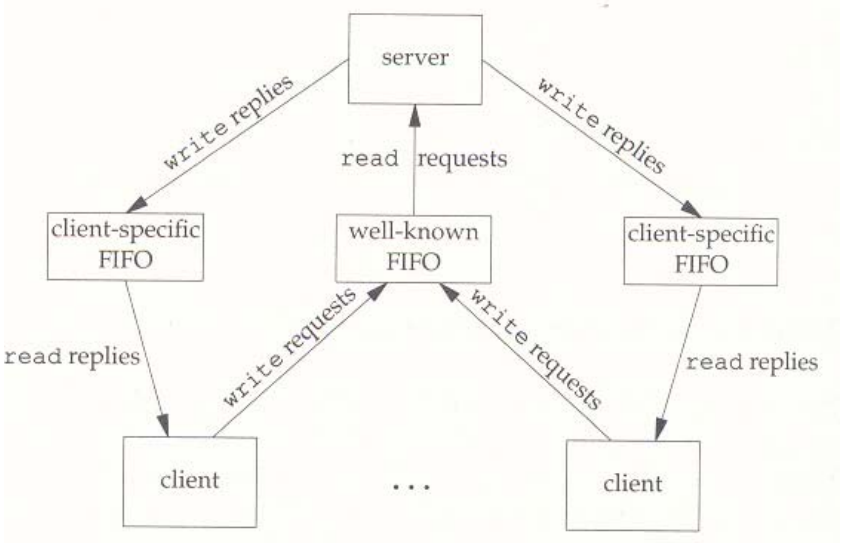
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void){
        char buf[256];
        int fd=open("fifo",O_RDONLY);           //read from fifo
        read(fd,buf,256);
        printf("%s",buf);
}
```

FIFO範例：複製輸出串流

- 下列指令可以將prog1的輸出串列同時丟到prog2與prog3，傳統的方法可能要先將prog1的結果存成暫存檔案，再分別丟給prog2與prog3，使用FIFO可以避免產生出暫存檔案
  - mkfifo fifo1
  - prog3 <fifo1 &
  - prog1 < infile | tee fifo1 | prog2



FIFO範例：clients與server溝通



System V IPC

- System V IPC總共包含下列三種IPC，它們都是由核心來管理，此外這三種IPC都有類似的使用介面與設計方式
  - Message queue
  - Semaphore
  - Shared memory
- 在linux下可以用下列指令來查詢與操作System V IPC
  - ipcs(8) 查看IPC目前的狀態，會列出目前系統中存在的各種System V IPC狀態
  - ipcrm(8) 用來刪除系統中存在的System V IPC
- System V IPC的優點與缺點
  - 優點
    - 許多作業系統都有支援，有較高的portability
  - 缺點
    - 使用介面複雜
    - IPC object的資源有限 (可以用 ipcs -l 來看)
    - 當行程結束時不會自動釋放
    - 不能使用multiplexing I/O
- 將會被POSIX IPC取代

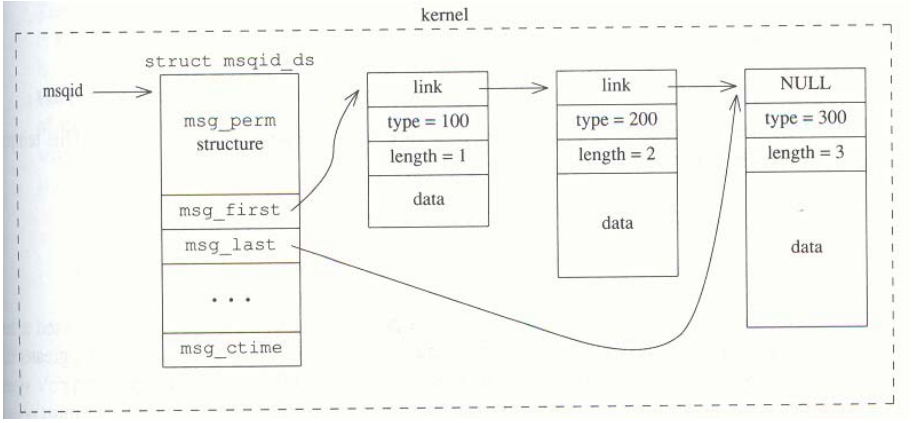
IPC object的識別(identifier)

- 每一個IPC object都有一個用來識別的id，同一類型的IPC object不會有重複的id。與file descriptor不同的是它不是每次都從零開始計算，而是一直累計上去
- IPC object ID是執行時期(runtime)才決定的，換句話說不是每次都相同。由於IPC是用來做行程通訊，因此行程間必須事先知道要使用哪一個IPC object，但是執行前又不知道該IPC object id為何，所以設計出一套從key轉成id的方式，所有要溝通的行程事先都知道同一個key，如此執行時期就可以透過該固定的key來查詢到不固定的id
- 如何決定key是一個問題，基本上有下列幾種方法:
  - key使用IPC_PRIVATE，如此一來保證一定有一個新的IPC object被建立出來，然後將id寫到磁碟檔案中，其它行程再讀該檔案取得id，或是子行程可以直接取得id
  - 將key寫在某個header檔中，要做行程通訊的行程使用這個header檔，雖然這樣可以讓這些行程取得相同的id，但是不能保證該key沒有其它的行程使用
  - 用ftok函式取得key，ftok函式有兩個參數，一個是路徑名，一個是project id，它可以將這兩個轉換成key，因此我們可以將這兩個資訊記錄到header中，然後每個行程在用ftok來取得key。這方法的問題是ftok函式可能還是會產生重複的key

Message Queue(1/2)

- Message queue是一個儲存在kernel中的linked list，行程可以將訊息丟到其中，而由另一個行程從該message queue讀出訊息，透過這種方式得到行程間通訊的目的
- 核心中可以有許多message queue，而每一個message queue皆有一個queue id，因此無論是寫入訊息或讀出訊息都必須透過queue id來指定操作的是哪一個message queue
- Message queue中的訊息基本上是先進先出的(FIFO)，但是透過指定訊息類別(message type)可以不照順序直接存取某個類別的訊息
- Message queue的相關函式如下:
  - msgget :用來取得message queue id或是建立新的message queue
  - msgsnd: 用來送訊息到message queue中
  - msgrcv: 用來從message queue中讀出訊息
  - msgctl: 用來刪除與控制message queue，例如刪除

Message Queue(2/2)



取得或建立message queue(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

      int msgget(key_t key, int msgflg);
```

