

Systemsoftware Praktikum Ausarbeitung

Marcus Fritsch und Markus Korzendorfer

m@fritschy.de markus@dselect.de

WS 2007

Inhaltsverzeichnis

1	Einführung	5
2	Semaphoren	6
3	Message Queues	8
4	Implementierungsvariante Message Queue	10
4.1	System V Release 4 (SVR4)	10
4.2	POSIX 1003.1 (Realtime Extensions)	12
5	Shared Memory	14
5.1	Shared Memory Objekte nach POSIX 1003.1	14
5.2	Shared Memory Objekte nach System V Release 4 (SVR4)	15
6	Zusammenfassung	17
7	Programmbeschreibung	19
7.1	Messagequeue-Datenstruktur und Toplevel-Funktionen	19
7.1.1	Message Queue Objekt	19
7.1.2	Nachrichtentypen	19
7.1.3	Message objekt	19
7.1.4	Toplevel Message Queue Funktionen	20
	open_msq ()	20
	send_msq ()	20
	recv_msq ()	21
	rm_msq ()	21
	equal_msq ()	21
	fail_msq ()	22
7.2	Message Queue Abstraktionsschichten	22
7.2.1	Toplevel Interface	23
7.2.2	Message Queue API Schicht	23
7.2.3	Queue Datenstruktur Schicht	24
7.2.4	Queue Implementierungsdetails	25
7.2.5	Nachricht	25
7.3	Emulierte Message Queue Datenstrukturen	26

8	Kommunikationsprotokoll	27
8.1	Client-Serverkommunikationssequenz	27
9	Aufrufgraphen	29
9.1	Messagequeue Toplevel Functions	29
9.1.1	open_msq ()	29
9.1.2	send_msq ()	29
9.1.3	recv_msq ()	29
9.1.4	rm_msq ()	30
9.1.5	equal_msq ()	30
9.2	Low Level Message Queue Data Structure	30
9.2.1	put_mq ()	30
9.2.2	get_mq ()	30
9.3	Client And Server Functions	31
9.3.1	client ()	31
9.3.2	server ()	32
9.4	Programmbenutzug	32
9.4.1	Programmaufruf	32
10	Source Code	34
10.1	Makefiles	34
10.1.1	chat/include/Makefile	34
10.1.2	chat/Makefile	34
10.1.3	chat/src/Makefile	34
10.2	Header Files	35
10.2.1	chat/include/client.h	35
10.2.2	chat/include/emumsq.h	35
10.2.3	chat/include/list.h	36
10.2.4	chat/include/message.h	37
10.2.5	chat/include/mq.h	37
10.2.6	chat/include/msq.h	38
10.2.7	chat/include/posixmsq.h	39
10.2.8	chat/include/server.h	39
10.2.9	chat/include/sysvmsq.h	39
10.2.10	chat/include/sysvsem.h	40
10.2.11	chat/include/sysvshm.h	41
10.2.12	chat/include/util.h	42
10.3	Implementations	42
10.3.1	chat/src/client.c	42
10.3.2	chat/src/emumsq.c	44
10.3.3	chat/src/list.c	45
10.3.4	chat/src/main.c	46
10.3.5	chat/src/mq.c	48
10.3.6	chat/src/msq.c	48

Inhaltsverzeichnis

10.3.7	chat/src/posixmsq.c	51
10.3.8	chat/src/server.c	52
10.3.9	chat/src/sysvmsq.c	54
10.3.10	chat/src/sysvsem.c	55
10.3.11	chat/src/sysvshm.c	56
10.3.12	chat/src/util.c	56

1 Einführung

Diese Arbeit entstand im Rahmen der Vorlesung System Software. Unsere Aufgabe war es ein Chatprogramm zu programmieren. Dieses sollte einen bestimmten Rahmen genügen.

Es soll ein Publish-And-Subscribe-Service lokal als eigenständiger Prozess implementiert werden und die Funktionalität mit mindestens zwei weiteren Client Prozessen (Publisher und Subscriber) implementiert werden. Die hier notwendige Kommunikation sollte auf folgende Art umgesetzt werden:

- beide Varianten Message Queue (SystemV und POSIX)
- Shared Memory mit eigener Synchronisation (SystemV oder POSIX)

Die Aufgabenstellung sollte modular umgesetzt werden und im Speziellen das Monitor Prinzip beachtet werden.

In den folgenden Kapiteln werden Semaphoren, Message Queues, der Nachrichtenaustausch mit Shared Memory und schliesslich der Unterschied zwischen Posix und SystemV erklärt. Der zweite Teil der Ausarbeitung beschäftigt sich mit der Dokumentation unserer Überlegungen die Anforderungen umzusetzen. Im Anhang findet sich dann der ausführliche Code bezüglich der Implementierung.

2 Semaphoren

Zum Erzeugen eines neuen Prozesses braucht man mit *fork(2)* nur wenige Zeilen Code, da das Betriebssystem die Initialisierung und das Verwalten der Prozesse übernimmt.

Diese ist eine grundlegende Funktion des Betriebssystems. Es ist der Überwacher aller Prozesse, die daher in einer eigenen Umgebung ausgeführt werden. Dadurch, dass das Betriebssystem die Kontrolle hat, entsteht für den Entwickler ein Problem: Wie ist es möglich, dass unabhängige Prozesse zusammenarbeiten?

Das Problem ist komplexer als es zunächst aussieht. Es ist nicht nur eine Frage der Synchronisation von Prozessen, sondern auch, wie man Daten teilen kann und gemeinsam lesen und schreiben.

Wenn zwei Prozesse dieselben Daten nur lesen, dann ist das offensichtlich kein Problem. Nun soll einer der beiden Prozesse die Daten modifizieren: Der andere wird unterschiedliche Daten sehen in Abhängigkeit davon, wann er auf die Daten zugreift, vor oder nach dem Schreiben des anderen Prozesses. Nehmen wir z.B. die zwei Prozesse A und B und die ganze Zahl d . Prozess A erhöht sie um eins und Prozess B druckt sie aus. In einer symbolischen Schreibweise kann man das so ausdrücken:

$$A \{ d \rightarrow d + 1 \} \& B \{ d \rightarrow output \}$$

das $\&$ beschreibt hier gleichzeitige Ausführung. Eine mögliche Lösung ist:

$$(-)d = 5 \text{ (A) } d = 6 \text{ (B) } output = 6$$

aber wenn B zuerst ausgeführt wird, dann erhalten wir:

$$(-)d = 5 \text{ (B) } output = 5 \text{ (A) } d = 6$$

Man versteht sofort, wie wichtig es ist, diese Situation richtig handzuhaben. Das Risiko, inkonsistente Daten zu erhalten, ist groß und nicht akzeptabel. Man denke nur daran, dass die Daten auch das Geld auf einem Bankkonto darstellen könnte und man wird das Problem nie unterschätzen.

Die einfachste Synchronisationsmethode ist wohl die *waitpid(2)* Funktion. Mit ihr kann

2 Semaphoren

ein Prozess auf den anderen warten, bis er fertig ist. Damit kann man schon einige der Konflikte beim Datenzugriff lösen: Prozess P2 kann warten, bis P1 mit dem Modifizieren der Daten fertig ist und dann selbst fortfahren.

Das ist sicher eine Lösung, aber nicht die Beste, weil P2 warten muss, bis P1 fertig ist, selbst wenn P1 nicht mehr an gemeinsamen Daten arbeitet, sondern etwas anderes macht. Es ist daher sinnvoll die Granularität zu erhöhen.

Die Idee einer Semaphore kann ohne große Modifikation für die Zugriffskontrolle auf Daten verwendet werden. Eine Semaphore ist eine Struktur mit einer Zahl größer oder gleich Null und damit werden eine Reihe von Prozessen verwaltet, die auf eine Zugriffserlaubnis warten. Semaphore sind sehr leistungsfähig und damit auch komplexer. Semaphore kann man verwenden, um Zugriffe zu verwalten: Der Wert der Semaphore beschreibt die Zahl der Prozesse, die Zugriff auf Daten oder andere Dinge haben. Jedes Mal, wenn ein Prozess Zugriff erhält, wird die Semaphore heruntergezählt und wenn der Zugriff wieder freigegeben wird, zählt man hoch. Sollen exklusive Zugriffe vergeben werden, so ist der Anfangswert 1 (nur ein Prozess kann zu jeder Zeit zugreifen).

3 Message Queues

Die Benutzung von Semaphoren erweist sich als zu komplex und zu begrenzt: komplex, weil jeder Person mit denen der anderen Prozesse synchronisieren müsste. Eingeschränkt, da es keinen Austausch von Parametern zwischen den Prozessen gestattet. Nimmt man als Beispiel den Aufbau eines neuen Prozesses, so sollte jeder beteiligte Prozess über das Ereignis informiert werden. Semaphoren erlauben es einem Prozess nicht solche Informationen auszutauschen.

Hiermit wird klar, dass die Semaphoren ein ungeeignetes Werkzeug zur Behandlung von komplexen Synchronisationsproblemen sind. Eine elegante Lösung für diese Aufgabe erhalten wir durch die Anwendung von *Message Queues*.

Jeder Prozess kann eine oder mehrere Datenstrukturen aufbauen, sie werden *Queue* genannt: jede Datenstruktur kann eine oder mehrere Messages unterschiedlicher Art von verschiedenen Quellen enthalten. Jeder beteiligte Prozess kann Messages in die Queues schicken, vorausgesetzt er kennt deren Identifikatoren. Der Prozess hat sequentiellen Zugriff (von der ältesten, zuerst angekommen bis zur jüngsten, zuletzt eingetroffenen) auf die Queue, er liest die Messages in chronologischer Reihenfolge selektiv, d.h. nur Messages von einem gewissen Typ werden erwogen: diese letztere Eigenschaft ermöglicht eine gewisse Kontrolle über die Priorität der zu lesenden Messages.

Der Gebrauch von Queues ist insofern die einfache Anwendung eines Mailsystems zwischen Prozessen: jeder Prozess hat eine Adresse und er kann mit anderen Prozessen korrespondieren. Ein Prozess kann also Messages, die an ihn gesandt werden, in einer bestimmten Reihenfolge lesen und entsprechend den vorgefundenen Anforderungen handeln.

Die Synchronisation zweier Prozesse kann infolgedessen einfach durch Messages zwischen ihnen erfolgen: Ressourcen besitzen ausserdem Semaphoren, welche die Prozesse über ihren Status informieren, der zeitliche Ablauf zwischen den Prozessen wird jedoch direkt durchgeführt. Hier wird sofort verständlich, dass der Gebrauch von Message Queues sehr vereinfacht, was anfangs als ein äusserst komplexes Problem erschien.

Ein noch erwähnenswertes Problem bezüglich der Synchronisation bezieht sich auf die Notwendigkeit eines Kommunikationsprotokoll.

3 Message Queues

Ein Protokoll ist eine Reihe von Regeln, welche die Zusammenarbeit von Elementen in einem Set behandeln. Die Anwendung von Message Queues erlaubt es uns komplexe Protokolle anzuwenden: man stelle sich vor, dass jedes Netzwerkprotokoll (TCP/IP, DNS, SMTP, ..) auf einer *message exchange architecture* aufgebaut ist, selbst wenn die Kommunikation zwischen Rechnern und nicht innerhalb ihrer Systeme stattfindet. Der Vergleich ist zwingend: es besteht kein grundlegender Unterschied zwischen der Kommunikation von Prozessen in einer Maschine oder zwischen verschiedenen Rechnern. Hier ein Beispiel:

Prozess B:

- Verarbeitete die Daten
- Wenn fertig, dann Message zu Prozess A
- Wenn A antwortet, schicke Resultat

Prozess A:

- Verarbeite die Daten
- auf Message von B warten
- auf Message antworten
- Daten von B empfangen und mit eigenem Result kombinieren

Die Bestimmung welcher Prozess die Daten kombinieren muss, ist in diesem Fall willkürlich, normalerweise ergibt sich das aus der Funktion der beteiligten Prozesse (z.B. Client/Server).

Dieses Protokoll kann einfach auf n Prozesse ausgedehnt werden: jeder Prozess, ausser A, bearbeitet seine eigenen Daten und schickt Message an A. A antwortet und jeder Prozess sendet seine Ergebnisse: die Struktur der individuellen Prozesse - ausser A - bleibt unverändert.

4 Implementierungsvariante Message Queue

4.1 System V Release 4 (SVR4)

Zum Erzeugen einer Message Queue dient die Funktion `msgget()`;

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

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

Der Rückgabewert von `msgget` ist der Handler der durch einen *Key* indentifizierten Message Queue. Wenn die Message Queue nur von verwandten Prozessen benutzt werden soll, kann als *Key* der Wert `IPC_PRIVATE` verwendet werden. Sollen nicht verwandte Prozesse die Queue nutzen, empfiehlt sich die Verwendung der Funktion `ftok()` zur Erzeugung des *Keys*:

```
\key_t ftok(const char *pathname, int proj_id);
```

Der Parameter `msgflg` von `msgget()` steuert die Zugriffsrechte und die Art der Erzeugung der Message Queue, ähnlich wie bei `open()`. Die einzelnen Optionen werden über ein bitweises OR kombiniert. Die Option `IPC_CREAT` sorgt dafür, dass die Queue erzeugt wird, wenn sie noch existiert, die Option `IPC_EXCL` sorgt dafür, dass `msgget()` fehlschlägt, wenn die Queue bereits existiert.

