POSIX MESSAGE QUEUES

范真瑋

Overview

mq_open() — 建立一個新的message queue或打開一個現有的queue

mq_send() — 把message寫入queue

mq_receive() — 從queue中讀取message

mq_close() — 關閉之前打開的message queue

mq_unlink() — 刪除message queue

另外還有一些function是POSIX message queue特有的：

- 每個message queue都有一組屬性。使用mq_open()建立或打開queue時，可以設定其中的一些屬性。還提供兩個function取得和設定queue的屬性：mq_getattr()和mq_setattr()

- mq_notify()讓process註冊queue的message通知。在註冊之後，通過傳遞信號(signal)或通過在thread中調用function來通知process message的可用性

Opening a message queue

mq_open() — 建立一個新的message queue或打開一個現有的queue

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);

        Returns a message queue descriptor on success, or (mqd_t) -1 on error
```

參數

name — 用來識別message queue

oflag — 是一個bit mask，用來控制mq_open()的各種操作

Bit values for the mq_open() oflag argument

Flag	Description
O_CREAT	Create queue if it doesn't already exist
O_EXCL	With O_CREAT, create queue exclusively
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_NONBLOCK	Open in nonblocking mode

O_NONBLOCK會使queue以nonblocking模式打開。
後續對mq_receive()或mq_send()的調用不會進行blocking，
會立即失敗，並顯示EAGAIN錯誤。

- 參數mode是一個bit mask，用來指定新的message queue的權限。
  可以指定的值和開檔open()一樣，設定讀取權限給對應的使用者。

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

- 參數attr是指定新的message queue的mq_attr屬性結構。
  如果attr為NULL，則使用預設屬性建立queue。

fork(), exec(), and process termination on message queue descriptors

在fork()期間，child接收parent的message queue descriptors的複製，這些descriptors對應到相同已開啟的message queue descriptors。但child不會繼承parent的任何message通知註冊(notification registrations)。

當process執行exec()或終止時，所有它打開的message queue descriptors都將關閉。關閉後，所有processes在對應queue上的message通知註冊(notification registrations)都被註銷(deregistered)。

Closing a message queue

mq_close() — 關閉message queue descriptor mqdes

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Returns 0 on success, or -1 on error

當porcess終止或調用exec()時，message queue descriptor會自動關閉。和file descriptors一樣，我們應該明確地關閉message queue不再需要的descriptors，以防止耗盡message queue descriptor。

Removing a message queue

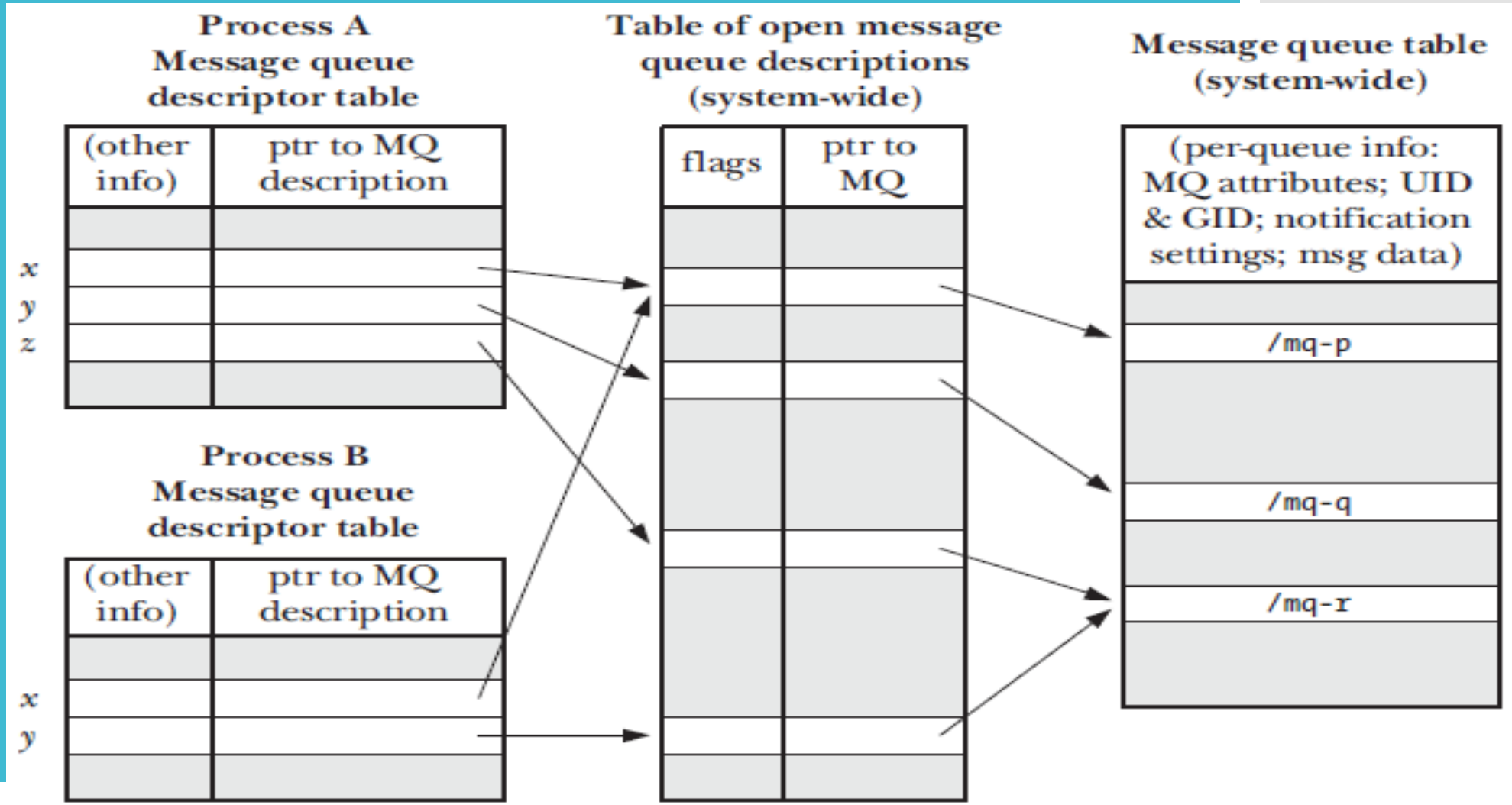
mq_unlink() — 刪除由name標識的message queue， queue會在所有processes停止使用時刪除。

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Returns 0 on success, or -1 on error

Relationship Between Descriptors and Message Queues



Message Queue Attributes

mq_open()，mq_getattr()和mq_setattr()都允許一個參數是指向mq_attr結構的指標。這個結構在<mqueue.h>中定義如下：