- msgget函式用來取得或建立某個message queue
  - 如果key為IPC_PRIVATE
    - 一定會建立出一個新的message queue
  - 如果key不為IPC_PRIVATE
    - 如果key所對應的message queue已經存在
      - 此函式會傳回該message queue的id。
      - 如果msgflg包含IPC_CREATE與IPC_EXCL就會產生錯誤
    - 如果key所對應的message queue不存在
      - 如果msgflg不包含IPC_CREATE，會產生錯誤
      - 如果msgflg包含IPC_CREAT，該message queue會被建立
- msgflg的最小9個bits為該message queue的存取權限(讀與寫)，只有在建立message queue時用的到(由於是System V所以權限不會被umask影響)
  - 例如 msgget(123456, IPC_CREAT | 0666);

取得或建立message queue(2/2)

CODE. 9-10

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(void){
      int qid;
      key_t key=0x12345;

      qid=msgget(key,IPC_CREAT|0666);
      printf("queue id:%d\n",qid);
}
```

結果

```
[root@dcsvpn9 279]# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00012345 163841     root       666        0            0
```

將訊息寫入message queue(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

    int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

```
struct msgbuf {
        long mtype;    /* message type, must be > 0 */
        char mtext[512];  /* message data */
};
```

- msgsnd函式用來將訊息丟入message queue中，它會將訊息接在message queue中的最後面。其中訊息的格式像msgbuf所定義，不過它可以根據需求來修改它，唯一的要求是第一個member一定要是一個long型別，代表該訊息的類別，之後接的就為該訊息的資料，例如最下面的例子
- msgsz參數為傳送訊息的大小，不包含mtype，所以其大小為sizeof(msgbuf) – sizeof(long)
- msgflag參數可以是0或是IPC_NOWAIT，前者當message queue滿時會block住，後者不會