Die Kommunikation über Message Queues erfolgt nicht wie bei Pipes direkt, sondern über Nachrichten (Messages). Eine Nachricht besteht aus einem Nachrichtenkopf (*header*) und einem Nachrichtentext (*body*). Der *header* besteht aus einer echt positiven Integer Zahl, die den Typ der Nachricht beschreibt. Der *body* besteht aus einem Character Array (nahazu) beliebiger, aber fast vorgegebener Größe. Eine Nachricht wird durch einen *struct* beschrieben:

4 Implementierungsvariante Message Queue

```
struct msgbuf{
    int mtype;
    char mtext[MSG_SIZE];
}
```

Zum Senden und Empfangen von Nachrichten dienen die Funktionen `msgsnd()` und `msgrcv()`;

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
    long msg-typ, int msgflg);
}
```

Der erste Parameter beschreibt die Queue, die verwendet werden soll, der zweite die zu sendende bzw. zu empfangende Nachricht. Der Parameter `msgsz` gibt die maximale Größe des Elements `mtext` an, der Parameter `msgflg` beschreibt das Verhalten der Funktionen. Ist `msgflg` gleich 0, so blockieren die Funktionen den ausführenden Prozeß wenn nicht gesendet oder empfangen werden kann. Dieses Verhalten kann durch das Setzen von `msgflg` auf `IPC_NOWAIT` geändert werden.

Der Parameter `msg-typ` von `msgrcv()` hat folgende Bedeutung:

- gleich 0: Die erste Nachricht in der Queue wird empfangen
- größer 0: Die erste Nachricht mit `mtype = msg-typ` wird empfangen
- kleiner 0: Die erste Nachricht mit dem kleinsten `mtype` kleiner oder gleich dem Absolutbetrag von `msg-typ` wird empfangen.

Eine Message Queue kann mit der Funktion `msgctl()` entfernt werden:

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

Wird `cmd` auf `IPC_RMID` gesetzt entfernt der Aufruf der Funktion die mit `msqid` referenzierte Message Queue.

Zu beachten ist, dass `msgsnd()` und `msgrcv()` durch ein Signal wie `SIGCONT` abgebrochen werden und -1 zurückliefern. Daher sollte man nach einem fehlgeschlagenen `msgsnd()` bzw. `msgrcv()` den Fehlercode `errno` auf `EINTR` prüfen und gegebenenfalls `msgsnd()` bzw. `msgrcv()` ein zweites mal probieren. Dies kann durch folgenden Quellcode durchgeführt werden:

```
if (msgrcv(...)==1
{
    if ((errno!=EINTR) || (msgrcv(...)==-1)
    {
        printf("\nFehler beim Lesen aus Queue: %s\n\n", sterror(errno));
    }
}
```

4.2 POSIX 1003.1 (Realtime Extensions)

POSIX Message Queue Prozesse ermöglichen den Austausch von Daten in Form von Nachrichten. Die API unterscheidet sich von der von System V, bietet aber ähnliche Funktionalitäten.

Hier der vollständige Code zum Erstellen einer Message Queue mit `mq_open`:

```
#include <unistd.h>
#ifdef _POSIX_MESSAGE_PASSING
#include <mqueue.h>

mqd_t mymq;
struct mq_attr my_attrs;
mode_t permissions;
char *mq_name;
char *msgbuf;
int msgsize;
unsigned int msgprio;

mqd_t mq_open(const char *mq_name, int oflag, mode_t create_mode,
              struct mq_attr *create_attrs);
```

Die `mq_open` liefert einen Deskriptor `mqd_t`, mit dem man auf die Message Queue zugreifen kann.

Nachrichten werden mit `mq_send` übertragen und mit `mq_receive` empfangen:

```
int mq_send(mqd_t message_queue, const char *message_data,
            size_t message_data_length, unsigned int priority);
```

4 Implementierungsvariante Message Queue

Außer den *priority* Parameter sieht der Syntax genau wie beim Schreiben aus. Dieser Aufruf sendet eine einzelene Nachricht, `message_data`, mit einer Länge von `message_data_length` zu der im Aufruf stehenden Message Queue.

```
size_t mq_receive(mqd_t message_queue, const char *message_buffer,  
                 size_t buffer_size, unsigned int *priority);
```

Dieser Aufruf verschiebt die Nachricht an den Anfang der `message queue`, und legt ihn in `message_buffer`. `message_size` lässt das System wissen, wie groß der `message_buffer` ist.

Jede Nachricht hat immer eine zugehörige Priorität und wird an die Message Queue mit der höchsten Priorität gesendet. Diese reicht von 0 (niedrig) bis `sysconf(_SC_MQ_PRIO_MAX) - 1` (hoch).

Mit `mq_close` wird eine Message Queue geschlossen, mit `mq_unlink` die Ressourcen vom System entfernt.

5 Shared Memory

Hierbei handelt es sich, wie der Name schon andeutet, um Speicherbereich, der von verschiedenen Prozessen geteilt wird. Ein Shared Memory Segment ist - ähnlich wie eine Message Queue - als Kernelstruktur realisiert und wird mit

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

erzeugt. Die Bedeutung der Parameter angefügt werden. Der erste Parameter ist dabei eine ID, die z.B. mit `shmget` erhalten wurde. Der zweite Parameter ist normalerweise Null und kann benutzt werden, um die Adresse, an der das Segment eingehängt wird vorzuschreiben. Der dritte Parameter ist Null für Lese- und Schreibzugriffe (dann benötigt der Prozeß die entsprechenden Rechte). Soll aus dem Segment nur gelesen werden, so kann das Flag `SHM_RDONLY` benutzt werden. Die Routine gibt einen Zeiger auf die Adresse zurück, an der das Segment eingehängt wurde. Auf diesen Bereich kann anschliessend wie auf eine gewöhnliche Variable zugegriffen werden. Anschliessend kann mit dem Befehl

```
int shmdt (char *shmaddr)
```

das Segment wieder abgehängt werden. Ähnlich wie bei Message Queue muss ein einmal eingerichtetes Segment auch wieder entfernt werden, was mit

```
shmctl(int shmid, IPC_RMID, 0);
```

erreicht werden kann.

5.1 Shared Memory Objekte nach POSIX 1003.1

Dies sind in der Regel Dateien, die in den Adressraum der Prozesse eingebunden werden. Die Funktionen sind in `<sys/mman.h>` deklariert.

Unterstützt werden diese Shared Memory Objekte von IRIX (6.5.6), Linux (2.2.18), QNX(6.1.0) und SunOS (5.7).

Shared Memory Objekte werden mit

5 Shared Memory

```
shmFD=shm_open(*name, O_RDWR|O_CREATE|O_EXCL,  
S_IRUSR|S_IWUSR)
```

kreiert. `*name` ist ein Zeiger auf einen String mit dem Namen des Shared Memory Objektes. Nach POSIX 1003.1 muss dieser Name mit einem Backslash beginnen und darf keinen weiteren Backslash enthalten. IRIX (6.5.6) weicht von dieser Definition ab. Dort wird der Dateiname incl. Pfad angegeben. Unter IRIX sollte daher nur der Name des Objektes oder `/tmp/name` angegeben werden.

Nach dem Erstellen eines neuen Shared Memory Objektes muss dessen Größe mit `ftruncate(shmFD, Objektgröße)` festgelegt werden.

Zum Öffnen eines vorhandenen Shared Memory Objektes verwendet man `shmFD=shmopen(*name, O_RDWR, S_IRUSR|S_IWUSR)`.

Das geöffnete Shared Memory Objekt wird dann mit

```
shmAdresse=mmap(0, Objektgröße, PROT_READ|PROT_WRITE,  
MAP_SHARED, shmFD, 0)
```

in den Adressraum des Prozesses eingebunden.

Mit `fork()` kreierte Kinderprozesse erben den Zugriff auf das Shared Memory Objekt. Starten die Kindprozesse jedoch mit `exec()` ein anderes Programm, so gehen die Zugriffsrechte verloren. D.h. das neu gestartete Programm kann nicht auf das Shared Memory Objekt zugreifen. Es ist auch nicht möglich den von `shm_open` zurückgelieferten Filedeskriptor an das neue Programm zu übergeben. Das neue Programm muss, um auf das Shared Memory Objekt zugreifen zu können, dieses mit `shm_open` öffnen und mit `mmap()` in den eigenen Prozessadressraum einbinden.

Mit `shm_unlink(*name)` wird das Shared Memory Objekt entfernt. Der belegte Speicher eines Shared Memory Objektes wird jedoch erst freigegeben, wenn alle Filedeskriptoren das Shared Memory Objektes geschlossen und alle Mappings entfernt sind.

5.2 Shared Memory Objekte nach System V Release 4 (SVR4)

In SVR4 sind die Shared Memory Objekte in `<sys/shm.h>` beschrieben.

5 Shared Memory

Die Shared Memory Objekte nach SVR4 werden von IRIX (6.5.6), Linux (2.2.18) und SunOS (5.7) unterstützt.

Das Erstellen eines neuen Shared Memory Objektes erfolgt mit

```
shmID=shmget(IPC_PRIVATE, Objektgröße, IPC_CREAT|  
IPC_EXCL|S_IWUSR|S_IRUSR)
```

Die erhaltene Kennung `shmAdresse=shmat(shmID, NULL, 0)` wird das Shared Memory Segment in den Prozessadressraum eingebunden.

Vor Programmende entfernt man das Shared Memory Segment durch `shmctl(shmID, IPC_RMID, NULL)`

6 Zusammenfassung

Für die Message Queues gibt es unter den verschiedenen Unix-Derivaten zwei verschiedene Möglichkeiten der Implementierung:

- Nach *POSIX 1003.1 (Realtime Extensions)* gibt es Message Queues mit den in `<mqqueue.h>` deklarierten Funktionen `mq_open()`, `mq_close()`, `mq_send()` und schließlich `mq_receive()`.

Diese Message Queues können als FIFO Queue oder als prioritätsgesteuerte FIFO Queue verwendet werden. Unterstützt werden diese Message Queues von *IRIX (6.5.6)*, *QNX (6.1.0)* und *SunOS (5.7)*.

- In *System V Release (SVR4)* gibt es es Message Queues mit den Funktionen `msgget()`, `msgctl()`, `msgsnd()` und `msgrcv()`, die in `<sys/msg.h>` deklariert sind.

Diese Message Queues haben den Vorteil, das die Messages mit einer Typkennung versehen werden können. Beim Empfang von Messages kann man angeben, dass man nur Messages eines bestimmten Typs empfangen will. Weißt man jedem lesendem Prozess einen eigenen Typ zu, z.B. seine PID, so können mehrere Prozesse die gleiche Message Queue verwenden und bekommen doch nur die für sie bestimmten Messages.

Diese *SVR4* Message Queues werden von *IRIX (6.5.6)*, *Linux (2.2.18)* und *SunOS (5.7)* unterstützt.

Ein großer Nachteil bei der Verwendung von Message Queues ist die Gefahr von Deadlocks. Diese können auftreten, wenn:

- ein Prozess Messages in die gleiche Message Queue schreibt, aus der er auch liest.
- ein Prozess, der Messages empfängt, auch Messages sendet und vom Betriebssystem eine Begrenzung der Gesamtzahl aller Messages im System besteht.

Die Implementierung von Message Queues mit Shared Memory und Semaphoren ist ein wenig aufwendiger als die Verwendung von Pipes. Vor allem muss man darauf achten, die belegten Ressourcen bei einem Programmende oder- abbruch wieder freizugeben. Unter *IRIX*, *LINUX* und *Solaris* kann man sich die belegten IPC-Ressourcen mit dem Shell Kommando `ipcs` anschauen, und die in der Programmentwicklungsphase entstandenen aber nicht gelöschten Ressourcen mit `ipcrm` löschen. Unter *QNX* sind die Shared Memory

6 Zusammenfassung

Objekte unter `/dev/shmem` zu finden. Das fertige Programm muss natürlich alle belegten Ressourcen selbst wieder freigeben.

Der große Vorteil der Benutzung von Message Queues oder Shared Memory und Semaphoren zur Interprozesskommunikation ist, dass auch nicht verwandte Prozesse miteinander kommunizieren können.

7 Programmbeschreibung

7.1 Messagequeue-Datenstruktur und Toplevel-Funktionen

7.1.1 Message Queue Objekt

`typedef struct msq msq_t` Die Messagequeue Datenstruktur sieht wie folgt aus:

type spezifiziert den Message Queue API Typ

msq enthält die Datenmember für die jeweilige API

sysv enthalet den System V Sempahore Array Identifier

ptr zeigt auf eine generelle Message Queue Implementierung

7.1.2 Nachrichtentypen

Gültige Nachrichtentypen sind

MTYPE_TEXT eine normale Textnachricht, im payload steht der Text als null terminierter `char *`.

MTYPE_JOIN eine Teilnahme Nachricht, wird zu Beginn einer Session von einem Client an den Server gesendet

MTYPE_EXIT eine beendete Nachricht wird vom Client an den Server verschickt um das Verabschieden zu signalisieren

MTYPE_PING eine ping Nachricht an den Server, im payload steht ein `struct timeval` welche vom Server in die Antwort Nachricht kopiert wird.

7.1.3 Message objekt

`typedef struct message message_t`

Das Nachrichtenobjekt sieht wie folgt aus:

message_t
+ type : int
+ sender : char [CHAT_NAME_LEN]
+ payload : char [MAX_MSG_SIZE - CHAT_NAME_LEN - sizeof (int)]

type definiert die art der Nachricht

sender ist ein Chat Partner Identifizierer (Nickname)

payload ist der eigentliche Text oder Nutzdaten der Nachricht

7.1.4 Toplevel Message Queue Funktionen

open_msq ()