```
struct mq_attr {
    long mq_flags;       /* Message queue description flags: 0 or
                            O_NONBLOCK [mq_getattr(), mq_setattr()] */
    long mq_maxmsg;      /* Maximum number of messages on queue
                            [mq_open(), mq_getattr()] */
    long mq_msgsize;     /* Maximum message size (in bytes)
                            [mq_open(), mq_getattr()] */
    long mq_curmsgs;     /* Number of messages currently in queue
                            [mq_getattr()] */
};
```

Setting message queue attributes during queue creation

當我們用mq_open()建立一個message queue時，以下的mq_attr部分決定了queue的屬性：

- mq_maxmsg定義了使用mq_send()可以放在queue中的message數量限制。該值必須大於0。
- mq_msgsize定義了可以放在queue中的每個message的大小上限。該值必須大於0。

建立message queue時，mq_maxmsg和mq_msgsize屬性是固定的；他們不能在之後改變。(使用mq_setattr()設定無效)

```c
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] [-m maxmsg] [-s msgsize] mq-name "
            "[octal-perms]\n", progName);
    fprintf(stderr, "    -c          Create queue (O_CREAT)\n");
    fprintf(stderr, "    -m maxmsg   Set maximum # of messages\n");
    fprintf(stderr, "    -s msgsize  Set maximum message size\n");
    fprintf(stderr, "    -x          Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    mqd_t mqd;
    struct mq_attr attr, *attrp;

    attrp = NULL;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 2048;
    flags = O_RDWR;

    /* Parse command-line options */

    while ((opt = getopt(argc, argv, "cm:s:x")) != -1) {
        switch (opt) {
        case 'c':
            flags |= O_CREAT;
            break;

        case 'm':
            attr.mq_maxmsg = atoi(optarg);
            attrp = &attr;
            break;

        case 's':
            attr.mq_msgsize = atoi(optarg);
            attrp = &attr;
            break;

        case 'x':
            flags |= O_EXCL;
            break;

        default:
            usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");

    mqd = mq_open(argv[optind], flags, perms, attrp);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    exit(EXIT_SUCCESS);
}
```

兩個指令選項允許指定message queue屬性：-m表示mq_maxmsg，
-s表示mq_msgsize。

Retrieving message queue attributes

mq_getattr() — 回傳一個mq_attr(屬性)結構，其中包含有關message queue description和與message queue關聯的descriptor mqdes的資訊。

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Returns 0 on success, or -1 on error

除了mq_maxmsg和mq_msgsize外，attr中還會回傳以下訊息：

mq_flags

只有一個flag可設定：O_NONBLOCK。這個flag是從mq_open()的oflag參數初始化的，可以用mq_setattr()來改變。

mq_curmsgs

這是當前在queue中的message數量。如果其他process正在從queue中讀取或向queue寫入message，則此資訊在mq_getattr()返回時可能已經更改。