```
struct msgbuf {
        long mtype;    /* message type, must be > 0 */
        int i;
        char data[10];
};
```


將訊息寫入message queue(2/2)

CODE. 9-11

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf {
    long mtype;    /* message type, must be > 0 */
    int a,b;       /* data */
};

int main(void){
    int qid;
    key_t key=0x12345;
    struct msgbuf buf;

    qid=msgget(key,0);

    buf.mtype=1; buf.a=10; buf.b=20;                    //prepare msg1
    msgsnd(qid,&buf,sizeof(buf)-sizeof(long),0);        //send msg1

    buf.mtype=9; buf.a=-10; buf.b=-20;                  //prepare msg2
    msgsnd(qid,&buf,sizeof(buf)-sizeof(long),0);        //send msg2
}
```

結果

```
[root@dcsvpn9 281]# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00012345 163841     root       666        16           2
```

從message queue中讀出訊息(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

    ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
```

- msgrcv函式用來讀取擺放在id為msgid的 message queue中的訊息
- 讀到的訊息會擺放到msgp所指向的空間
- msgsz參數代表讀取訊息的資料大小，如果訊息資料大小大於msgsz，會有下列兩種可能
  - 如果msgflg有設定MSG_NOERROR，超過的部分會被截掉
  - 如果msgflg沒有設定MSG_NOERROR，msgrcv會傳回錯誤，而errno會被設為E2BIG
- 之前提到過message queue中的訊息不一定是FIFO，這個特點是透過指定msgtyp參數來達成
  - msgtyp=0時，拿取第一個訊息，也就是FIFO的形式
  - msgtyp>0時，拿取第一個mtype=msgtyp的訊息
  - msgtyp<0時，拿取mtype<=|msgtyp|中最小\mtype的第一個訊息
- msgflg參數可以包含IPC_NOWAIT，如果沒有設定IPC_NOWAIT，且message queue是空的，msgrcv函式就會被block住，如果有設定IPC_NOWAIT，msgrcv函式會馬上返回，並將errno設定為ENOMSG

從message queue中讀出訊息(2/2)

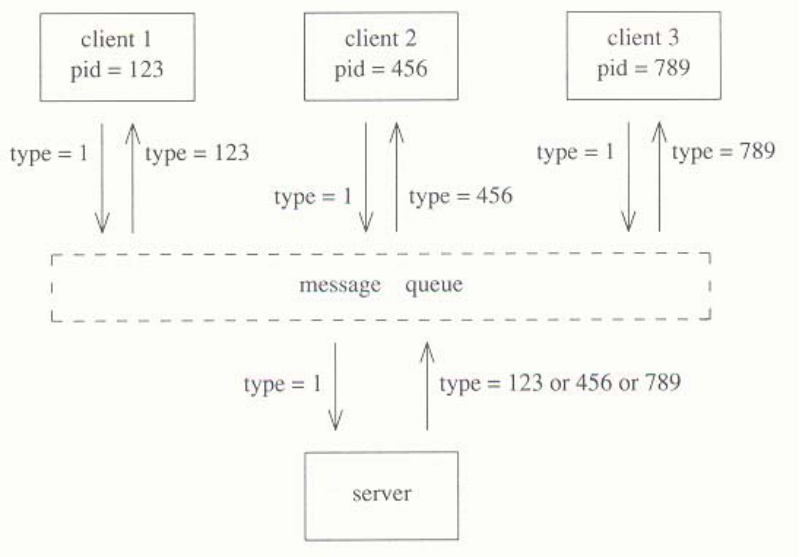
CODE. 9-12

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf {
        long mtype;     /* message type, must be > 0 */
        int a,b;        /* data */
};

int main(void){
        int qid;
        key_t key=0x12345;
        struct msgbuf buf;

        qid=msgget(key,0);

        msgrcv(qid,&buf,sizeof(buf)-sizeof(long),9,0);                  //read first msg in the queue
        printf("msg type:%d a=%d b=%d\n",buf.mtype,buf.a,buf.b);

        msgrcv(qid,&buf,sizeof(buf)-sizeof(long),0,0);                  //read second msg in the queue
        printf("msg type:%d a=%d b=%d\n",buf.mtype,buf.a,buf.b);
}
```

```
[root@dcsvpn9 283]# ./recvmsg
msg type:9 a=-10 b=-20
msg type:1 a=10 b=20
```