```
msq_t *open_msq (const char *name, const int flags, const int id,  
                const msq_apo_t type)
```

- Beschreibung

Öffnet eine Message Queue des Systems, dabei sind folgende Parameter relevant:

name Pfadname des Message Queue Objekts

flags *flags* und *modes* um ein Message Queue Objekt zu öffnen bzw, es zu erstellen.
Siehe auch *semget(2)*.

id id des ipc Objekts auf dem Pfadname

type API Typ

Rückgabewert der Funktion ist ein Pointer auf ein neues msq_t objekt.

- Fehler werden durch `msq_t.type == -1` signalisiert.

send_msq ()

```
int send_msq (const msq_t * msq, const message_t * msg)
```

- Beschreibung

Senden einer Nachricht.

msq Message Queue Objekt das die Nachricht erhalten soll

msg Nachricht

- Fehler werden von der darunter liegenden Schicht heraufpropagiert; Fehler sollten durch einen Rückgabewert von -1 signalisiert werden.

recv_msq ()

```
ssize_t recv_msq (const msq_t * msq, message_t * msgbuf)
```

- Beschreibung
Empfangen einer Nachricht
msq Message Queue Objekt von der eine Nachricht empfangen werden soll
msgbuf ein gültiger Zeiger auf ein Message Objekt, die empfangene Nachricht wird hier gespeichert
- Fehler werden von der darunter liegenden Schicht heraufpropagiert; Fehler sollten durch einen Rückgabewert von -1 signalisiert werden.

rm_msq ()

```
int rm_msq (const msq_t * msq)
```

- Beschreibung
Löschen eines Message Queue Objektes und der dazugehörigen Systemressourcen.
msq Message Queue Objekt das gelöscht werden soll
- Fehler werden von der darunter liegenden Schicht heraufpropagiert; Fehler sollten durch einen Rückgabewert von -1 signalisiert werden.

equal_msq ()

```
int equal_msq (const msq_t * a, const msq_t * b)
```

- Beschreibung
Vergleicht man zwei msq_t Objekte auf Gleichheit, wird nur im Falle einer Gleichheit *true* zurückgegeben.
a erstes msq_t Objekt
b zweites msq_t Objekt

- Fehler; es gibt keine Fehler die signalisiert werden, Für kritische Fehler, z.B.:
`a == NULL || b == NULL` wird eine Warnung ausgegeben und `false` zurückgegeben.

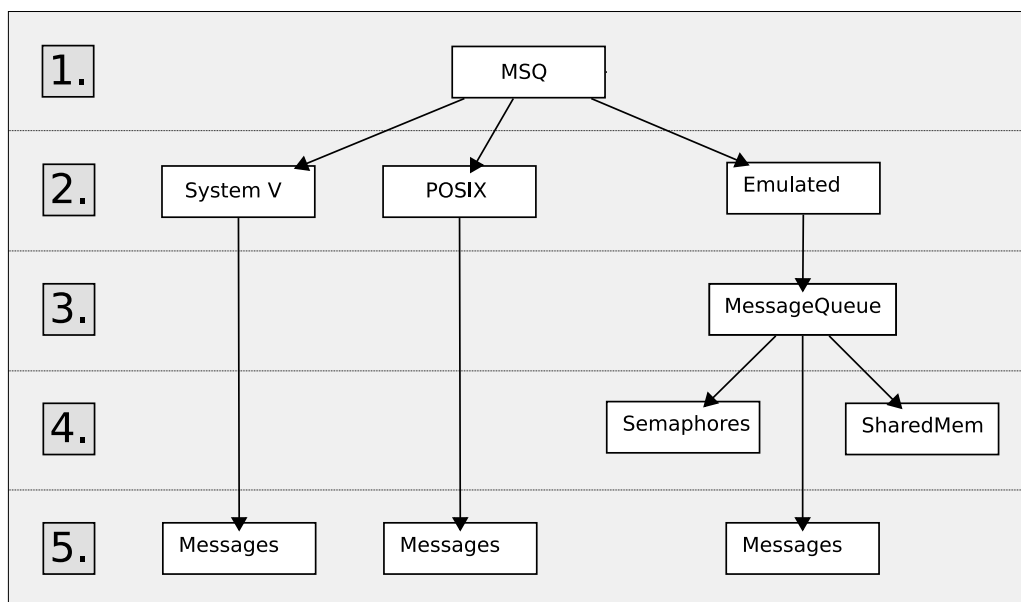
`fail_msq ()`

```
int fail_msq (const msq_t * msq)
```

- Beschreibung
ermittle Gültigkeit eines `msq_t` Objektes
msq message queue
- Fehler werden nicht signalisiert. Es wird eine Warnung ausgegeben und `false` zurückgegeben falls `msq == NULL` ist.

7.2 Message Queue Abstraktionsschichten

Im Folgenden werden die einzelnen Abstraktionsschichten erläutert und zu den anderen Schichten in Relation gestellt.



7.2.1 Toplevel Interface

Höchste Abstraktion, Clients des Message Queue Systems werden nicht mit tieferliegenden Schichten arbeiten. Hier angesiedelte Funktionen sind in Kapitel 7.1 ausführlich beschrieben.

7.2.2 Message Queue API Schicht

Diese Schicht repräsentiert die Verbindung zu den einzelnen Message Queue Implementierungen, also entweder System V oder Emuliert.

System V funktionen dieser Schicht sind:

open_sysv_msq () öffnen einer Message Queue
send_sysv_msq () senden einer Nachricht
recv_sysv_msq () empfangen einer Nachricht
rm_sysv_msq () löschen einer Message Queue
equal_sysv_msq () Gleichheitstest

Emulierte (auf System V shared memory bauende) Message Queue Funktionen dieser Schicht.

open_emulated_msq () öffnen einer Message Queue
send_emulated_msq () senden einer Nachricht
recv_emulated_msq () empfangen einer Nachricht
rm_emulated_msq () löschen einer Message Queue
equal_emulated_msq () Gleichheitstest

Die dazu gehörige Datenstruktur ist die `emsq_t`, siehe Grafik weiter unten.

POSIX Message Queue Funktionen dieser Schicht.

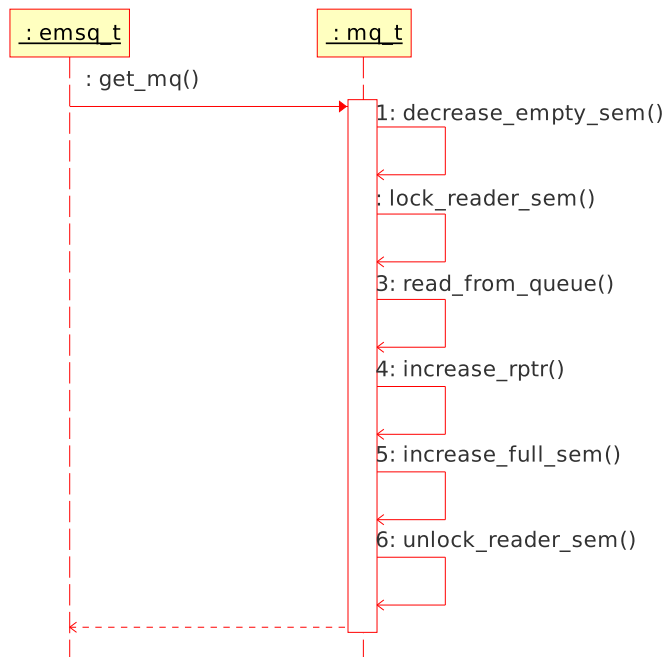
open_posix_msq () öffnen einer Message Queue
send_posix_msq () senden einer Nachricht
recv_posix_msq () empfangen einer Nachricht
rm_posix_msq () löschen einer Message Queue
equal_posix_msq () Gleichheitstest

7.2.3 Queue Datenstruktur Schicht

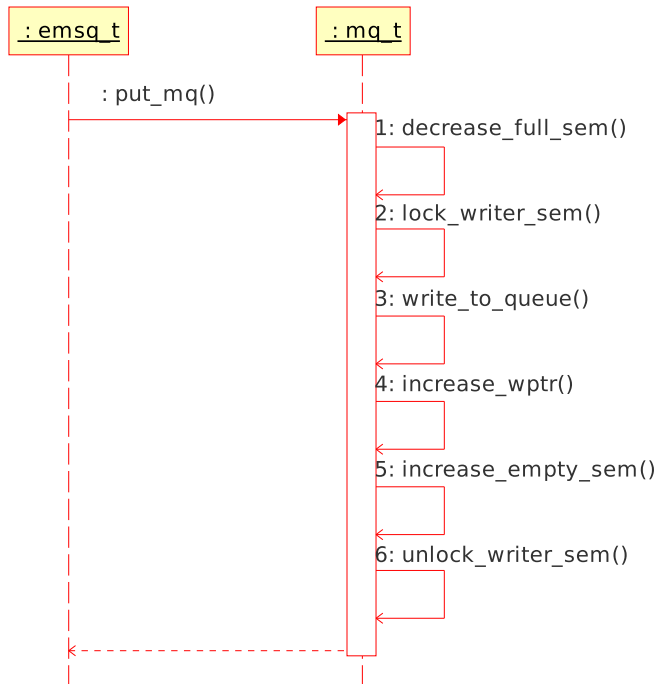
Diese Schicht repräsentiert die Datenstruktur einer Message Queue als solche, repräsentiert durch `mq_t` in der unteren Grafik.

fail_mq Test auf Gültigkeit der Queue

get_mq eine Nachricht von der Queue lesen



put_mq eine Nachricht in die Queue schreiben



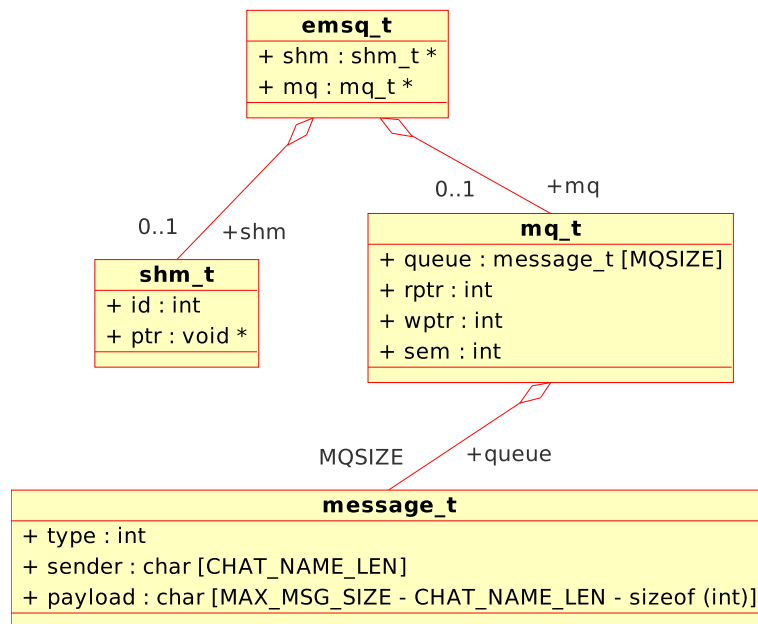
7.2.4 Queue Implementierungsdetails

Implementierungsdetails wie die verwendeten Semaphoren und Shared Memory Segmente.

7.2.5 Nachricht

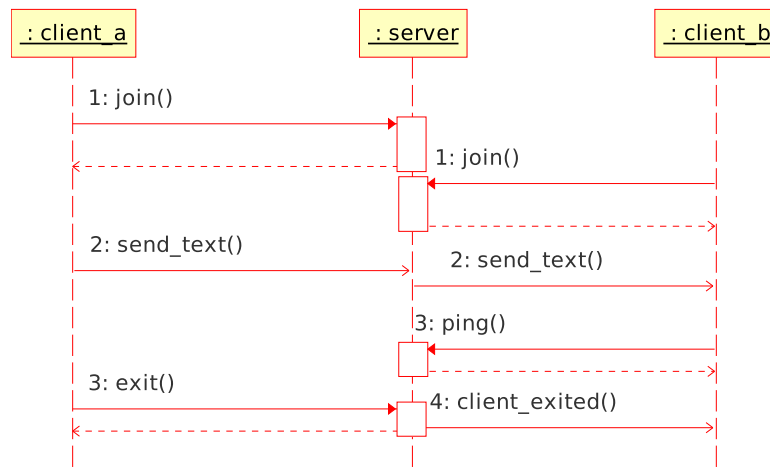
Diese Schicht ist die eigentliche Nachricht, ohne Message Queue spezifische Parameter, repräsentiert durch die Datenstruktur `message_t`.

7.3 Emulierte Message Queue Datenstrukturen



8 Kommunikationsprotokoll

8.1 Client-Serverkommunikationssequenz



client_a Clientnachrichten

1. join (): der Client tritt dem Chat bei, der Server wird alle anwesenden Clients informieren.
2. send_text (): eine Text Nachricht wird verschickt, der Server wird diese an alle anwesenden Clients weiterleiten. Es gibt keine Antwort.
3. exit (): der Client hat vor die Verbindung zu beenden und schickt ein exit, alle Clients werden davon informiert. Es gibt keine Antwort. Der Server wird darauf hin die Verwaltungsdaten des verlassenden Clients sowie dessen IPC Verbindung schliessen.

client_b Clientnachrichten

1. join (): der Client tritt dem Chat bei
2. send_text (): der Client erhält eine Text Nachricht vom Client_a, diese wurde vom Server weitergeleitet.
3. ping (): der Client sendet einen ping an den Server, der mit einer gleichartigen Antwort und den Daten die der Client mitgeschickt hat wieder zurückschicken.

8 *Kommunikationsprotokoll*

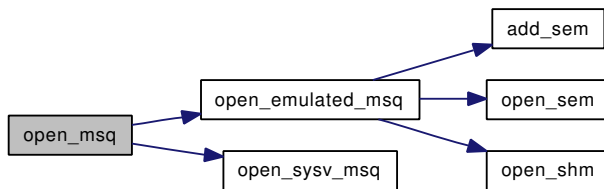
4. `client_exited ()`: der Client erhält die Nachricht, dass ein anderer Client soeben den Chat verlassen hat.

server Der Server wird nur Nachrichten weiterleiten und beantworten.

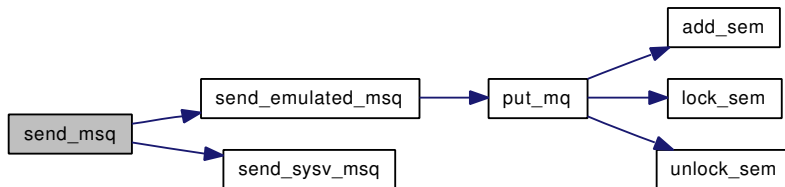
9 Aufrufgraphen

9.1 Messagequeue Toplevel Functions

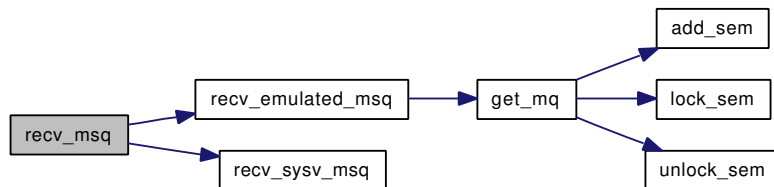
9.1.1 open_msq ()



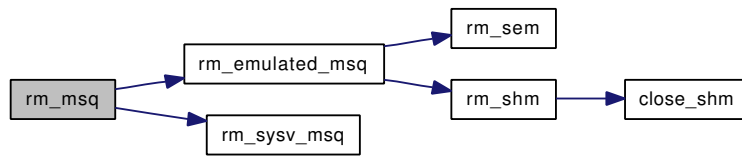
9.1.2 send_msq ()



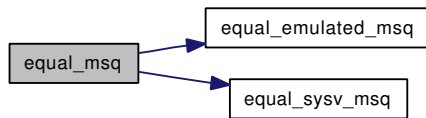
9.1.3 recv_msq ()



9.1.4 rm_msq ()

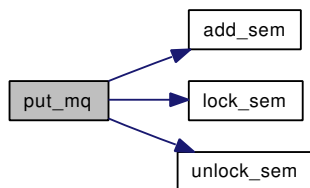


9.1.5 equal_msq ()

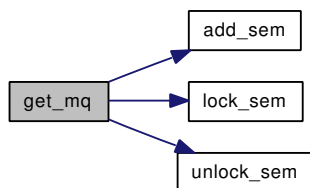


9.2 Low Level Message Queue Data Structure

9.2.1 put_mq ()

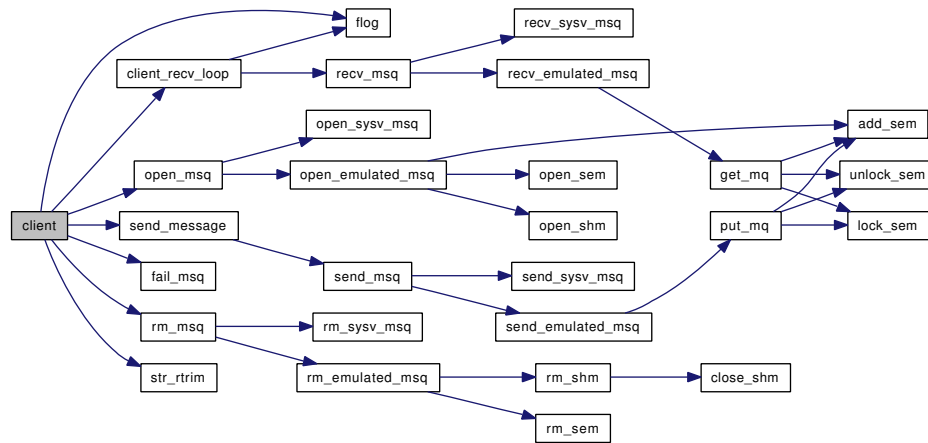


9.2.2 get_mq ()

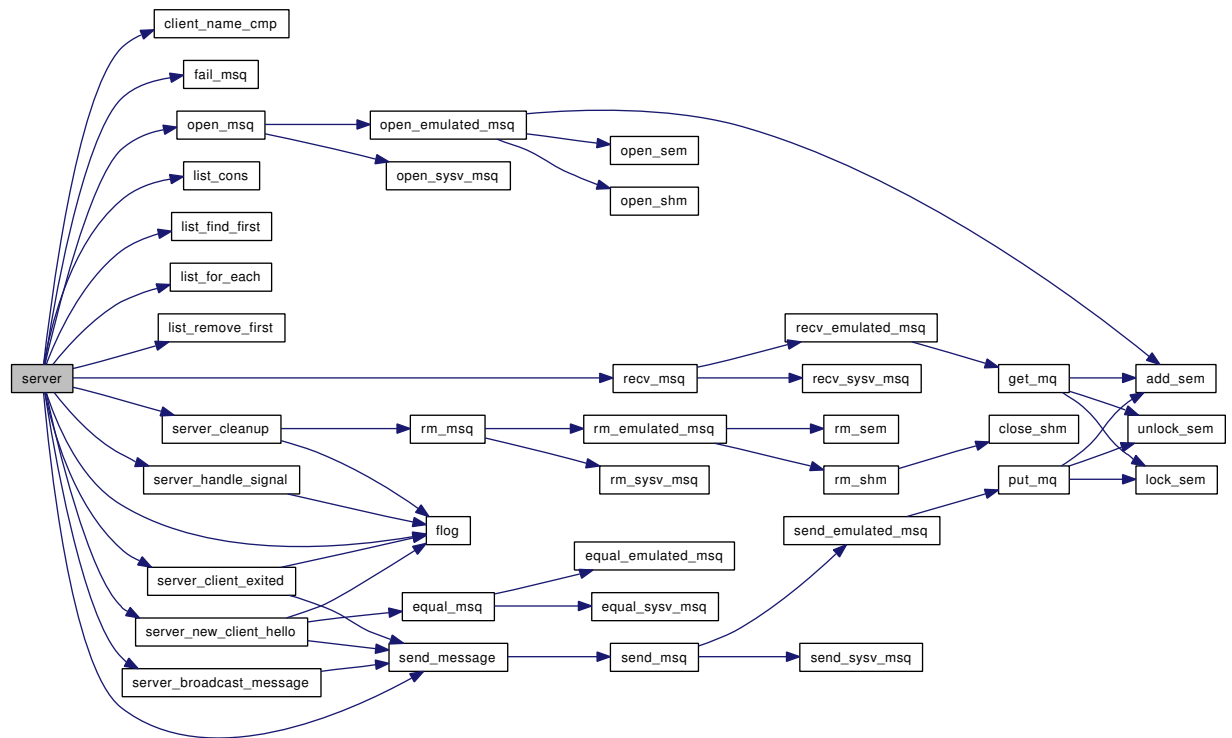


9.3 Client And Server Functions

9.3.1 client ()



9.3.2 server ()



9.4 Programmbezug

9.4.1 Programmaufruf

Das Programm findet sich nach der Kompilierung im Unterverzeichnis src/ des Projektverzeichnis. Es kann wie folgt aufgerufen werden:

```
Usage: ./src/chat [-a API] [-p PATHSPEC] [-h] -s|-c NICK
-a API      API is sysv or emu
            sysv   for System V native message queues
            posix  for POSIX native message queues
            shm    for System V message queue through shm and semaphores
-s          act as server
-c NICK     act as client with nick NICK
-h          this help message
```

9 Aufrufgraphen

Es muss entweder die option `-s` oder `-c $NICKNAME` angegeben werden um Server- oder Clientfunktionalität auszuwählen. Des weiteren muss bei der Ausführung auch die zu benutzende Message Queue Methode mit der option `-a $NAME` gewählt werden.

Bevor clients benutzt werden können muss natürlich ein Server-Prozess gestartet werden. Clients verbinden automatisch auf einen bestehenden Server.

Ein Client prozess wird nach erfolgreichem Starten auf eingabe warten, jede eingabe wird an den Server weitergeleitet, welche die Nachricht an alle anderen angemeldeten Clients weiterreichen wird.

Zusätzlich steht ein kommando `/ping` zur Verfügung mit dem ein ping an den Server gesendet wird, den dieser umgehend beantwortet. Ausgabe des Pings ist die Länge der Round-Trip-Time zum Server.

Beenden werden die clients auf EOF (meist `^D`). Der Server kann per SIGINT *sauber* beendet werden.

10 Source Code

10.1 Makefiles

10.1.1 chat/include/Makefile

```
1 all:
  @echo "Hey, nothin to see here, do this again and see what happens!"
3  @false
```

10.1.2 chat/Makefile

```
1 # configure modules
  USE_SYSV_MSQ = YesPlease
3 USE_SHM_MSQ = YesPlease
  USE_POSIX_MSQ = YesPlease
5
  export USE_SYSV_MSQ USE_SHM_MSQ USE_POSIX_MSQ
7
  # want to know where we are...
9 BASE = $(shell pwd)
  export BASE
11
  include rules.mk
13
  # dead simple, clean if cfg has changed...
15 all:
  @touch .config
17 @echo "sv$(USE_SYSV_MSQ)px$(USE_POSIX_MSQ)sm$(USE_SHM_MSQ)" > /tmp/dacfg
  @cmp /tmp/dacfg .config >/dev/null 2>&1 || $(MAKE) clean
19 @mv /tmp/dacfg .config
  $(MAKE) -C src
21
  docs:
23  doxygen
25
  clean:
  $(MAKE) -C src clean
27  rm -rf doxy
29
  depend:
  $(MAKE) -C src depend
31
  git-clean:
33  git clean -d -x
```

10.1.3 chat/src/Makefile

```
1 CPPFLAGS += -I../include -I$(BASE)/include
  OBJECTS = main.o util.o client.o server.o msq.o list.o
3
```

10 Source Code

```
1 ifeq ($(USE_SYSV_MSQ),YesPlease)
5 OBJECTS += sysvmsq.o
  CPPFLAGS += -DUSE_SYSV_MSQ
7 endif

9 ifeq ($(USE_POSIX_MSQ),YesPlease)
  OBJECTS += posixmsq.o
11 CPPFLAGS += -DUSE_POSIX_MSQ
  endif

13 ifeq ($(USE_SHM_MSQ),YesPlease)
15 OBJECTS += sysvshm.o sysvsem.o mq.o emumsq.o
  CPPFLAGS += -DUSE_SHM_MSQ
17 endif

19 PROGRAM = chat

21 ifndef RULES_INCLUDED
  include ../rules.mk
23 endif

25 all: $(PROGRAM)

27 $(PROGRAM): $(OBJECTS)
  $(CC) $^ $(LDFLAGS) -o $@

29 clean:
31 -rm -f *.o $(PROGRAM) core

33 depend:
  makedepend $(CPPFLAGS) *.c
```

10.2 Header Files

10.2.1 chat/include/client.h

```
1 #ifndef CLIENT_H_
2 #define CLIENT_H_

4 #include "msq.h"

6 /** @brief client process function
  * @param ipcname server ipc path name
  * @param nick client name
  * @param type message queue API type */
10 void client (const char * ipcname, const char * nick, const msq_api_t type);

12 #endif /* !CLIENT_H_ */
```

10.2.2 chat/include/emumsq.h

```
1 #ifndef EMUMSQ_H_
2 #define EMUMSQ_H_

4 #ifndef _XOPEN_SOURCE
  #define _XOPEN_SOURCE
6 #endif

8 #ifndef _GNU_SOURCE
  #define _GNU_SOURCE
10 #endif

12 #include <unistd.h>

14 #include "message.h"

16 /** @brief open message queue
```

10 Source Code

```
18  * @param name message queue path name
19  * @param flags message queue flags and mode, see semget(2) or shmget(2)
20  * @param id message queue path name id, see ftok(2) */
21 void * open_emulated_msq (const char * name, const int flags);
22
23 /** @brief send message
24  * @param msq message queue to send message too :D
25  * @param msgbuf message to send */
26 int send_emulated_msq (void * msq, const message_t * msgbuf);
27
28 /** @brief receive a message from a queue
29  * @param msq message queue to receive from
30  * @param msgbuf message buffer, the message will be stored in */
31 ssize_t recv_emulated_msq (void * msq, message_t * msgbuf);
32
33 /** @brief remove/unlink a message queue
34  * @param msq the message queue */
35 int rm_emulated_msq (void * msq);
36
37 /** @brief equality comparison from emsq objects */
38 int equal_emulated_msq (void * a, void * b);
39
40 #endif /* !EMUMSQ_H__ */
```

10.2.3 chat/include/list.h