```c
#include <mqueue.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    mqd_t mqd;
    struct mq_attr attr;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

    mqd = mq_open(argv[1], O_RDONLY);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    printf("Maximum # of messages on queue:   %ld\n", attr.mq_maxmsg);
    printf("Maximum message size:             %ld\n", attr.mq_msgsize);
    printf("# of messages currently on queue: %ld\n", attr.mq_curmsgs);
    exit(EXIT_SUCCESS);
}
```

此程式使用mq_getattr()來取得在指令參數中指定的message queue屬性，然後輸出那些屬性。

在下面的shell session中，建立一個預設屬性的message queue，並使用剛剛的程式顯示queue的屬性，可以在Linux上看到預設設置。

```
$ ./pmsg_create -cx /mq                    c-O_CREAT
$ ./pmsg_getattr /mq                       x-O_EXCL
Maximum # of messages on queue:   10
Maximum message size:             8192
# of messages currently on queue: 0
$ ./pmsg_unlink /mq
```

從上面的輸出中，我們看到mq_maxmsg和mq_msgsize的Linux預設值分別是10和8192。

Modifying message queue attributes

mq_setattr() — 設定descriptor mqdes的屬性，並且可選擇回傳關於message queue的資訊。

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

Returns 0 on success, or −1 on error

- 使用newattr來更改flag。
- 如果oldattr不為NULL，則回傳包含先前的flag和mq_attr(屬性)結構(即，與由mq_getattr()執行的任務相同)。

Sending Messages

mq_send() — 將msg_ptr指向的buffer中的message添加到由descriptor mqdes指到的message queue中。

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
```

Returns 0 on success, or −1 on error

參數msg_len指定為msg_ptr指向的message長度。
該值必須小於或等於queue的mq_msgsize屬性；
否則，mq_send()失敗，並顯示錯誤EMSGSIZE。

- 每個message都有一個非負整數優先級，由參數msg_prio指定。message在queue中以優先級降序排列(0是最低優先級)。
- 當新message被添加到queue中時，它被放置在其他具有相同優先級的message之後。
- System V message總是以FIFO的順序排隊，但是msgrcv()允許以不同的方式選擇message。

```c
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-n] mq-name msg [prio]\n", progName);
    fprintf(stderr, "    -n           Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;

    flags = O_WRONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind + 1 >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    prio = (argc > optind + 2) ? atoi(argv[optind + 2]) : 0;

    if (mq_send(mqd, argv[optind + 1], strlen(argv[optind + 1]), prio) == -1)
        errExit("mq_send");
    exit(EXIT_SUCCESS);
}
```

此程式為mq_send()提供了一個命令界面。
稍後會搭配mq_receive()演示如何使用這個程式。

Receiving Messages

mq_receive() — 從mqdes指到的message queue中刪除具有最高優先級最早的message，並在msg_ptr指向的buffer中回傳該消息。

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);

        Returns number of bytes in received message on success, or –1 on error
```

如果message queue當前為空，則mq_receive()會阻塞(block)，直到有message；或者，如果有用O_NONBLOCK flag，則立即失敗並回傳錯誤EAGAIN。

```c
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-n] mq-name\n", progName);
    fprintf(stderr, "    -n            Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;
    void *buffer;
    struct mq_attr attr;
    ssize_t numRead;

    flags = O_RDONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

    numRead = mq_receive(mqd, buffer, attr.mq_msgsize, &prio);
    if (numRead == -1)
        errExit("mq_receive");

    printf("Read %ld bytes; priority = %u\n", (long) numRead, prio);
    if (write(STDOUT_FILENO, buffer, numRead) == -1)
        errExit("write");
    write(STDOUT_FILENO, "\n", 1);

    exit(EXIT_SUCCESS);
}
```

為mq_receive()提供了一個命令界面。該程式的命令格式顯示在usageError()中。

以下shell session演示了send和receive程式的使用。首先建立一個message queue，然後發送幾個不同優先級的message：

```
$ ./pmsg_create -cx /mq
$ ./pmsg_send /mq msg-a 5
$ ./pmsg_send /mq msg-b 0
$ ./pmsg_send /mq msg-c 10
```

然後我們執行一系列命令來從queue中取得message：