結果:注意讀出的順序

message multiplexing



控制與刪除message queue(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

    int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- msgctl函式可以做三件事，由cmd參數來指定：
  - IPC_RMID，用來刪除message queue
    msgctl(msgid, IPC_RMID, NULL);
  - IPC_STAT，用來取得message queue的狀態
    struct msqid_ds buf;
    msgctl(msgid, IPC_STAT, &buf);
  - IPC_SET，用來改變message queue的UID(msg_perm.uid)，GID(msg_perm.gid)，存取權限(msg_perm.mode)與容量(msg_qbytes)
    struct msqid_ds buf;
    buf.msg_perm.mode=0444;
    msgctl(msgid, IPC_SET, &buf);

控制與刪除message queue(2/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(void){
        int qid;
        key_t key=0x12345;
        struct msqid_ds buf;

        qid=msgget(key,0);

        msgctl(qid,IPC_STAT,&buf);              //讀出message queue的相關資訊

        printf("uid:%d\n",buf.msg_perm.uid);    //印出effective uid
        printf("gid:%d\n",buf.msg_perm.gid);    //印出effective gid
        printf("cuid:%d\n",buf.msg_perm.cuid);  //印出creator's uid
        printf("cgid:%d\n",buf.msg_perm.cgid);  //印出creator's gid
        printf("mode:%o\n",buf.msg_perm.mode);//印出權限
        printf("size:%d\n",buf.msg_qbytes);     //印出queue的大小
}
```

結果

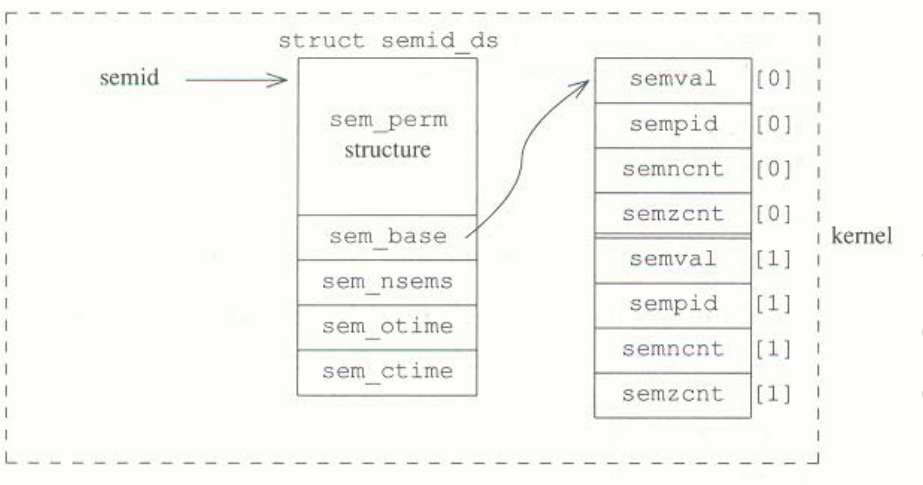
```
[root@dcsvpn9 285]# ./ctlmsg
uid:0
gid:0
cuid:0
cgid:0
mode:666
size:16384
```

CODE. 9-13

Semaphore(1/2)

- semaphore與其它的IPC不同，它並沒有在行程間交換資訊。semaphore是用來控制存取共用的資源，避免race condition的情況出現。
- semaphore可以稱為號誌，就像是紅綠燈一樣，當semaphore的值大於0時，代表綠燈，也就是可以通過，當semaphore的值等於0時代表紅燈 ，也就是必須停下來等待變成綠燈才能通過
- semaphore的相關函式
  - semget:用來取得semaphore id或是建立新的semaphore
  - semop:用來操作semaphore
  - semctl:用來刪除與控制semaphore

Semaphore(2/2)



建立semaphore(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

    int semget(key_t key, int nsems, int semflg);
```

