**CIC/UnB – Departamento de Ciência da Computação**

Noções de Sistemas Operacionais – Primeiro Semestre 2009

Professor Marco Aurelio

O presente texto é uma tradução livre do texto de **Brian E. Hall** da California State University capturado na url http://www.ecst.csuchico.edu/~beej/guide/ipc/mq.html, a respeito de filas de mensagens, e dá uma idéia inicial de suas funcionalidades. No texto original, em inglês, é possível fazer downloads dos programas aqui referidos e fazer testes com eles. Veja no Moodle o link para o original.

# Filas de Mensagens

Este documento descreve o uso e as funcionalidade do interessantíssimo sistema de filas de mensagem do Unix System V.

Vamos lançar uma visão geral antes de descer aos detalher. Uma fila de mensagens funciona como uma fila FIFO, mas oferece algumas facilidades adicionais. Em geral as mensagens são tiradas da fila na ordem em que foram colocadas. Entretanto, é possível tirar certas mensagens antes que elas atinjam a ponta.

Um processo pode criar uma nova fila de mensagens ou pode se conectar a uma já existente. Havendo dois processos conectados a um fila, eles podem começar a trocar mensagens.

Uma vez criada uma fila de mensagem ela só desaparece quando você a destroi. Mesmo que todos os processos que a usaram saiam, a fila continua a existir. Uma prática boa é usar o comento ipcs para ver se existe alguma fila de mensagem flutuando por aí. Uma fila pode ser destruída pelo comando ipcrm. .

## Onde está minha fila?

Vamos fazer a fila andar! A primeira coisa é conectar a uma fila ou criar uma, caso não exista ainda. A chamada para fazer isso é msgget( ):,

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

`msgget()` retorna a identificação da fila, em caso de sucesso ou -1, em caso de falha.

Os argumentos são um pouco estranhos, mas pode-se entendê-los com um pequeno esforço. O primeiro, *key*, é um identificador único no sistema descrevendo a fila. Qualquer outro processo que quiser se conectar a ela tem que fornecer esse mesmo número.

O outro argumento, *msgflg*, diz ao msgget ( ) o que fazer com a fila em questão. Para criar uma fila esse campo precisa ser igual ao padrão IPC_CREAT superposto (OR) com as permissões para essa fila (as permissões para uma fila são idênticas às permissões padrão para arquivos). Na sessão seguinte é apresentado um exemplo.

## "É você o mestre da fila?"

Que negócio é esse de *key*? Comos criamos uma *key*? do we create one? Bem, uma vez que o tipo *key_t* é apenas um *long*, você pode usar qualquer número que você queira. Mas, e se você escolher um número e algum outro processo escolher o mesmo número para outra fila? A solução é usar a função *ftok ( )* que gera a *key* a partir de dois argumentos:

```
key_t ftok(const char *path, int id);
```

Ok, isso está ficando estranho. Basicamente, *path* tem que ser o nome de um arquivo que esse processo possa ler. O outro argumento, *id* deve ser um caractere arbitrário, como por exemplo 'A'. A função *ftok()* usa informação sobre o arquivo indicado (como o inode number, etc.), e o *id,* para gerar uma chave provavelmente única para *msgget()*. Programas queira usar a mesma chave precisam gerar o mesmo valor, portanto, têm que passar os mesmos parâmetros para *ftok()*.

Finalmente, vamos fazer a chamada:

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

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

No exemplo acima, as permissões foram setadas para `666` (ou `rw-rw-rw-`), se isso faz mais sentido para você. E agora já temos *msqid* que vai ser usada para enviar e receber mensagens da fila... messages from the queue.

## Enviando para a fila

Uma vez conectado à fila usando *msgget()*, você está pronto para enviar e receber mensagens dela. Para enviar:

Cada mensagem é composta de duas partes, que são definidas na estrutura *struct msgbuf*, como definido em *sys/msg.h*:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

O campo *mtype* é usado mais tarde, quando recuperando mensagens da fila, e pode ter qualquer valor positivo. *mtext* contem os dados que serão colocados na fila..

"O que?! Você só pode colocar arrays de bits na fila?! Não vale a pena!!" Bem, não exatamente. Você pode usar qualquer estrutura que você quiser, desde que o primeiro elemento seja uma variável *long*. Por exemplo, você poderia usar essa estrutura para armazenar dados sobre piratas operando na costa da Somália:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    char name[30];
    char ship_type;
```

```
    int notoriety;
    int cruelty;
    int booty_value;      /* valor das mercadorias roubadas */
};
```

Ok, então como passamos essa informação para a fila de mensagens? Usemos *msgsnd():*

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

*msqid* é o identificador da fila, retornado por *msgget()*. O ponteiro *msgp* é um ponteiro para o dado que você quer colocar na fila. *msgsz* é o tamanho, em bytes, do dado a ser colocado na fila. Finalmente, *msgflg* permite definir alguns flags opcionais, que ignoraremos por enquanto, informando o valor 0.

A seguir é apresentado um código que mostra uma de nossas estruturas piratas da fila:

```
#include

key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, "L'Olonais", 'S', 80, 10, 12035};

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgsnd(msqid, &pmb, sizeof(pmb), 0);  /* stick him on the queue */
```

Afora a preocupação de verificar eventuais erros de retornos das funções, isso é tudo a se fazer. Note que o valor de *mtype* foi arbitrariamente definido como 2. Isso é importante para a próxima sessão.

# Recebendo da fila

Agora que nossa mensagem sobre o pirata "L'Olonais" está parada na fila de mensagem, como nós a tiraremos? Como você deve imaginar, existe uma contrapartida para *msgsnd()*: é *msgrcv()*.

Uma chamada a *msgrcv()* é mais ou menos assim:

```
#include

key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where the message is to be kept */

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgrcv(msqid, &pmb, sizeof(pmb), 2, 0);  /* get it off the queue! */
```

Observe o parâmetro 2 na chamada a *msgrcv()*: qual o seu significado? Aqui está o formato da chamada:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

O 2 representa o tipo de mensagem - *msgtyp*. Lembre-se que no msgsnd() nós escolhemos botar 2 no *msgsnd ( ),* de forma a requerer esse tipo de mensagem na recepção.

Na verdade, o comportamento de *msgrcv()* pode ser modificado drasticamente escolhendo-se *msgtyp* positivo, zero ou negativo:

| *msgtyp* | Efeito em msgrcv() |
|---|---|
| Zero | Recupera a próxima mensagem da fila, independente de seu mtype. |
| Positive | Pega a próxima mensagem com mtype igual ao *msgtyp*. especificado |
| Negative | Recupera a primeira mensagem na fila cujou mtype é menor ou igual ao valor absoluto de *msgtyp*. |

**Tabela 1. Efeitos de *msgtyp* em msgrcv().**

Então, se você simplesmente quiser a próxima mensagem da fila, o parâmetro mtype deve ser definido como zero.

# Destruindo uma fila de mensagens

Chegará um momento em que você deve destruir uma fila de mensagens. Isso deve ser feito para que você não fique gastando recursos do sistema. Existem duas formas para fazer isso:

1. Use o comando Unix *ipcs* para obter uma lista de filas de mensagens definidas e use o comando *ipcrm* para destruir a fila.
2. Escreva um programa para fazer isso.

Frequentemente a segunda alternativa é a melhor, uma vez que você pode querer que seu programa faça isso de vez em quando. Para isso temos que usar outra função: *msgctl()*.

Sua sinopse é:

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

O *msqid* é o identificador da fila obtido de *msgget()*. O argumento mais importante é **cmd** que diz a *msgctl()* como se comportar. Podem ser várias ações, mas falaremos agora de IPC_RMID, que é usada para remover a fila de mensagem. O argumento *buf* pode ficar NULL para os propósitos de IPC_RMID.

Digamos que nós tenhamos a fila das mercadorias piratas – podemos destruí-la assim:

```
#include
.
.
msgctl(msqid, IPC_RMID, NULL);
```

E não haverá mais fila....

# Alguém tem um exemplo de programa?

Para completar, incluiremos programas que vão comunicar-se usando filas de mensagens. O primeiro, kirk.c adiciona mensagens à fila, e spock.c as retira.

Fonte de kirk.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */
    while(gets(buf.mtext), !feof(stdin)) {
        if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

A forma que `kirk` funciona permite que você entre linhas de texto.Cada linha é colocada em uma mensagem e adicionada à fila de mensagens, que será lida por `spock`.

Fonte para spock.c:

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};
```

```
int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock: ready to receive messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

Note que `spock`, na chamada a *`msgget()`*, não inclui a opção *`IPC_CREAT`*. Nós deixamos isso ao encargo de `kirk. Spock` vai retornar um erro se KIRK não tiver feito isso.

Note o que acontece quando você executa ambos em janelas separadas e você mata um ou o outro. Também tente rodar duas cópias de `kirk` ou de `spock` para ter uma idéia do que acontece quando você tem dois leitores ou dois escritores. Outra demonstração interessante é rodar `kirk`, entrar um monte de mensagens, então rodar `spock` e vê-lo recuperando todas as mensagens de vez. Brincando com esses dois programas você vai entender melhor o que acontece nas filas.

## Conclusões

Existe muito mais coisas a respeito de filas de mensagens do que esse pequeno tutorial mostra. Estude melhor as páginas de manual das funções, especialmente as de *`msgctl()`*. Existem também mais opções para controlar o que acontece com *`msgsnd()`* e *`msgrcv()`* quando a fila está cheia ou vazia, respectivamente.

## HPUX man pages

*If you don't run HPUX, be sure to check your local man pages!*

- `ftok()`
- `ipcs`
- `ipcrm`

- `msgctl()`
- `msgget()`
- `msgsnd()`

---

Back to the IPC main page (http://www.ecst.csuchico.edu/~beej/guide/ipc/)