```
$ ./pmsg_receive /mq
Read 5 bytes; priority = 10
msg-c
$ ./pmsg_receive /mq
Read 5 bytes; priority = 5
msg-a
$ ./pmsg_receive /mq
Read 5 bytes; priority = 0
msg-b
```

從上面的輸出中可以看出，message按照優先級順序進行取得。

此時，queue是空的。當我們執行另一個阻塞接收(blocking receive)時，運作阻塞(block)：(用Contorl-C終止程式)

```
$ ./pmsg_receive /mq
^C
```

另一方面，如果我們執行非阻塞接收(nonblocking receive)，則立即以失敗狀態回傳：

```
$ ./pmsg_receive -n /mq                                    n - nonblocking receive
ERROR [EAGAIN/EWOULDBLOCK Resource temporarily unavailable] mq_receive
```

Sending and Receiving Messages with a Timeout

如果操作不能立即執行，並且沒有指定O_NONBLOCK flag，則abs_timeout參數指定阻塞(block)的時間限制。

```
#define _XOPEN_SOURCE 600
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned int msg_prio, const struct timespec *abs_timeout);
```

Returns 0 on success, or −1 on error

```
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                 unsigned int *msg_prio, const struct timespec *abs_timeout);
```

Returns number of bytes in received message on success, or −1 on error

如果使用mq_timedsend()或mq_timedreceive()超時而不能完成操作，則失敗，並顯示錯誤ETIMEDOUT。

Message Notification

當message到達由descriptor mqdes指到的空queue時，mq_notify()會給process一個通知。

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
                                        Returns 0 on success, or -1 on error
```

- 參數notification指定了通知process的機制。
- 可以通過將mq_notify()中的參數notification設為NULL註銷通知。

列出sigevent結構與mq_notify()相關的部分：

```
union sigval {
    int    sival_int;          /* Integer value for accompanying data */
    void  *sival_ptr;          /* Pointer value for accompanying data */
};
```

```
struct sigevent {
    int    sigev_notify;          /* Notification method */
    int    sigev_signo;           /* Notification signal for SIGEV_SIGNAL */
    union sigval sigev_value;     /* Value passed to signal handler or
                                     thread function */
    void (*sigev_notify_function) (union sigval);
                                  /* Thread notification function */
    void  *sigev_notify_attributes;   /* Really 'pthread_attr_t' */
};
```

此結構的sigev_notify被設置為以下值之一：

SIGEV_NONE、 SIGEV_SIGNAL、 SIGEV_THREAD

SIGEV_NONE

註冊這個process的通知，但是不實際通知process，當新message到達空queue時，註冊將被刪除。

SIGEV_SIGNAL

通過產生sigev_signo中指定的信號(signal)來通知process。如果sigev_signo是一個realtime signal，則sigev_value將指定伴隨信號的資料。這個資料可以通過傳遞給signal handler的siginfo_t結構的si_value來取得，也可以通過調用sigwaitinfo()或sigtimedwait()來回傳。

SIGEV_THREAD

通過使用sigev_notify_function中指定的function來通知process，就像是thread中的啟動函數一樣。sigev_value中指定的union sigval值作為此函數的參數。

```c
#include <signal.h>
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

#define NOTIFY_SIG SIGUSR1

static void
handler(int sig)
{
    /* Just interrupt sigsuspend() */
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    mqd_t mqd;
    struct mq_attr attr;
    void *buffer;
    ssize_t numRead;
    sigset_t blockMask, emptyMask;
    struct sigaction sa;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

①   mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

②   if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

③   buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

④   sigemptyset(&blockMask);
    sigaddset(&blockMask, NOTIFY_SIG);
    if (sigprocmask(SIG_BLOCK, &blockMask, NULL) == -1)
        errExit("sigprocmask");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(NOTIFY_SIG, &sa, NULL) == -1)
        errExit("sigaction");

⑤   sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = NOTIFY_SIG;
    if (mq_notify(mqd, &sev) == -1)
        errExit("mq_notify");

    sigemptyset(&emptyMask);

    for (;;) {
⑥       sigsuspend(&emptyMask);         /* Wait for notification signal */

⑦       if (mq_notify(mqd, &sev) == -1)
            errExit("mq_notify");

⑧       while ((numRead = mq_receive(mqd, buffer, attr.mq_msgsize, NULL)) >= 0)
            printf("Read %ld bytes\n", (long) numRead);

        if (errno != EAGAIN)            /* Unexpected error */
            errExit("mq_receive");
    }
}
```