- semget函式用來建立或取得semaphore
  - 如果key為IPC_PRIVATE
    - 一定會建立出一個新的semaphore
  - 如果key不為IPC_PRIVATE
    - 如果key所對應的semaphore已經存在
      - 此函式會傳回該semaphore的id。
      - 如果msgflg包含IPC_CREATE與IPC_EXCL就會產生錯誤
    - 如果key所對應的semaphore不存在
      - 如果msgflg不包含IPC_CREATE，會產生錯誤
      - 如果msgflg包含IPC_CREAT，該semaphore會被建立
- msgflg的最小9個bits為該semaphore的存取權限(讀與修改)，只有在建立semaphore時用的到，跟message queue不同的是，它沒有寫的權限，取而代之的是修改的權限(因為不能直接寫入semaphore)
  - 例如 semget(123456, IPC_CREAT | 0666);
- 特別注意，每一個semaphore object裡包含一群semaphores，因此semget函式每次是建立出nsems個semaphores，而不是只有一個。在semaphore object中的semaphore從0開始編號
- semget函式無法設定semaphores的初始值，必須使用semctl函式來設定

建立semaphore(2/2)

CODE. 9-14

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

int main(void){
        int sid;
        FILE *fp;
        fp=fopen("/tmp/sid.tmp","w");

        sid=semget(IPC_PRIVATE,1,0660);       //建立新的semaphore object
        printf("semaphore id:%d\n",sid);

        fprintf(fp,"%d",sid);                 //將semaphore id寫到檔案裡供之後使用
}
```

結果

```
[root@dcsvpn9 287]# ipcs -s

------ Semaphore Arrays --------
key        semid     owner      perms      nsems     status
0x00000000 163841    root       660        1
```


操作semaphore(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

    int semop(int semid, struct sembuf *sops, unsigned nsops);
```

```
struct sembuf{
        unsigned short sem_num;  /* semaphore number */
        short sem_op;            /* semaphore operation */
        short sem_flg;           /* operation flags */
};
```

- semop函式用來操作semaphore，其操作的semaphore object指定在semid，而所操作的指令寫在sops所指向的空間，參數nsops代表有多少個指令在sops中
- 每一個指令的格式定義在sembuf中，其中sem_num為semaphore在semaphore object中的編號，sem_flg可以為SEM_UNDO(當行程結束自動undo，也就是釋放)與IPC_NOWAIT(不block)。sem_op可以有下列三類:
  - sem_op>0: 代表該資源已經被釋放，semaphore的值會增加
  - sem_op<0: 代表行程需要該資源，如果無法馬上取得資源，會block住等待(除非sem_flg設定為IPC_NOWAIT)，semaphore的值在取得資源後會減少
  - sem_op=0: 會block住直到semaphore的值變為0
- 同一個semop呼叫中的sops指令為atomic

操作semaphore(2/2)

CODE. 9-15

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <signal.h>

int sid;

struct sembuf lockop[2]={
        0, 0, 0,
        0, 1, SEM_UNDO
};

struct sembuf unlockop[1]={
        0, -1, IPC_NOWAIT
};

void unlock(int arg){
        printf("unlock\n");
        semop(sid,unlockop,1);
}
```

```
int main(void){
        FILE *fp;
        signal(SIGUSR1,unlock);        //註冊signal

        fp=fopen("/tmp/sid.tmp","r");
        fscanf(fp,"%d",&sid);
        printf("semaphore id:%d\n",sid);

        semop(sid,lockop,2);           //取得lock
        printf("get lock\n");
        for(;;) sleep(60);
}
```


控制與刪除semaphore(1/2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

    int semctl(int semid, int semnum, int cmd, ...);
```