```
1  #ifndef LIST_H__
2  #define LIST_H__
3
4  /** @brief list item pointer type */
5  typedef struct list * list_p;
6  /** @brief list item type */
7  typedef struct list list_t;
8
9  /** @brief simple singly linked list */
10 struct list {
11     /** @brief data member of list cell */
12     void * elem;
13     /** @brief pointer to next list cell */
14     list_p next;
15 };
16
17 /** @brief determine list length
18  * @param l list */
19 size_t list_len (list_p l);
20
21 /** @brief prepend data element to list
22  * @param a list
23  * @param e element to prepend to a */
24 list_p list_cons (list_p a, void * e);
25
26 /** @brief call back supplied to list_for_each
27  * @param arg list item (data member) of current list cell
28  * @param userdata user supplyable data element for call back */
29 typedef void (* for_each_func_t) (void * arg, void * userdata);
30
31 /** @brief iterate through a list and call func for each list cell
32  * @param l list
33  * @param func callback
34  * @param userdata to supply to callback */
35 void list_for_each (list_p l, for_each_func_t func, void * data);
36
37 /** @brief list comparator function, return 0 for not equal, else !0
38  * @param arg list item
39  * @param userdata user specified element to compare with */
40 typedef int (* list_cmp_func_t) (void * arg, void * userdata);
41
42 /** @brief remove and return first element that equals with cmpfunc
43  * @param l list
44  * @param cmpfunc comparator call back
45  * @param data user data, passed to cmpfunc for comparison */
46 list_p list_remove_first (list_p * l, list_cmp_func_t cmpfunc, void * data);
47
48 /** @brief return first list cell that compares equal with cmpfunc
49  * @param l list
50  * @param cmpfunc comparator call back
51  * @param data user data passed to cmpfunc */
52 list_p list_find_first (list_p l, list_cmp_func_t cmpfunc, void * data);
```

10 Source Code

```
53 #endif /* !LIST_H__ */
```

10.2.4 chat/include/message.h

```
1 #ifndef MESSAGE_H__
2 #define MESSAGE_H__
3
4 /** @brief maximum nickname length + 1 */
5 #define CHAT_NAME_LEN 32
6 /** @brief maximum overall message size, i.e. w/ sender name */
7 #define MAX_MSG_SIZE 256
8
9 /** @brief message type for text messages */
10 #define MTYPE_EXIT 1
11 /** @brief message type for join messages */
12 #define MTYPE_JOIN 2
13 /** @brief message type for client exit messages */
14 #define MTYPE_TEXT 3
15 /** @brief message type for ping message from client->server->client */
16 #define MTYPE_PING 4
17
18 typedef struct message message_t;
19 /** @brief message passed over the message queues, fixed length to keep it
20  * simple*/
21 struct message {
22     int type;
23     char sender [CHAT_NAME_LEN];
24     char payload [MAX_MSG_SIZE - CHAT_NAME_LEN - sizeof (int)];
25 };
26 #endif /* !MESSAGE_H__ */
```

10.2.5 chat/include/mq.h

```
1 #ifndef MQ_H__
2 #define MQ_H__
3
4 #include "message.h"
5
6 #define MQSIZE 32
7
8 /** @brief mq.sem read semaphore
9  *
10  * Read from message queue exclusive lock */
11 #define RDSEM 0
12 /** @brief mq.sem write semaphore
13  *
14  * Write to message queue exclusive lock */
15 #define WRSEM 1
16 /** @brief mq.sem signalling semaphore
17  *
18  * This semaphore locks get_mq if the mq is empty, i.e. messages need to be
19  * written before anything can be read from it. */
20 #define MPTYSEM 2
21 /** @brief mq.sem signalling semaphore~-1
22  *
23  * This semaphore locks put_mq if the mq is full, i.e. messages need to be
24  * read before anything can be sent over it. */
25 #define FULLSEM 3
26
27 typedef struct mq mq_t;
28
29 /** @brief low level message queue, to be placed in an shm segment */
30 struct mq {
31     /** @brief the fixed size queue */
32     message_t queue [MQSIZE];
33     /** @brief read pointer */
34     int rptr;
35     /** @brief write pointer */
36     int wptr;
37 }
```

10 Source Code

```
37  /** @brief semaphore identifier, current implementation holds 3 semaphores */
38  int      sem;
39  };
40
41  #if 0
42  /** @brief test for mq failed, i.e. readptr == writeptr */
43  static int fail_mq (mq_t * mq)
44  {
45      return_val_if_fail (mq != NULL, 0);
46      return mq->rptr == mq->wptr;
47  }
48  #endif
49
50  /** @brief get a message from the queue
51   * @param mq message queue to read message from */
52  message_t * get_mq (mq_t * mq);
53
54  /** @brief put a message to a queue
55   * @param mq message queue to write message too
56   * @param msg message to write */
57  int put_mq (mq_t * mq, const message_t * msg);
58
59  #endif /* !MQ_H__ */
```

10.2.6 chat/include/msq.h

```
1  #ifndef MSQ_H__
2  #define MSQ_H__
3
4  #include <unistd.h>
5
6  #include "message.h"
7
8  #define OPEN_EXCLUSIVE 0x31415
9
10 /** @brief defines symbolic message queue api names. Used by ipc open
11  * calls to select native message queues or the emulated counterpart. */
12  enum msq_api {
13      MSQ_API_SYSV = 0,
14      MSQ_API_EMULATED = 1,
15      MSQ_API_POSIX = 2
16  };
17  typedef enum msq_api msq_api_t;
18
19  #if defined (USE_SYSV_MSQ) || defined (USE_SHM_MSQ)
20  /** @brief chat ID for ipc system calls */
21  #define IPC_CHAT_ID 161803399
22  #endif
23
24 /** @brief general message queue wrapper, for sysv and emulated msq */
25 typedef struct msq msq_t;
26 /** @brief general message queue wrapper, for sysv and emulated msq */
27 struct msq {
28     /** @brief message queue type, i.e. API type */
29     int type;
30     /** @brief message queue data */
31     union {
32         /** @brief system v message queue identifier */
33         int sysv;
34         /** @brief emulated message queue identifier
35          *
36          * This is void as to reserve some pretty useless generaslity */
37         void * ptr;
38     } msq;
39 };
40
41 /** @brief open a message queue object
42  * @param name message queue path name
43  * @param flags message queue flags and mode, see msgget(2), shmget(2) or semget(2)
44  * @param id message queue path name id
45  * @param type message queue API to use, one of MSQ_API_SYSV, MSQ_API_EMULATED */
46  msq_t * open_msq (const char * name, const int flags, const msq_api_t type);
47
48 /** @brief general message sending function
49  * @param msq message queue
50  * @param msg message */
51  int send_msq (const msq_t * msq, const message_t * msg);
```

10 Source Code

```
53 /** @brief general message receiving function
   * @param msq message queue
55 * @param msgbuf the message buffer */
   ssize_t recv_msq (const msq_t * msq, message_t * msgbuf);
57
   /** @brief general message queue removing function
59 * @param msq message queue */
   int rm_msq (const msq_t * msq);
61
   /** @brief general message queue equality comparison */
63 int equal_msq (const msq_t * a, const msq_t * b);
65
   /** @brief general message queue failure check */
67 int fail_msq (const msq_t * msq);
   #endif /* !MSQ_H_ */
```

10.2.7 chat/include/posixmsq.h

```
1 #ifndef POSIXMSQ_H_
   #define POSIXMSQ_H_
2
   #include <mqueue.h>
4
   #include "message.h"
6
   /** @brief open a posix message queue
   * @param name message queue path name
   * @param flags message queue flags and mode, see msgget(2)
   * @param id message queue path name id, see ftok(2) */
12 void * open_posix_msq (const char * name, const int flags);
14
   /** @brief send a message over a posix message queue
   * @param mq message queue identifier
   * @param msgbuf message buffer */
16 int send_posix_msq (void * mq, const message_t * msgbuf);
18
   /** @brief receive a message from a posix message queue
   * @param mq message queue identifier
   * @param msgbuf message buffer, where the received message shall be stored */
22 ssize_t recv_posix_msq (void * mq, message_t * msgbuf);
24
   /** @brief remove a posix message queue identifier from the system
   * @param mq message queue identifier */
26 int rm_posix_msq (void * mq);
28
   /** @brief equality comparison for message queue identifiers */
   int equal_posix_msq (void * a, void * b);
30
   #endif /* !POSIXMSQ_H_ */
```

10.2.8 chat/include/server.h

```
1 #ifndef SERVER_H_
   #define SERVER_H_
3
   #include "msq.h"
5
   /** @brief server process main function
   * @param ipcname server ipc path name
   * @param type ipc API to use */
9 void server (const char * ipcname, const msq_api_t type);
11
   #endif /* !SERVER_H_ */
```

10.2.9 chat/include/sysvmsq.h

10 Source Code

```
1 #ifndef SYSVMSQ_H__
2 #define SYSVMSQ_H__
3
4 #ifndef _XOPEN_SOURCE
5 #define _XOPEN_SOURCE 600
6 #endif
7
8 #ifndef _GNU_SOURCE
9 #define _GNU_SOURCE
10 #endif
11
12 #include <sys/ipc.h>
13 #include <sys/shm.h>
14 #include <sys/sem.h>
15 #include <sys/types.h>
16
17 #include "msq.h"
18
19 /** @brief sysv message type identifier, not used, . so constant */
20 #define SYSV_MSG_TYPE 11
21
22 /** @brief open a system v message queue
23  * @param name message queue path name
24  * @param flags message queue flags and mode, see msgget(2)
25  * @param id message queue path name id, see ftok(2) */
26 int open_sysv_msq (const char * name, const int flags);
27
28 /** @brief send a message over a system v message queue
29  * @param msgid message queue identifier
30  * @param msgbuf message buffer */
31 int send_sysv_msq (int msgid, const message_t * msgbuf);
32
33 /** @brief receive a message from a system v message queue
34  * @param msgid message queue identifier
35  * @param msgbuf message buffer, where the received message shall be stored */
36 ssize_t recv_sysv_msq (int msgid, message_t * msgbuf);
37
38 /** @brief remove a system v message queue identifier from the system
39  * @param msgid message queue identifier */
40 int rm_sysv_msq (int msgid);
41
42 /** @brief equality comparison for message queue identifiers */
43 int equal_sysv_msq (int a, int b);
44
45 #endif /* !SYSVMSQ_H__ */
```

10.2.10 chat/include/sysvsem.h

```
1 #ifndef SYSVSEM_H__
2 #define SYSVSEM_H__
3
4 #ifndef _XOPEN_SOURCE
5 #define _XOPEN_SOURCE
6 #endif
7
8 #ifndef _GNU_SOURCE
9 #define _GNU_SOURCE
10 #endif
11
12 #include <sys/ipc.h>
13 #include <sys/types.h>
14 #include <sys/sem.h>
15
16 /** @brief open a system v message queue
17  * @param name message queue path name
18  * @param flags message queue flags and mode, see semget(2)
19  * @param id message queue path id, see ftok(2)
20  * @param nsems number of semaphores to open
21  *
22  * Opens a system v message queue, propagating system errors. */
23 int open_sem (const char * name, const int flags, const int nsems);
24
25 /** @brief lock a semaphore
26  * @param semid semaphore array identifier
27  * @param semnum semaphore index
28  *
29  * Locks a semaphore, but waits for 0 before locking (i.e. emulating
```

10 Source Code

```
31  * a degraded semaphore (mutex) */
int lock_sem (const int semid, const int semnum);

33  /** @brief unlock a semaphore
  * @param semid semaphore array identifier to unlock
35  * @param semnum semaphore index */
int unlock_sem (const int semid, const int semnum);

37
/** @brief lock w/o wait\
  * @param semid semaphore array identifier
  * @param semnum semaphore index
41  *
  * This function increments the resource count of the specified
43  * semaphore by 1, see sub_sem for decrement. */
int add_sem (const int semid, const int semnum);

45
/** @brief alias for unlock
  * @param semid semaphore array identifier
  * @param semnum semaphore index */
47  extern int (* sub_sem) (const int semid, const int semnum);

51  /** @brief remove a semaphore
  * @param semid semaphore array identifier to remove
53  * @param semnum semaphore index */
int rm_sem (const int semid, const int semnum);

55
#endif /* !SYSVSEM_H_ */
```

10.2.11 chat/include/sysvshm.h

```
#ifndef SYSVSHM_H_
2 #define SYSVSHM_H_

4 #ifndef _XOPEN_SOURCE
#define _XOPEN_SOURCE
6 #endif

8 #ifndef _GNU_SOURCE
#define _GNU_SOURCE
10 #endif

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

14
/** @brief shared memory wrapper structure */
16 typedef struct {
  /** @brief shared memory identifier */
18  int id;
  /** @brief pointer to the attached shared memory.
  *
  * Needed to cleanly unmap the shared memory segment as
22  * semat does not yield same results for different calls
  * in the same process. */
24  void * ptr;
} shm_t;

26
/** @brief open a shared memory segment
  * @param name shm path name to open
  * @param size size of the shm o request
  * @param flags shared memory flags and mode
  * @param id shm pathname id
32  *
  * See shmget(2) for how flags handles shm permissions and shmflags */
34 shm_t * open_shm (const char * name, const size_t size, const int flags);

36 /** @brief close a shared memory segment, i.e. detach/unmap it
  * @param shm shared memory to detach */
38 int close_shm (const shm_t * shm);

40 /** @brief remove/unlink an shm id
  *
  * unmap a shared memory segment then remove the shared memory segment */
42 int rm_shm (const shm_t * shm);