Receiving Notification via a Thread

```c
#include <pthread.h>
#include <mqueue.h>
#include <signal.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void notifySetup(mqd_t *mqdp);

static void                     /* Thread notification function */
threadFunc(union sigval sv)
{
    ssize_t numRead;
    mqd_t *mqdp;
    void *buffer;
    struct mq_attr attr;

    mqdp = sv.sival_ptr;

    /* Determine mq_msgsize for message queue, and allocate an input buffer
       of that size */

    if (mq_getattr(*mqdp, &attr) == -1)
        errExit("mq_getattr");

    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

    notifySetup(mqdp);

    while ((numRead = mq_receive(*mqdp, buffer, attr.mq_msgsize, NULL)) >= 0)
        printf("Read %ld bytes\n", (long) numRead);

    if (errno != EAGAIN)                        /* Unexpected error */
        errExit("mq_receive");

    free(buffer);
}
```

① threadFunc

② notifySetup(mqdp);

```c

static void
notifySetup(mqd_t *mqdp)
{
    struct sigevent sev;

    sev.sigev_notify = SIGEV_THREAD;            /* Notify via thread */
    sev.sigev_notify_function = threadFunc;
    sev.sigev_notify_attributes = NULL;
            /* Could be pointer to pthread_attr_t structure */
    sev.sigev_value.sival_ptr = mqdp;           /* Argument to threadFunc() */

    if (mq_notify(*mqdp, &sev) == -1)
        errExit("mq_notify");
}

int
main(int argc, char *argv[])
{
    mqd_t mqd;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s mq-name\n", argv[0]);

    mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    notifySetup(&mqd);
    pause();                    /* Wait for notifications via thread function */
}
```

③ sev.sigev_notify = SIGEV_THREAD;

④ sev.sigev_value.sival_ptr = mqdp;

⑤ mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);

⑥ notifySetup(&mqd);

Displaying and deleting message queue objects via the command line

POSIX IPC物件是虛擬檔案系統中的檔案實現的，這些檔案可以用ls、rm列出和刪除。

建立一個message queue，使用ls來顯示它在檔案系統中可見，然後刪除message queue：

```
$ ./pmsg_create -c /newq
$ ls /dev/mqueue
newq
$ rm /dev/mqueue/newq
```

Obtaining information about a message queue

可以在檔案系統中顯示檔案的內容。每個檔案都包含關於message queue的資訊：

```
$ ./pmsg_create -c /mq
$ ./pmsg_send /mq abcdefg
$ cat /dev/mqueue/mq
QSIZE:7          NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

- QSIZE是queue中資料的總bytes數。如果NOTIFY_PID不為零，則有指定PID的process已經從這個queue中註冊了message通知
- NOTIFY是和sigev_notify常數之一對應的值：0表示SIGEV_SIGNAL，1表示SIGEV_NONE，2表示SIGEV_THREAD。
- 如果通知方法是SIGEV_SIGNAL，則SIGNO指示哪個signal被傳遞用於message通知。

以下shell session說明了這些顯示的資訊：