- semctl函式用來控制與刪除semaphore，它必須指定是哪一個semaphore object(semid)中的哪一個semaphore(semnum)，它可以做下列事(cmd)：
  - GETVAL: 取得單一semaphore的值
  - SETVAL: 設定單一semaphore的值(用到第四個參數)
  - GETPID: 取得最後一個操作該semaphore(呼叫semop函式)的行程id
  - GETNCNT: 取得有多少行程在等待semaphore增加
  - GETZCNT:取得有多少行程在等待semaphore變0
  - GETALL:取得semaphore object中所有semaphore的值(用到第四個參數)
  - SETALL: 設定semaphore object中所有semaphore的值(用到第四個參數)
  - IPC_RMID: 刪除semaphore object
  - IPC_SET: 設定semaphore object的uid，gid與存取權限
  - IPC_STAT:取得semaphore object的狀態
- 第四個參數為一union，根據不同的cmd有不同的參數，必須在自己的程式碼中宣告

```
union semun {
        int val;                /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT, IPC_SET */
        unsigned short *array;  /* array for GETALL, SETALL */
};
```

控制與刪除semaphore(2/2)

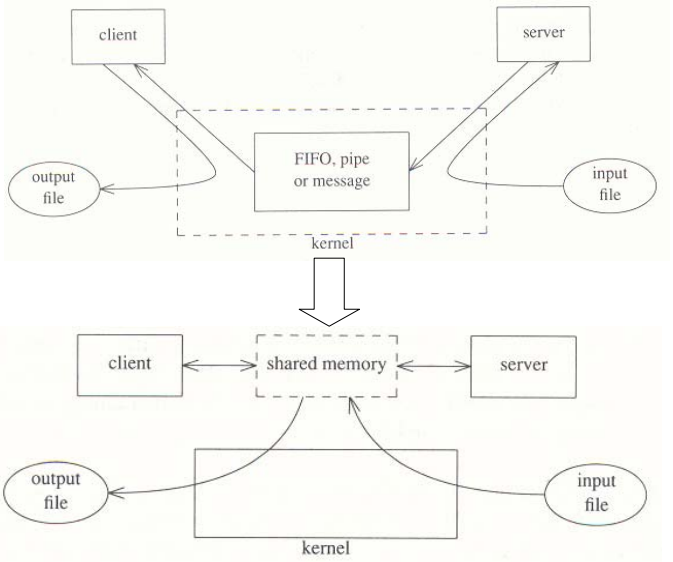
CODE. 9-16

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

int main(void){
        int sid;
        FILE *fp;
        fp=fopen("/tmp/sid.tmp","r");
        fscanf(fp,"%d",&sid);

        printf("semaphore id:%d\n",sid);

        semctl(sid,0,IPC_RMID);          //刪除semaphore object
}
```


共享記憶體(shared memory)(1/2)

- 共享記憶體是一塊存在kernel中的記憶體，而這塊記憶體可以同時連結到多個行程中，透過這一共用的空間，行程可以彼此交換資料
- 對於共享記憶體所做的任何改變都是即時的(連結到同一塊共享記憶體的行程都可以馬上看到改變)
- 共享記憶體的對應是以頁(PAGE_SIZE)為單為
- 相關函式
  - shmget:用來取得已存的shmid或是建立新的共享記憶體區段
  - shmat: 用來將共享記憶體區段對應到行程位址
  - shmdt: 取消共享記憶體的對應
  - shmctl: 用來刪除與控制共享記憶體

共享記憶體(shared memory)(2/2)



建立共享記憶體