44
#endif /* !SYSVSHM_H_ */
```

10.2.12 chat/include/util.h

```

1 #ifndef UTIL_H__
2 #define UTIL_H__
3
4 #include "msq.h"
5
6 #include <stdlib.h>
7 #include <stdio.h>
8
9 /** @brief allocate N*sizeof(T) of zeroed memory
10  * @param T ttype to allocate and cast too
11  * @param N number of elements of sizeof (T) to allocate */
12 #define new0(T,N) ((T*) calloc (N, sizeof (T)))
13
14 /** @brief a nice assertion macro to return some error value */
15 #define return_val_if_fail(EXPR,RETVAL) do { \
16     if (!(EXPR)) { \
17         fprintf (stderr, "ASSERTION FAILURE: %s:%d: '" # EXPR, __FILE__, __LINE__); \
18         fprintf (stderr, "'\n"); \
19         return (RETVAL); \
20     } \
21 } while (0)
22
23 /** @brief a nice assertion macro that just returns */
24 #define return_if_fail(EXPR) do { \
25     if (!(EXPR)) { \
26         fprintf (stderr, "ASSERTION FAILURE: %s:%d: '" # EXPR, __FILE__, __LINE__); \
27         fprintf (stderr, "'\n"); \
28         return; \
29     } \
30 } while (0)
31
32 /** @brief server logging function, prints message with time info */
33 void flog (FILE * f, char * fmt, ...);
34
35 /** @brief return a newly allocated string representing the users home directory */
36 char * get_user_home (void);
37
38 /** @brief right trim character chr from str
39  * @param str string to trim
40  * @param chr char to trim from str */
41 char * str_rtrim (char * str, char chr);
42
43 /** @brief send a message
44  * @param msq message queue
45  * @param from sender id, i.e. chat nick name
46  * @param msg text message to send
47  * @param type message type */
48 int send_message (const msq_t * msq, const char * from, const char * msg, const size_t size, const int type);
49
50 #endif /* !UTIL_H__ */

```

10.3 Implementations

10.3.1 chat/src/client.c

```

1 #define _GNU_SOURCE
2
3 #include <sys/time.h>
4
5 #include <pthread.h>
6 #include <time.h>
7 #include <string.h>
8 #include <errno.h>
9 #include <fcntl.h>
10 #include <signal.h>
11 #include <assert.h>
12
13 #include "util.h"
14 #include "client.h"

```

10 Source Code

```
16 /** @brief client receiver send loop
17  * @param args message queue to listen on */
18 static void * client_recv_loop (void * args)
19 {
20     msq_t * msq = (msq_t *)args;
21     message_t msg;
22     ssize_t n;
23
24     while ((n = recv_msq (msq, &msg)) != -1) {
25         switch (msg.type) {
26             case MTYPE_TEXT:
27                 flog (stdout, "%s: %s\n", msg.sender, msg.payload);
28                 break;
29             case MTYPE_JOIN:
30                 flog (stdout, "Client '%s' joined\n", msg.sender);
31                 break;
32             case MTYPE_EXIT:
33                 flog (stdout, "Client '%s' exited\n", msg.sender);
34                 break;
35             case MTYPE_PING:
36                 {
37                     double t0, t1;
38                     struct timeval tv0, tv1;
39                     gettimeofday (&tv1, NULL);
40                     memcpy (&tv0, msg.payload, sizeof (struct timeval));
41                     t0 = tv0.tv_sec * 1e6 + tv0.tv_usec;
42                     t1 = tv1.tv_sec * 1e6 + tv1.tv_usec;
43                     flog (stdout, "RTT to Server %f us\n", t1 - t0);
44                 }
45                 break;
46         }
47     }
48
49     if (n == -1)
50         flog (stderr, "ERROR: Could not receive message: %s\n", strerror (errno));
51
52     return NULL;
53 }
54
55 static int have_sysv () {
56     #if defined(USE_SYSV_MSQ) || defined(USE_SHM_MSQ)
57         return 1;
58     #endif
59     return 0;
60 }
61
62 /** @brief client process function
63  * @param ipcname server ipc path name
64  * @param nick client name
65  * @param type message queue API type */
66 void client (const char * ipcname, const char * nick, const msq_api_t type)
67 {
68     msq_t * rmq = NULL, * smq = NULL;
69     char * client_ipcname = 0;
70     pthread_t pth;
71
72     if (have_sysv () && (type == MSQ_API_SYSV || type == MSQ_API_EMULATED)) {
73         client_ipcname = tempnam ("/tmp", "ipc");
74         do {
75             int fd = open (client_ipcname, O_CREAT|O_EXCL|O_RDWR, 0600);
76             if (fd == -1) {
77                 flog (stderr, "ERROR: Could not create tmp file: %s\n", strerror (errno));
78                 return;
79             }
80             close (fd);
81         } while (0);
82     } else {
83         client_ipcname = tempnam ("ipc", "");
84         char * x;
85         while ((x = strchr (client_ipcname+1, '/'))
86             *x = '_');
87     }
88
89     rmq = open_msq (client_ipcname, OPEN_EXCLUSIVE, type);
90     if (fail_msq (rmq)) {
91         flog (stderr, "ERROR: Could not create message queue: %s\n", strerror (errno));
92         goto unlink_return;
93     }
94
95     signal (SIGINT, SIG_IGN);
96
97     smq = open_msq (ipcname, 0, type);
98     if (fail_msq (smq)) {
```

10 Source Code

```
100     flog (stderr, "ERROR: Could not open message queue: %s\n", strerror (errno));
101     goto unlink_return;
102 }
103
104 pthread_create (&pth, NULL, client_rcv_loop, rmq);
105
106 if (send_message (smq, nick, client_ipcname, strlen (client_ipcname)+1, MTYPE_JOIN) == -1) {
107     flog (stderr, "ERROR: Could not send JOIN to server: %s\n", strerror (errno));
108     goto unlink_return;
109 }
110
111 do {
112     char * lineptr = NULL;
113     size_t n;
114
115     while (getline (&lineptr, &n, stdin) > 0) {
116         usleep (1000);
117         str_rtrim (lineptr, '\n');
118         if (strstr (lineptr, "/ping") == lineptr) {
119             struct timeval tv;
120             gettimeofday (&tv, NULL);
121             send_message (smq, nick, (char *) &tv, sizeof (tv), MTYPE_PING);
122         } else
123             send_message (smq, nick, lineptr, strlen (lineptr), MTYPE_TEXT);
124         free (lineptr);
125         lineptr = 0;
126         n = 0;
127     }
128 } while (0);
129
130 flog (stdout, "Ending here...\n");
131
132 usleep (100000);
133
134 if (send_message (smq, nick, "", 1, MTYPE_EXIT) == -1) {
135     flog (stderr, "ERROR: Could not send EXIT to server: %s\n", strerror (errno));
136 }
137
138 unlink_return:
139     rm_msq (rmq);
140     free (rmq);
141     rmq=0;
142     unlink (client_ipcname);
143     free (smq);
144 }
```

10.3.2 chat/src/emumsq.c

```
1 #define _XOPEN_SOURCE
2 #define _GNU_SOURCE
3
4 #include <string.h>
5 #include <assert.h>
6 #include <strings.h>
7
8 #include "util.h"
9 #include "mq.h"
10 #include "emumsq.h"
11 #include "sysvshm.h"
12 #include "sysvsem.h"
13
14 /* {{{1 emulated message queue wrappers */
15 typedef struct emsq emsq_t;
16
17 /** @brief emulated message queue wrapper structure */
18 struct emsq {
19     /** @brief shared memory wrapper structure */
20     shm_t * shm;
21     /** @brief low level message queue, data structure */
22     mq_t * mq;
23 };
24
25 #define EMSQ(X) ((emsq_t *) (X))
26
27 /** @brief open message queue
28  * @param name message queue path name
29  * @param flags message queue flags and mode, see semget(2) or shmget(2)
30  * @param id message queue path name id, see ftok(2) */
```

10 Source Code

```
void * open_emulated_msq (const char * name, const int flags)
32 {
    int i;
34     emsq_t * msq = new0 (emsq_t, 1);
    msq->shm = open_shm (name, sizeof (mq_t), flags);
36     if (! msq->shm)
        goto cleanup;
38     msq->mq = msq->shm->ptr;
    bzero (msq->mq, sizeof (mq_t));
40     if ((msq->mq->sem = open_sem (name, flags, 4)) == -1)
        goto cleanup;
42     for (i=0; i<MQSIZE; ++i)
        add_sem (msq->mq->sem, FULLSEM);
44     return msq;
cleanup:
46     free (msq);
    return NULL;
48 }

50 /** @brief send message
    * @param msq message queue to send message too :D
    * @param msgbuf message to send */
52 int send_emulated_msq (void * msq, const message_t * msgbuf)
54 {
    return_val_if_fail (msq != NULL, -1);
56     return put_mq (EMSQ(msq)->mq, msgbuf);
}

58 /** @brief receive a message from a queue
    * @param msq message queue to receive from
    * @param msgbuf message buffer, the message will be stored in */
62 ssize_t recv_emulated_msq (void * msq, message_t * msgbuf)
{
64     message_t * m;
    return_val_if_fail (msq != NULL, -1);
66     return_val_if_fail (msgbuf != NULL, -1);
    m = get_mq (EMSQ(msq)->mq);
68     if (m == NULL)
        return -1;
70     memcpy (msgbuf, m, sizeof (message_t));
    free (m);
72     return 0;
}

74 /** @brief remove/unlink a message queue
    * @param msq the message queue */
76 int rm_emulated_msq (void * msq)
78 {
    emsq_t * m = EMSQ (msq);
80     int ret = rm_sem (m->mq->sem, RDSEM) | \
        rm_sem (m->mq->sem, WRSEM) | \
82         rm_sem (m->mq->sem, MPTYSEM) | \
        rm_sem (m->mq->sem, FULLSEM) | \
84         rm_shm (m->shm);
    bzero (msq, sizeof (emsq_t));
86     return ret;
}

88 /** @brief equality comparison from emsq objects */
90 int equal_emulated_msq (void * a, void * b)
{
92     return_val_if_fail (a != NULL, 0);
    return_val_if_fail (b != NULL, 0);
94     return EMSQ(a)->mq == EMSQ(b)->mq;
}
```

10.3.3 chat/src/list.c

```
#include "util.h"
2 #include "list.h"

4 /** @brief determine list length
    * @param l list */
6 size_t list_len (list_p l)
{
8     size_t n = 0;
    for (; l; l = l->next) ++n;
10     return n;
}
```

10 Source Code

```
12 }
13
14 /** @brief prepend data element to list
15 * @param a list
16 * @param e element to prepend to a */
17 list_p list_cons (list_p a, void * e)
18 {
19     list_p l = new0 (list_t, 1);
20     l->next = a;
21     l->elem = e;
22     return l;
23 }
24
25 /** @brief iterate through a list and call func for each list cell
26 * @param l list
27 * @param func callback
28 * @param data userdata to supply to callback */
29 void list_for_each (list_p l, for_each_func_t func, void * data)
30 {
31     for (; l; l = l->next)
32         func (l->elem, data);
33 }
34
35 /** @brief remove and return first element that equals with cmpfunc
36 * @param l list
37 * @param cmpfunc comparator call back
38 * @param data user data, passed to cmpfunc for comparison */
39 list_p list_remove_first (list_p * l, list_cmp_func_t cmpfunc, void * data)
40 {
41     list_p i = 0, j = 0, f = 0;
42
43     return_val_if_fail (l != NULL, NULL);
44
45     for (i = *l; i; i = i->next)
46         if (cmpfunc (i->elem, data)) {
47             f = i;
48             break;
49         }
50     else
51         j = i;
52
53     if (f) {
54         if (j)
55             j->next = f->next;
56         else
57             *l = f->next;
58     }
59
60     return f;
61 }
62
63 /** @brief return first list cell that compares equal with cmpfunc
64 * @param l list
65 * @param cmpfunc comparator call back
66 * @param data user data passed to cmpfunc */
67 list_p list_find_first (list_p l, list_cmp_func_t cmpfunc, void * data)
68 {
69     for (; l; l = l->next)
70         if (cmpfunc (l->elem, data))
71             return l;
72     return NULL;
73 }
```

10.3.4 chat/src/main.c

```
1 #define _GNU_SOURCE
2
3 #include <getopt.h>
4 #include <string.h>
5
6 #include "util.h"
7 #include "client.h"
8 #include "server.h"
9
10 #if !defined(USE_SYSV_MSQ) && !defined(USE_SHM_MSQ) && !defined(USE_POSIX_MSQ)
11 #error "You have to define at least one of USE_SYSV_MSQ, USE_SHM_MSQ or USE_POSIX_MSQ!"
12 #endif
```

10 Source Code

```
14 /** @brief report program usage and exit with error code */
void usage (char ** v)
16 {
    fprintf (stderr, "Usage: %s [-a API] [-p PATHSPEC] [-h] -s|-c NICK\n", *v);
18     puts ("\t-a API      API is sysv or emu");
    #ifdef USE_SYSV_MSQ
20     puts ("          sysv    for System V native message queues");
    #endif
22 #ifdef USE_POSIX_MSQ
    puts ("          posix   for POSIX native message queues");
24 #endif
    #ifdef USE_SHM_MSQ
26     puts ("          shm     for System V message queue through shm and semaphores");
    #endif
28     puts ("\t-s          act as server");
    puts ("\t-c NICK     act a scilient with nick NICK");
30     puts ("\t-h          this help message");
    puts ("");
32     exit (1);
}
34
/* {{{1 main () { return *(int*)0=0; } */
36 int main (const int c, char ** v)
{
38     /** @brief defaulting to users $HOME for ipc path name */
    const char * pathspec = get_user_home ();
40     /** @brief nick can be { 0: server, -1: invalid, default: client } */
    const char * nick = (char *) -1;
42     /** @brief defaulting to emulated message queues */
    int api = -1;
44
    #ifdef USE_SHM_MSQ
46     api = MSQ_API_EMULATED;
    #endif
48 #ifdef USE_SYSV_MSQ
    api = MSQ_API_SYSV;
50 #endif
    #ifdef USE_POSIX_MSQ
52     api = MSQ_API_POSIX;
    #endif
54
    while (1) {
56         int opt = getopt (c, v, "a:sc:h");
58
        if (opt == -1)
            break;
60
        switch (opt) {
62             case 's':
                nick = 0;
                break;
64             case 'c':
                nick = optarg;
                break;
66             case 'a':
                if (!0)
70                 ;
            #ifdef USE_SYSV_MSQ
72             else if (!strcasecmp (optarg, "SystemV") || !strcasecmp (optarg, "sysv"))
                api = MSQ_API_SYSV;
74 #endif
            #ifdef USE_SHM_MSQ
76             else if (!strcasecmp (optarg, "emulated") || !strcasecmp (optarg, "emu") || !strcasecmp (optarg, "shm"))
                api = MSQ_API_EMULATED;
78 #endif
            #ifdef USE_POSIX_MSQ
80             else if (!strcasecmp (optarg, "posix") || !strcasecmp (optarg, "psx"))
                api = MSQ_API_POSIX;
82 #endif
                break;
84             case 'h':
                default:
86                 usage (v);
            }
88         }
90
        if (nick == (char *) -1)
            usage (v);
92
        if (api == -1)
            usage (v);
94
96 #ifdef USE_POSIX_MSQ
```

10 Source Code