```
$ ./mq_notify_sig /mq &          Notify using SIGUSR1
[1] 2778
$ cat /dev/mqueue/mq
QSIZE:7          NOTIFY:0      SIGNO:10     NOTIFY_PID:2778
$ kill %1
[1]+  終止                    ./mq notify_sig /mq
$ ./mq notify_thread /mq &       Notify using a thread
[1] 2786
$ cat /dev/mqueue/mq
QSIZE:7          NOTIFY:2      SIGNO:0      NOTIFY_PID:2786
```

Using message queues with alternative I/O models

在Linux實現中，message queue descriptor實際上是一個file descriptor。我們可以使用select()、poll()或epoll來監視這個file descriptor。這讓我們可以避免在嘗試等待message queue和file descriptor時遇到System V message queue的困難。但是，這個功能是非標準的；SUSv3不要求message queue descriptor被實現為file descriptor。

Message Queue Limits

SUSv3為POSIX message queue定義了兩個限制：

MQ_PRIO_MAX
　　定義一個message的最大優先級。

MQ_OPEN_MAX
　　定義一個process可以保持打開的message queue的最大數量。由於Linux將message queue descriptors實現為file descriptors，所用的限制是用於file descriptors的限制。

除SUSv3指定的限制外，Linux還提供許多/ proc檔案供查看和更改控制POSIX message queue使用的限制。以下三個文件位於/proc/sys/fs/mqueue目錄中：

msg_max

此限制規定了message queue的mq_maxmsg屬性的上限。此限制的預設值為10。當一個privileged process（CAP_SYS_RESOURCE）使用 mq_open()時，msg_max限制被忽略，但HARD_MSGMAX(kernel常數)仍然是attr.mq_maxmsg的上限。

msgsize_max

此限制指定message queue的mq_msgsize屬性的上限。此限制的預設值是8192。當privileged process(CAP_SYS_RESOURCE)使用mq_open()時，此限制將被忽略。

queues_max

這是建立message queue數量的限制。達到此限制後，只有 privileged process(CAP_SYS_RESOURCE)才能建立新的queue。此限制的預設值是256。

Comparison of POSIX and System V Message Queues

與System V message queue相比，POSIX message queue具有以下優勢：

- message通知功能允許通過signal異步通知一個process，或當message到達先前為空的queue時執行thread。
- 在Linux，可以使用poll()，select()和epoll監視POSIX message queue。System V message queue不提供此功能。

與System V message queue相比，POSIX message queue也有一些缺點：

- POSIX message queue的可攜性較差。即使在Linux系統中，這個問題也適用，因為message queue僅在kernel 2.6.6之後才可用。
- 按類型選擇System V message的功能比POSIX message的嚴格優先級排序提供了更大的靈活性。

Summary

POSIX message queue每個message都有一個整數優先級，message按優先級順序排隊。

POSIX message queue可以異步地通知process在空queue上message的到達。但POSIX message queue的可攜性不如System V message queue。

POSIX SEMAPHORES

范真瑋

Overview

SUSv3指定了兩種類型的POSIX semaphores：

- Named semaphores：這種類型的semaphore有一個名字。通過以相同的名稱使用sem_open()，不相關的processes可以訪問相同的semaphore。

- Unnamed semaphores：這種類型的semaphore沒有名稱。在processes間共享時，semaphore必須駐留在共享內存的區域中。當在threads之間共享時，semaphore可以駐留在threads共享的內存區域中(例如，在heap或在全域變數中)。

一個POSIX信號是一個整數，它的值不能低於0。如果試圖將一個信號的值減到0以下，根據所使用的function，這個調用將阻塞或者失敗並產生一個錯誤，表示該操作目前不可行。

有些系統不提供POSIX信號的完整實現。一個典型的限制是只支援unnamed thread-shared semaphores。(Linux 2.4)；Linux 2.6和一個提供NPTL(Native POSIX Thread Library)的glibc，POSIX semaphores的完整實現是可用的。

Named Semaphores

sem_open() – 打開或建立信號，如果信號是由調用建立的，則初始化。

sem_post()和sem_wait() – 分別遞增和遞減一個信號的值。

sem_getvalue() – 取得信號當前的值。

sem_close() – 刪除process與之前打開的信號的關聯。

sem_unlink() – 刪除信號。

Opening a Named Semaphore

sem_open()建立並打開一個新的有名稱的信號或打開一個現有的信號。

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );

                Returns pointer to semaphore on success, or SEM_FAILED on error
```

參數name標識信號。
參數oflag是一個bit mask，如果oflag是0，表示訪問現有的信號。

如果在flag指定了O_CREAT，則要多兩個參數：mode和value。

- 參數mode是一個bit mask，用於指定要放置在新信號上的權限。bit值和開檔的相同。許多實現(包括Linux)在打開信號時都假設O_RDWR的訪問模式。

- 參數value是一個無號整數，指定要分配給新信號的初始值。信號的建立和初始化是自動執行的。這避免了System V信號初始化所需的複雜性

如果試圖在由sem_open()的回傳值所指向的sem_t變數的副本上執行操作(sem_post()，sem_wait()等等)，SUSv3指出結果是未定義的。下面的sem2的使用是不允許的：

```
sem_t *sp, sem2
sp = sem_open(...);
sem2 = *sp;
sem_wait(&sem2);
```

當通過fork()建立child時，它將繼承所有在parent中打開的已命名信號的引用。在fork()之後，parent和child可以使用這些信號來同步它們的操作。

```c
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name [octal-perms [value]]\n", progName);
    fprintf(stderr, "    -c   Create semaphore (O_CREAT)\n");
    fprintf(stderr, "    -x   Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;

    flags = 0;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':   flags |= O_CREAT;           break;
        case 'x':   flags |= O_EXCL;            break;
        default:    usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    /* Default permissions are rw-------; default semaphore initialization
       value is 0 */

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");
    value = (argc <= optind + 2) ? 0 : getInt(argv[optind + 2], 0, "value");

    sem = sem_open(argv[optind], flags, perms, value);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    exit(EXIT_SUCCESS);
}
```

程式為sem_open()提供了一個簡單的命令行界面。該程式的指令格式顯示在usageError()中。

以下shell session演示了該程式的使用。首先使用umask指令來限制(遮蔽)其他用戶的所有權限。接著，建立一個信號並檢查包含命名信號於特定Linux的虛擬目錄的內容。

```
$ umask 007
$ ./psem_create -cx /demo 666
$ ls -l /dev/shm/sem.*
-rw-rw---- 1 wei wei 32 12月 18 14:38 /dev/shm/sem.demo
```

ls指令的輸出顯示，umask更改了指定的讀取和寫入權限。

如果我們再嘗試建立一個具有相同名稱的信號，則操作將失敗，因為該名稱已經存在。

```
$ ./psem_create -cx /demo 666
ERROR [EEXIST File exists] sem_open     Failed because of O_EXCL
```