```
#include <sys/ipc.h>
#include <sys/shm.h>

    int shmget(key_t key, int size, int shmflg);
```

- shmget函式用來建立或取得共享記憶體
  - 如果key為IPC_PRIVATE
    - 一定會建立出一個新的共享記憶體
  - 如果key不為IPC_PRIVATE
    - 如果key所對應的共享記憶體已經存在
      - 此函式會傳回該共享記憶體的id。
      - 如果shmflg包含IPC_CREATE與IPC_EXCL就會產生錯誤
    - 如果key所對應的共享記憶體不存在
      - 如果shmflg不包含IPC_CREATE，會產生錯誤
      - 如果shmflg包含IPC_CREAT，該共享記憶體會被建立
- shmflg的最小9個bits為該共享記憶體的存取權限(讀與寫)，只有在建立共享記憶體時用的到
  - 例如 shmget(123456, IPC_CREAT | 0666);
- size參數用來指定共享記憶體的大小，如果不是PAGE_SIZE的倍數，shmget會自動調整成PAGE_SIZE的倍數

連結到共享記憶體(1/2)

```
#include <sys/types.h>
#include <sys/shm.h>

      void *shmat(int shmid, const void *shmaddr, int shmflg);
      int shmdt(const void *shmaddr);
```

- 在使用共享記憶體前，我們必須將之對應到行程中的某段記憶體，shmat函式就是用來做這件事
- shmat函式指定將id為shmid的共享記憶體對應到行程指標shmaddr所指的位置上， shmat返回的位址即為行程所對應的位址，其中shmaddr參數有兩種可能:
  - shmaddr==NULL，由系統決定要對應到哪裡(通常用這種)
  - shmaddr!=NULL，由程式設計師指定要對應到哪裡，如果shmflg不包含SHM_RND，shmaddr必須要是SHMLBA的倍數，否則會發生錯誤。如果shmflg包含SHM_RND，則會自動對應到最相近shmaddr且為SHMLBA倍數的位置
- shmflg參數用來指定一些對應的方式，除了SHM_RND外還有下列可能:
  - 如果包含SHM_RDONLY，代表共享記憶體是唯讀的
  - 如果不包含SHM_RDONLY，代表共享記憶體是可讀可寫的
- shmdt函式用來取消shmaddr的共享記憶體對應

連結到共享記憶體(2/2)

CODE. 9-17

建立shared memory

```
#include <sys/ipc.h>
#include <sys/shm.h>

int main(void){
      key_t key=0x12345;
      int mid;
      mid=shmget(key,1024,IPC_CREAT|0660);
}
```

writer

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main(int argc,char *argv[]){
      char *buf;
      key_t key=0x12345;
      int mid,len;

      mid=shmget(key,0,0);
      printf("shared memory id:%d\n",mid);

      buf=shmat(mid,0,0);
      len=strlen(argv[1]);
      strncpy(buf,argv[1],len);
      buf[len]=0;
      shmdt(buf);
}
```

reader

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main(int argc,char *argv[]){
      char *buf;
      key_t key=0x12345;
      int mid;

      mid=shmget(key,0,0);
      printf("shared memory id:%d\n",mid);

      buf=shmat(mid,0,0);
      printf("%s\n",buf);
      shmdt(buf);
}
```

控制與刪除共享記憶體

```
#include <sys/ipc.h>
#include <sys/shm.h>
      int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

```
struct shmid_ds {
            struct ipc_perm shm_perm;       /* operation perms */
            int shm_segsz;                  /* size of segment (bytes) */
            time_t shm_atime;               /* last attach time */
            time_t shm_dtime;               /* last detach time */
            time_t shm_ctime;               /* last change time */
            unsigned short shm_cpid;        /* pid of creator */
            unsigned short shm_lpid;        /* pid of last operator */
            short shm_nattch;               /* no. of current attaches */
            ...
};
```

- shmctl函式用來對id為shmid的共享記憶體做控制，cmd可以有下列選擇
  - IPC_STAT: 用來取得共享記憶體的狀態
  - IPC_SET: 用來設定共享記憶體的uid,gid,存取權限
  - IPC_RMID: 用來刪除共享記憶體
  - IPC_LOCK: 將share memory鎖定在記憶體中，也就是不會被置換出去(只有root能做)
  - IPC_UNLOCK: 將share memroy解除鎖定在記憶體中