```
    if (api == MSQ_API_POSIX) {
98      char * x;
        while ((x = strchr (pathspec+1, '/'))
100          *x = '_');
    }
102 #endif

104 if (nick) client (pathspec, nick, api);
    else server (pathspec, api);
106
    return 0;
108 }
```

10.3.5 chat/src/mq.c

```
#define _XOPEN_SOURCE
2 #define _GNU_SOURCE

4 #include <string.h>
  #include <assert.h>
6 #include <strings.h>

8 #include "sysvsem.h"
  #include "util.h"
10 #include "mq.h"

12 /** @brief get a message from the queue
  * @param mq message queue to read message from */
14 message_t * get_mq (mq_t * mq)
  {
16   message_t * m = NULL;
     return_val_if_fail (mq != NULL, NULL);
18   return_val_if_fail (sub_sem (mq->sem, MPTYSEM) != -1, NULL);
     return_val_if_fail (lock_sem (mq->sem, RDSEM) != -1, NULL);
20   m = new0 (message_t, 1);
     memcpy (m, &mq->queue [mq->rptr], sizeof (*m));
22   memset (&mq->queue [mq->rptr], 0, sizeof (*m));
     ++mq->rptr;
24   mq->rptr %= MQSIZE;
     assert (add_sem (mq->sem, FULLSEM) != -1);
26   assert (unlock_sem (mq->sem, RDSEM) != -1);
     return m;
28 }

30 /** @brief put a message to a queue
  * @param mq message queue to write message too
  * @param msg message to write */
32 int put_mq (mq_t * mq, const message_t * msg)
34 {
     return_val_if_fail (mq != NULL, -1);
36   return_val_if_fail (msg != NULL, -1);
     return_val_if_fail (sub_sem (mq->sem, FULLSEM) != -1, -1);
38   return_val_if_fail (lock_sem (mq->sem, WRSEM) != -1, -1);
     memcpy (&mq->queue [mq->wptr], msg, sizeof (*msg));
40   ++mq->wptr;
     mq->wptr %= MQSIZE;
42   assert (add_sem (mq->sem, MPTYSEM) != -1);
     assert (unlock_sem (mq->sem, WRSEM) != -1);
44   return 0;
}
```

10.3.6 chat/src/msq.c

```
1 #define _XOPEN_SOURCE
3 #include <assert.h>
5 #include "util.h"
  #include "msq.h"
7
  #ifndef USE_SYSV_MSQ
```

10 Source Code

```
9 #include "sysvmsq.h"
#endif
11
12 #ifdef USE_SHM_MSQ
13 #include "emumsq.h"
#endif
15
16 #ifdef USE_POSIX_MSQ
17 #include "posixmsq.h"
#endif
19
20 /** @brief open a message queue object
21  * @param name message queue path name
22  * @param flags message queue flags and mode, see msgget(2), shmget(2) or semget(2)
23  * @param id message queue path name id
24  * @param type message queue API to use, one of MSQ_API_SYSV, MSQ_API_EMULATED */
25 msq_t * open_msq (const char * name, const int flags, const msq_api_t type)
26 {
27     msq_t * msq = new0 (msq_t, 1);
29     switch (type) {
30 #ifdef USE_SYSV_MSQ
31     case MSQ_API_SYSV:
32         msq->msq.sysv = open_sysv_msq (name, flags);
33         msq->type = msq->msq.sysv == -1 ? -1 : type;
34         break;
35 #endif
37 #ifdef USE_SHM_MSQ
38     case MSQ_API_EMULATED:
39         msq->msq.ptr = open_emulated_msq (name, flags);
40         msq->type = msq->msq.ptr == 0 ? -1 : type;
41         break;
42 #endif
43 #ifdef USE_POSIX_MSQ
44     case MSQ_API_POSIX:
45         msq->msq.ptr = open_posix_msq (name, flags);
46         msq->type = msq->msq.ptr == 0 ? -1 : type;
47         break;
48 #endif
51     default:
52         assert (! "Unknown API type");
53     }
55     return msq;
56 }
57
58 /** @brief general message sending function
59  * @param msq message queue
60  * @param msg message */
61 int send_msq (const msq_t * msq, const message_t * msg)
62 {
63     return_val_if_fail (msq != NULL, -1);
64     return_val_if_fail (msg != NULL, -1);
65
66     switch (msq->type) {
67 #ifdef USE_SYSV_MSQ
68     case MSQ_API_SYSV:
69         return send_sysv_msq (msq->msq.sysv, msg);
70 #endif
71 #ifdef USE_SHM_MSQ
72     case MSQ_API_EMULATED:
73         return send_emulated_msq (msq->msq.ptr, msg);
74 #endif
75 #ifdef USE_POSIX_MSQ
76     case MSQ_API_POSIX:
77         return send_posix_msq (msq->msq.ptr, msg);
78 #endif
81     default:
82         assert (! "Unknown API type");
83     }
85     return -1; /* notreached */
86 }
87
88 /** @brief general message receiving function
89  * @param msq message queue
90  * @param msgbuf the message buffer */
91 ssize_t recv_msq (const msq_t * msq, message_t * msgbuf)
```

10 Source Code

```
93 {
94     return_val_if_fail (msq != NULL, -1);
95     return_val_if_fail (msgbuf != NULL, -1);
96
97     switch (msq->type) {
98     #ifdef USE_SYSV_MSQ
99         case MSQ_API_SYSV:
100             return recv_sysv_msq (msq->msq.sysv, msgbuf);
101     #endif
102
103     #ifdef USE_SHM_MSQ
104         case MSQ_API_EMULATED:
105             return recv_emulated_msq (msq->msq.ptr, msgbuf);
106     #endif
107
108     #ifdef USE_POSIX_MSQ
109         case MSQ_API_POSIX:
110             return recv_posix_msq (msq->msq.ptr, msgbuf);
111     #endif
112
113     default:
114         assert (! "Unknown API type");
115     }
116
117     return -1; /* notreached */
118 }
119
120 /** @brief general message queue removing function
121  * @param msq message queue */
122 int rm_msq (const msq_t * msq)
123 {
124     return_val_if_fail (msq != NULL, -1);
125
126     switch (msq->type) {
127     #ifdef USE_SYSV_MSQ
128         case MSQ_API_SYSV:
129             return rm_sysv_msq (msq->msq.sysv);
130     #endif
131
132     #ifdef USE_SHM_MSQ
133         case MSQ_API_EMULATED:
134             return rm_emulated_msq (msq->msq.ptr);
135     #endif
136
137     #ifdef USE_POSIX_MSQ
138         case MSQ_API_POSIX:
139             return rm_posix_msq (msq->msq.ptr);
140     #endif
141
142     default:
143         assert (! "Unknown API type");
144     }
145
146     return -1; /* notreached */
147 }
148
149 /** @brief general message queue equality comparison */
150 int equal_msq (const msq_t * a, const msq_t * b)
151 {
152     return_val_if_fail (a != NULL || b != NULL, 0);
153
154     if (a->type != b->type)
155         return 0;
156
157     switch (a->type) {
158     #ifdef USE_SYSV_MSQ
159         case MSQ_API_SYSV:
160             return equal_sysv_msq (a->msq.sysv, b->msq.sysv);
161     #endif
162
163     #ifdef USE_SHM_MSQ
164         case MSQ_API_EMULATED:
165             return equal_emulated_msq (a->msq.ptr, b->msq.ptr);
166     #endif
167
168     #ifdef USE_POSIX_MSQ
169         case MSQ_API_POSIX:
170             return equal_posix_msq (a->msq.ptr, b->msq.ptr);
171     #endif
172
173     default:
174         assert (! "Unknown API type");
175     }
176
177     return 0; /* notreached */
178 }
```

10 Source Code

```
177 }
179 /** @brief general message queue failure check */
181 int fail_msq (const msq_t * msq)
183 {
    return_val_if_fail (msq != NULL, 1);
    return msq->type == -1;
}
```

10.3.7 chat/src/posixmsq.c

```
1 #define _GNU_SOURCE
2 #include <string.h>
3
4 #include "util.h"
5 #include "posixmsq.h"
6
7 typedef struct pmsq pmsq_t;
8
9 struct pmsq {
10     mqd_t mqd;
11     char * name;
12 };
13
14 #define PMSQ(X) ((pmsq_t *) (X))
15
16 /** @brief open a posix message queue
17  * @param name message queue path name
18  * @param flags message queue flags and mode, see msgget(2)
19  * @param id message queue path name id, see ftok(2) */
20 void * open_posix_msq (const char * name, const int flags)
21 {
22     pmsq_t * r = new0 (pmsq_t, 1);
23     if (flags == OPEN_EXCLUSIVE) {
24         struct mq_attr attr = {
25             .mq_flags = 0,
26             .mq_maxmsg = 5,
27             .mq_msgsize = MAX_MSG_SIZE,
28             .mq_curmsgs = 0
29         };
30         r->mqd = mq_open (name, O_EXCL | O_CREAT | O_RDWR, 0600, &attr);
31     } else {
32         r->mqd = mq_open (name, O_RDWR, 0600);
33     }
34     if (r->mqd == (mqd_t) -1)
35         free (r), r = NULL;
36     else
37         r->name = strdup (name);
38     return r;
39 }
40
41 /** @brief send a message over a posix message queue
42  * @param mq message queue identifier
43  * @param msgbuf message buffer */
44 int send_posix_msq (void * mq, const message_t * msgbuf)
45 {
46     return_val_if_fail (mq != NULL, -1);
47     return_val_if_fail (msgbuf != NULL, -1);
48
49     mqd_t mqdes = PMSQ(mq)->mqd;
50     mqd_t r = mq_send (mqdes, (const char *) msgbuf, sizeof (*msgbuf), 1);
51
52     return r == (mqd_t) -1 ? -1 : 0;
53 }
54
55 /** @brief receive a message from a posix message queue
56  * @param mq message queue identifier
57  * @param msgbuf message buffer, where the received message shall be stored */
58 ssize_t recv_posix_msq (void * mq, message_t * msgbuf)
59 {
60     return_val_if_fail (mq != NULL, -1);
61     return_val_if_fail (msgbuf != NULL, -1);
62
63     mqd_t mqdes = PMSQ(mq)->mqd;
64     unsigned prio;
65     return mq_receive (mqdes, (char *) msgbuf, sizeof (*msgbuf), &prio);
66 }
67
```

10 Source Code

```
69 /** @brief remove a posix message queue identifier from the system
70  * @param mq message queue identifier */
71 int rm_posix_msq (void * mq)
72 {
73     return_val_if_fail (mq != NULL, -1);
74     int r = mq_close (PMSQ(mq)->mqd) | mq_unlink (PMSQ(mq)->name);
75     free (PMSQ(mq)->name);
76     free (mq);
77     return r;
78 }
79 /** @brief equality comparison for message queue identifiers */
80 int equal_posix_msq (void * a, void * b)
81 {
82     return_val_if_fail (a != NULL, 0);
83     return_val_if_fail (b != NULL, 0);
84     return ! strcmp (PMSQ(a)->name, PMSQ(b)->name);
85 }
```

10.3.8 chat/src/server.c

```
1 #define _XOPEN_SOURCE
2
3 #include <string.h>
4 #include <errno.h>
5 #include <signal.h>
6
7 #include "server.h"
8 #include "util.h"
9 #include "list.h"
10 #include "sysvmsq.h"
11
12 /** @brief shortcut of a cast to client_t */
13 #define CLIENT(X) ((client_t *) (X))
14
15 /* internal client definition for book keeping on server */
16 typedef struct client client_t;
17
18 /** @brief client representation for book keeping in server process */
19 struct client {
20     /** @brief chat client name (i.e. nick) */
21     char name [CHAT_NAME_LEN];
22     /** @brief message queue where the client listens on */
23     msq_t * msq;
24 };
25
26 /** @brief client name compare */
27 static int client_name_cmp (void * a_, void * b_)
28 {
29     return ! strcmp (CLIENT(a_)->name, b_);
30 }
31
32 /** @brief new client callback for list_for_each
33  * @param l current list item
34  * @param data user data, i.e. the new client */
35 static void server_new_client_hello (void * l, void * data)
36 {
37     const char * msg = "Welcome to the chat...";
38     client_t * client = CLIENT (l);
39     client_t * newcl = CLIENT (data);
40
41     if (equal_msq (client->msq, newcl->msq)) {
42         if (send_message (client->msq, "SERVER", msg, strlen (msg)+1, MTYPE_TEXT) == -1)
43             flog (stderr, "ERROR: Could not send message: %s\n", strerror (errno));
44     } else
45         send_message (client->msq, newcl->name, "", 1, MTYPE_JOIN);
46 }
47
48 /** @brief send exit message to client
49  * @param l current client
50  * @param d exited client */
51 static void server_client_exited (void * l, void * d)
52 {
53     if (-1 == send_message (CLIENT(l)->msq, CLIENT(d)->name, "", 1, MTYPE_EXIT))
54         flog (stderr, "ERROR: Could not send message: %s\n", strerror (errno));
55 }
56
57 /** @brief broadcast a message to all clients but the one in d
```

10 Source Code