Closing a Semaphore

當process打開一個已命名的信號時，系統記錄process和信號之間的關聯。sem_close()終止這個關聯(關閉信號)，釋放系統與該process的信號相關聯的任何資源。

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns 0 on success, or −1 on error

Removing a Named Semaphore

sem_unlink()刪除name標識的信號，會在所有processes停止使用時刪除。

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns 0 on success, or −1 on error

Semaphore Operations

POSIX信號操作在以下幾個方面與System V信號不同：

- 更改信號的函數sem_post()和sem_wait()一次只能操作一個信號。相比之下，System V semop()可對一個集合中的多個信號進行操作。
- sem_post()和sem_wait()遞增和遞減一個信號的值。相比之下，semop()可以加減任意值。
- 沒有提供和System V信號等待信號為零相同功能的操作(semop()，其中sops.sem_op被指定為0)。

Waiting on a Semaphore

sem_wait()遞減(減1)由sem指到的信號的值。

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Returns 0 on success, or −1 on error

如果信號當前具有大於0的值，則sem_wait()立即回傳。如果信號的值目前是0，則sem_wait()阻塞，直到信號值超過0。

sem_trywait()是sem_wait()的非阻塞版本。

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

Returns 0 on success, or −1 on error

如果遞減操作不能立即執行，則sem_trywait()將失敗，並顯示錯誤EAGAIN。

sem_timedwait()是sem_wait()的一個變形。它允許呼叫者指定被阻塞的時間的限制。

```
#define _XOPEN_SOURCE 600
#include <semaphore.h>

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Returns 0 on success, or −1 on error

如果sem_timedwait()超時而不能減小信號，則調用將失敗，並顯示錯誤ETIMEDOUT。

Posting a Semaphore

sem_post()遞增(增加1)sem指到的信號的值。

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns 0 on success, or −1 on error

若多個processes（或threads）在sem_wait()中被阻塞，那麼如果processes在預設的round-robin分時策略下進行調度，則不確定哪一個將被喚醒並允許遞減信號。

Retrieving the Current Value of a Semaphore

sem_getvalue()將sem指向的信號的當前值放到由sval指向的整數。

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);

                                    Returns 0 on success, or -1 on error
```

如果一個或多個processes（或threads）當前被阻塞，等待減小信號的值，則在sval中回傳的值取決於實現。SUSv3允許0或負數，負數的絕對值是在sem_wait()中阻塞的等待者的數量。Linux和其他一些實現採用了前者(0)；一些其他的實現採用後者(負數)。

以下shell session演示了named semaphores程式的使用。首先建立一個初始值為零的信號，然後再執行一個背景程式試圖減小信號：

```
$ ./psem_create -c /demo 600 0
$ ./psem_wait /demo &
[1] 3046
```

背景命令阻塞，因為信號值當前為0，因此不能減少。
然後取得信號值：

```
$ ./psem_getvalue /demo
0
```

看到上面的值為0。在其他一些實現中，我們可能會看到值為-1，表示一個process正在等待信號。

執行一個增加信號的命令。這會使背景程式中阻塞的sem_wait()完成：

```
$ ./psem_post /demo
3046 sem_wait() succeeded
[1]+  Done                    ./psem_wait /demo
```

然後對信號執行進一步的操作：

```
$ ./psem_post /demo          Increment semaphore
$ ./psem_getvalue /demo
1
$ ./psem_unlink /demo
```

Unnamed Semaphores

無名稱的信號（也稱為memory-based semaphores）是儲存在由程式分配的內存的變數。信號是通過將其放置在它們共享的內存區域中，使其可用於processes或threads。

對無名稱信號的操作和有名稱的使用相同的function(sem_wait()，sem_post()，sem_getvalue())來操作信號。另外還需要兩個功能：

- sem_init()初始化信號，並通知系統信號是在processes之間還是在單個process的threads之間共享。
- sem_destroy(sem)銷毀一個信號。

Unnamed versus named semaphores

使用無名稱的信號在以下情況下，可能很有用：

- **threads之間**共享的信號不需要名稱。
- **相關processes之間**共享的信號不需要名稱。如果parent在共享內存區域中分配了一個無名稱的信號，那麼child將自動繼承映射。
- 如果構建一個**動態資料結構**（如binary tree），其中每個項目都需要相關的信號，最簡單的方法就是在每個項目中分配一個無名稱的信號。若為每個項目打開一個有名稱的信號，還需要為每個項目生成唯一的信號名稱，並管理這些名稱。