```
59  * @param l current client from clients list
60  * @param d originating message from source client */
61  static void server_broadcast_message (void * l, void * d)
62  {
63      message_t * msg = d;
64      client_t * cl = l;
65      if (strcmp (cl->name, msg->sender))
66          send_message (cl->msq, msg->sender, msg->payload, sizeof (msg->payload), MTYPE_TEXT);
67  }
68
69  /** @brief server cleanup handler for atexit(3)
70  * @param msq message queue, to initialize static mq */
71  static void server_cleanup (msq_t * msq)
72  {
73      static msq_t * mq = 0;
74      if (! mq) {
75          mq = msq;
76          atexit ((void (*) (void)) server_cleanup);
77          flog (stdout, "registered exit handler for msq cleanup...\n");
78          return;
79      }
80      flog (stdout, "cleaning up msq...\n");
81      rm_msq (mq);
82      free (mq);
83  }
84
85  /** @brief server signal handler, exit on signal */
86  static void server_handle_signal (int sig)
87  {
88      flog (stdout, "exiting on signal...\n");
89      exit (1);
90  }
91
92  /** @brief server process main function
93  * @param ipcname server ipc path name
94  * @param type ipc API to use */
95  void server (const char * ipcname, const msq_api_t type)
96  {
97      msq_t * mq = NULL;
98      mq = open_msq (ipcname, OPEN_EXCLUSIVE, type);
99      list_p clients = 0;
100
101      if (fail_msq (mq)) {
102          flog (stderr, "ERROR: Could not create message queue: %s\n", strerror (errno));
103          return;
104      }
105
106      server_cleanup (mq);
107      signal (SIGINT, server_handle_signal);
108
109      while (1) {
110          message_t buf;
111          memset (&buf, 0, sizeof (buf));
112          ssize_t size = recv_msq (mq, &buf);
113
114          if (size == -1) {
115              flog (stderr, "ERROR: Could not receive from msq: %s\n", strerror (errno));
116              exit (1);
117          }
118
119          switch (buf.type) {
120              case MTYPE_JOIN:
121                  {
122                      client_t * client = new0 (client_t, 1);
123                      client->msq = open_msq (buf.payload, 0, type);
124                      if (fail_msq (client->msq)) {
125                          flog (stderr, "ERROR: Could not open client MQ: %s\n", strerror (errno));
126                          free (client);
127                          break;
128                      }
129                      strncpy (client->name, buf.sender, strlen (buf.sender));
130                      clients = list_cons (clients, client);
131                      flog (stdout, "Client '%s' joined...\n", buf.sender);
132                      list_for_each (clients, server_new_client_hello, client);
133                      flog (stdout, "Client '%s' announced...\n", buf.sender);
134                  }
135                  break;
136              case MTYPE_EXIT:
137                  {
138                      list_p x = list_remove_first (&clients, client_name_cmp, buf.sender);
139                      if (! x) {
140                          flog (stderr, "ERROR: UNKNOWN CLIENT EXITED\n");
141                          break;
142                      }
143                  }
144          }
145      }
146  }
```

10 Source Code

```
143     flog (stdout, "Client '%s' exited...\n", buf.sender);
144     list_for_each (clients, server_client_exited, CLIENT(x->elem));
145     free (CLIENT(x->elem)->msq);
146     free (x->elem);
147     free (x);
148 }
149 break;
150 case MTYPE_TEXT:
151 {
152     list_p l = list_find_first (clients, client_name_cmp, buf.sender);
153     if (! l) {
154         flog (stderr, "ERROR: UNKNOWN CLIENT\n");
155         break;
156     }
157     flog (stdout, "Broadcasting message from '%s': %s\n", buf.sender, buf.payload);
158     list_for_each (clients, server_broadcast_message, &buf);
159 }
160 break;
161 case MTYPE_PING:
162 {
163     list_p l = list_find_first (clients, client_name_cmp, buf.sender);
164     if (! l) {
165         flog (stderr, "ERROR: UNKNOWN CLIENT\n");
166         break;
167     }
168     send_message (CLIENT(l->elem)->msq, buf.sender, buf.payload, sizeof (buf.payload), MTYPE_PING);
169     flog (stdout, "Replied ping from client '%s'\n", buf.sender);
170 }
171 }
172 }
```

10.3.9 chat/src/sysvmsq.c

```
1 #define _XOPEN_SOURCE 600
2 #define _GNU_SOURCE
3
4 #include <string.h>
5 #include <sys/msg.h>
6
7 #include "util.h"
8 #include "sysvmsq.h"
9
10 /** @brief msgbuf for system v message queue messages */
11 struct xmsgbuf {
12     long mtype;
13     char mtext[MAX_MSG_SIZE];
14 };
15
16 /** @brief open a system v message queue
17  * @param name message queue path name
18  * @param flags message queue flags and mode, see msgget(2)
19  * @param id message queue path name id, see ftok(2) */
20 int open_sysv_msq (const char * name, int flags)
21 {
22     key_t key = ftok (name, IPC_CHAT_ID);
23     flags = (flags == OPEN_EXCLUSIVE ? IPC_EXCL | IPC_CREAT : 0) | 0600;
24     return !name || key == -1 ? -1 : msgget (key, flags);
25 }
26
27 /** @brief send a message over a system v message queue
28  * @param msgid message queue identifier
29  * @param msgbuf message buffer */
30 int send_sysv_msq (int msgid, const message_t * msgbuf)
31 {
32     struct xmsgbuf msg;
33     return_val_if_fail (msgbuf != NULL, -1);
34     msg.mtype = SYSV_MSG_TYPE; /* we do not rely on that crappy sysv api thing */
35     memcpy (msg.mtext, msgbuf, sizeof (message_t));
36     return msgsnd (msgid, &msg, sizeof (msg), 0);
37 }
38
39 /** @brief receive a message from a system v message queue
40  * @param msgid message queue identifier
41  * @param msgbuf message buffer, where the received message shall be stored */
42 ssize_t recv_sysv_msq (int msgid, message_t * msgbuf)
43 {
44     struct xmsgbuf buf;
```

10 Source Code

```
46     ssize_t ret;
47     return_val_if_fail (msgbuf != NULL, -1);
48     ret = msgrcv (msgid, &buf, sizeof (buf), SYSV_MSG_TYPE, 0);
49     if (ret != -1)
50         memcpy (msgbuf, buf.mtext, sizeof (message_t));
51     return ret;
52 }
53
54 /** @brief remove a system v message queue identifier from the system
55  * @param msgid message queue identifier */
56 int rm_sysv_msq (int msgid)
57 {
58     return msgctl (msgid, IPC_RMID, 0);
59 }
60
61 /** @brief equality comparison for message queue identifiers */
62 int equal_sysv_msq (int a, int b)
63 {
64     return a == b;
65 }
```

10.3.10 chat/src/sysvsem.c

```
1 #define _XOPEN_SOURCE
2 #define _GNU_SOURCE
3
4 #include <string.h>
5 #include <assert.h>
6 #include <strings.h>
7
8 #include "util.h"
9 #include "sysvsem.h"
10
11 /** @brief open a system v message queue
12  * @param name message queue path name
13  * @param flags message queue flags and mode, see semget(2)
14  * @param id message queue path id, see ftok(2)
15  * @param nsems number of sempahores to open
16  *
17  * Opens a system v message queue, propagating system errors. */
18 int open_sem (const char * name, int flags, const int nsems)
19 {
20     key_t key = ftok (name, IPC_CHAT_ID);
21     flags = (flags == OPEN_EXCLUSIVE ? IPC_EXCL | IPC_CREAT : 0) | 0600;
22     return key == -1 ? -1 : semget (key, nsems, flags);
23 }
24
25 /** @brief lock a semaphore
26  * @param semid semaphore array identifier
27  * @param semnum semaphore index
28  *
29  * Locks a semaphore, but waits for 0 before locking (i.e. emulating
30  * a degraded semaphore (mutex) */
31 int lock_sem (const int semid, const int semnum)
32 {
33     struct sembuf sb [2] = {
34         { .sem_num = semnum, .sem_op = 0, .sem_flg = 0 },
35         { .sem_num = semnum, .sem_op = 1, .sem_flg = 0 }
36     };
37     return semop (semid, sb, 2);
38 }
39
40 /** @brief unlock a semaphore
41  * @param semid semaphore array identifier to unlock
42  * @param semnum semaphore index */
43 int unlock_sem (const int semid, const int semnum)
44 {
45     struct sembuf sb = { .sem_num = semnum, .sem_op = -1, .sem_flg = 0 };
46     return semop (semid, &sb, 1);
47 }
48
49 /** @brief lock w/o wait\
50  * @param semid sempahore array identifier
51  * @param semnum semaphore index
52  *
53  * This function increments the ressource count of the specified
54  * semaphore by 1, see sub_sem for decrement. */
55 int add_sem (const int semid, const int semnum)
```

10 Source Code

```
57 {
    struct sembuf sb = { .sem_num = semnum, .sem_op = 1, .sem_flg = 0 };
    return semop (semid, &sb, 1);
59 }

61 /** @brief alias for unlock
    * @param semid semaphore array identifier
63 * @param semnum semaphore index */
int (* sub_sem) (const int semid, const int semnum) = unlock_sem;
65

67 /** @brief remove a semaphore
    * @param semid semaphore array identifier to remove
    * @param semnum semaphore index */
69 int rm_sem (const int semid, const int semnum)
    {
71     return semctl (semid, semnum, IPC_RMID);
    }
}
```

10.3.11 chat/src/sysvshm.c

```
1 #define _XOPEN_SOURCE
  #define _GNU_SOURCE
3
4 #include "util.h"
5 #include "sysvshm.h"
6
7 /** @brief open a shared memory segment
    * @param name shm path name to open
    * @param size size of the shm o request
    * @param flags shared memory flags and mode
    * @param id shm pathname id
    *
13 * See shmget(2) for how flags handles shm permissions and shmflags */
shm_t * open_shm (const char * name, const size_t size, int flags)
15 {
    shm_t * s = new0 (shm_t, 1);
17     key_t key = ftok (name, IPC_CHAT_ID);
    if (key == -1) goto error;
19     flags = (flags == OPEN_EXCLUSIVE ? IPC_EXCL | IPC_CREAT : 0) | 0600;
    s->id = shmget (key, size, flags);
21     if (s->id == -1) goto error;
    s->ptr = shmat (s->id, 0, 0);
23     if (s->ptr == (void *) -1) goto eclose;
    return s;
25 eclose:
    shmctl (s->id, IPC_RMID, 0);
27 error:
    return NULL;
29 }

31 /** @brief close a shared memory segment, i.e. detach/unmap it
    * @param shm shared memory to detach */
33 int close_shm (const shm_t * shm)
    {
35     return_val_if_fail (shm != NULL, -1);
    return shmdt (shm->ptr);
37 }

39 /** @brief remove/unlink an shm id
    *
41 * unmap a shared memory segment then remove the shared memory segment */
int rm_shm (const shm_t * shm)
43 {
    close_shm (shm);
45     return shmctl (shm->id, IPC_RMID, 0);
    }
}
```

10.3.12 chat/src/util.c

```
1 #define _GNU_SOURCE
```

10 Source Code

```
3 #include <getopt.h>
4 #include <stdarg.h>
5 #include <time.h>
6 #include <string.h>
7
8 #include "util.h"
9
10 /** @brief send a message
11  * @param msq message queue
12  * @param from sender id, i.e. chat nick name
13  * @param msg text message to send
14  * @param type message type */
15 int send_message (const msq_t * msq, const char * from, const char * msg, const size_t size, const int type)
16 {
17     int fl;
18     message_t buf;
19
20     return_val_if_fail (msq != NULL, -1);
21
22     fl = strlen (from);
23     fl = fl > CHAT_NAME_LEN-1 ? CHAT_NAME_LEN : fl;
24
25     bzero (&buf, sizeof (buf));
26     memcpy (buf.payload, msg, size);
27     memcpy (buf.sender, from, fl);
28     buf.type = type;
29
30     return send_msq (msq, &buf);
31 }
32
33 /** @brief server logging function, prints message with time info */
34 void flog (FILE * f, char * fmt, ...)
35 {
36     char tbuf [30];
37     time_t t;
38     va_list l;
39
40     return_if_fail (f != NULL);
41     return_if_fail (fmt != NULL);
42
43     va_start (l, fmt);
44
45     time (&t);
46     strftime (tbuf, 30, "%T", localtime (&t));
47     fprintf (f, "[%s] ", tbuf);
48     vfprintf (f, fmt, l);
49
50     va_end (l);
51 }
52
53 /** @brief return a newly allocated string representing the users home directory */
54 char * get_user_home (void)
55 {
56     char * p = getenv ("HOME");
57     return strdup (p ? p : (fprintf (stderr, "$HOME not defined, using /", "/")));
58 }
59
60 /** @brief right trim character chr from str
61  * @param str string to trim
62  * @param chr char to trim from str */
63 char * str_rtrim (char * str, char chr)
64 {
65     return_val_if_fail (str != NULL, str);
66     int l = strlen (str);
67     while (str[--l]==chr)
68         str[l]=0;
69     return str;
70 }
```