Initializing an Unnamed Semaphore

sem_init()將由sem指到的無名稱信號初始化為由value指定的值。

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
                                        Returns 0 on success, or -1 on error
```

參數pshared指示信號是在threads之間還是在processes之間共享。

- 如果pshared為0，則信號將在process的threads之間共享。
- 如果pshared非零，信號將在processes之間共享。

參數pshared是必要的，原因如下：

- 有些實現不支援process-shared信號。在這些系統上，為pshared指定一個非零值會導致sem_init()回傳一個錯誤。Linux不支援無名稱的process-shared信號，直到kernel 2.6和NPTL threading實現的出現。

- 在支援process-shared和thread-shared信號的實現上，指定需要哪種共享可能是必要的，因為系統必須採取特殊的行動來支援請求的共享。提供這些資訊也可能使系統根據共享類型做最佳化。

```c
#include <semaphore.h>
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static sem_t sem;

static void *                   /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1)
            errExit("sem_wait");

        loc = glob;
        loc++;
        glob = loc;

        if (sem_post(&sem) == -1)
            errExit("sem_post");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    /* Initialize a semaphore with the value 1 */

    if (sem_init(&sem, 0, 1) == -1)
        errExit("sem_init");

    /* Create two threads that increment 'glob' */

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    /* Wait for threads to terminate */

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

使用無名稱的thread-shared信號保護兩個threads訪問相同全域變數的critical section。

Destroying an Unnamed Semaphore

sem_destroy()銷毀信號sem，它必須是之前使用sem_init()初始化的無名稱信號。只有在沒有processes或threads在等待的情況下才能銷毀。

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Returns 0 on success, or −1 on error

在使用sem_destroy()銷毀無名稱的信號之後，可以使用sem_init()重新初始化。

POSIX semaphores versus System V semaphores

與System V信號相比，POSIX信號具有以下優點：

- POSIX信號介面比System V信號介面簡單得多。
- POSIX有名稱的信號消除了System V信號的初始化問題。

- 將一個POSIX無名稱的信號與一個動態分配的內存物件相關聯比較容易，信號可以簡單嵌入到物件內部。

- 在信號高度競爭的情況下(即信號的操作經常被阻塞)，那麼POSIX信號和System V 信號是相似的。但是，如果信號的競爭較少，POSIX信號會比System V信號好得多(在作者所測試的系統上，性能的差異超過了一個數量級)。POSIX信號在這種情況下表現得更好，因為它們的實現方式當競爭發生時只需要一個system call，System V信號操作不管競爭與否總是需要system call。

但是，與System V信號相比，POSIX信號還具有以下缺點：

- POSIX信號可攜性較差。(在Linux，從kernel 2.6起，才支援有名稱的信號)
- POSIX信號不提供與System V信號undo功能等效的功能(還原semaphore)。(但是，這個功能在某些情況下可能沒有用處)

POSIX semaphores versus Pthreads mutexes

POSIX信號和Pthreads mutexes都可以用來在同一個process中同步threads的動作，然而，mutexes通常是較好的，因為只有鎖定mutex的thread才能解鎖它。相比之下，一個thread可以遞增由另一個thread遞減的信號，這種靈活性可能導致結構不良的同步設計。

有些情況是mutexes不能用於multithread的程式，因為信號是async-signalsafe(異步信號安全)，所以可以在signal handler中使用sem_post()來與另一個thread同步。這對於mutexes來說是不可能的，因為用於在mutexes上操作的Pthreads函數不是異步信號安全的。然而，因為通過使用sigwaitinfo()接受異步信號而不使用signal handler來處理異步信號通常更好，所以很少需要信號的這個優點。

Semaphore Limits

SUSv3定義了適用於信號的兩個限制：

SEM_NSEMS_MAX

這是一個process可能擁有最大數量的POSIX信號。在Linux，POSIX信號的數量僅受可用內存有效限制。

SEM_VALUE_MAX

這是POSIX信號可能達到的最大值。信號可以從0到這個極限值。SUSv3要求此限制至少為32,767； Linux實現允許最大值為INT_MAX。

Summary

POSIX信號允許processes或threads同步他們的動作。POSIX信號有兩種類型：**named**(有名稱)和**unnamed**(無名稱)。

POSIX信號介面比System V信號介面**簡單**。信號是各自分配和操作的，而wait和post操作將信號量的值調整1(+1, -1)。

POSIX信號與System V信號相比具有許多優點，但POSIX的**可攜性較差**。對於**multithread**程式的同步，**mutexes**通常優於信號。