

Programmation Système

Cours 10 — System V IPC

Stefano Zacchioli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2014-2015

URL <http://epsilon.cc/zack/teaching/1415/progsyst/>
Copyright © 2011-2015 Stefano Zacchioli
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US



Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

- 1 Introduction to System V IPC
- 2 System V Message Queues
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

System V IPC

The expression “**System V IPC**” (or “SysV IPC”, for short) refers to 3 distinct IPC mechanisms:

- 1 SysV **message queues** — used to pass messages between processes
 - ▶ boundary management, i.e. **message granularity**
 - ▶ messages are typed with integer values, allowing to cherry pick messages of a specific type

System V IPC

The expression “**System V IPC**” (or “SysV IPC”, for short) refers to 3 distinct IPC mechanisms:

- 1 SysV **message queues** — used to pass messages between processes
 - ▶ boundary management, i.e. **message granularity**
 - ▶ messages are typed with integer values, allowing to cherry pick messages of a specific type
- 2 SysV **semaphores** — used for process synchronization
 - ▶ kernel-maintained integers that can be **atomically** tested & decremented (or tested & incremented)
 - ▶ “taking” a unit of a semaphore value indicates that the taker is working on a **shared resource**

System V IPC

The expression “**System V IPC**” (or “SysV IPC”, for short) refers to 3 distinct IPC mechanisms:

- 1 SysV **message queues** — used to pass messages between processes
 - ▶ boundary management, i.e. **message granularity**
 - ▶ messages are typed with integer values, allowing to cherry pick messages of a specific type
- 2 SysV **semaphores** — used for process synchronization
 - ▶ kernel-maintained integers that can be **atomically** tested & decremented (or tested & incremented)
 - ▶ “taking” a unit of a semaphore value indicates that the taker is working on a **shared resource**
- 3 SysV **shared memory** – used to share memory regions
 - ▶ similar to shared mmap mappings, but with **kernel persistence**

Why together?

System V IPCs are quite different in function.

Why discuss them together?

- history

late 70s they first appear together in *Columbus UNIX*, a Bell UNIX for database and efficient transaction processing

1983 they land together in *System V* that made them popular in mainstream UNIX-es, hence the name

2001 SUSv3 is published and require implementation of all of them for XSI conformance, hence they are also called **XSI IPC**

- uniformity

- ▶ System V IPC mechanisms share many API traits (persistence, namespace, protocol, etc.)

Overview of System V IPC APIs

System V IPC mechanisms at a glance:

aspect	msg queues	semaphores	shared memory
header	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
data type	msqid_ds	semid_ds	shmid_ds
get object	msgget	semget	shmget / shmat
close object	—	—	shmdt
manipulation	msgctl	semctl	shmctl
communicate	msgsnd / msgrcv	semop	memory access

Note the uniformity in naming and operations, to the extent of what's possible for different communication primitives.

System V IPC concepts

object instance of the IPC mechanism that can be used for communication

- e.g.:
 - ▶ a specific message queue
 - ▶ a (set of) semaphores
 - ▶ a shared memory region
- each object defines a **communication context**

identifier unique identifier for an IPC object

key name used by processes to *rendez-vous* on a common object, to ensure they communicate in the same context

System V IPC protocol

The usage of all System V IPC mechanisms goes through a common “protocol”:

- 1 obtain a key
- 2 get an object
- 3 store its identifier
- 4 communicate through the object

Steps (1) and (2) are optional: object **identifiers are stable integers** that can be stored and passed around as processes see fit.

System V IPC protocol

The usage of all System V IPC mechanisms goes through a common “protocol”:

- 1 obtain a key
 - ▶ *mechanism-independent*
- 2 get an object
 - ▶ passing the key to a *mechanism-specific* syscall
- 3 store its identifier
 - ▶ *mechanism-independent*
- 4 communicate through the object
 - ▶ passing the identifier to *mechanism-specific* syscalls

Steps (1) and (2) are optional: object identifiers are stable integers that can be stored and passed around as processes see fit.

System V IPC vs file-based communication

At a macro level, using System V IPC is similar to communicate via shared files:

file	↔	System V IPC
file name	↔	key
file	↔	object
FD	↔	identifier

Notable **differences between object identifiers and FDs**:

- IPC identifiers are kernel-persistent; FDs are process-persistent
- IPC identifiers are globally visible; FDs are restricted to related processes

Getting IPC object

Each mechanism provide a **get syscall** to get an IPC object

- msgget, semget, shmget

Similarly to open for files, get syscalls are used to either:

- 1 **create a new IPC object** and return its identifier; or
- 2 return the identifier of a **preexisting IPC object**.

Either way, get syscalls receive as arguments:¹

- the **key** for the object we want to get
- a set of **flags** that always support:

IPC_CREAT	request object creation, if needed
IPC_EXCL	request object creation, fail if already exists
S_IRUSR, S_IWUSR, ...	permission mask

the analogy with open continues. . .

¹additional, mechanism-specific args and flags are used by specific syscalls

Getting IPC object — example

```
int id;  
  
/* ... */  
  
/* get message queue for key */  
if ((id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR)) < 0)  
    err_sys("msgget error");
```

Notes:

- the message queue will be created if it doesn't exist, otherwise the existing one corresponding to *key* will be returned
- if created, the message queue will be readable and writable only by processes belonging to the owner of the current process

System V object persistence

System V IPC objects have **kernel persistence**: they remain available, no matter the number of “users” they have, until **kernel shutdown** or **explicit deletion**.

Advantages

- processes can access the object, change its state, and then exit without having to wait; other processes can come up later and check the (modified) state
 - ▶ i.e. System V IPC objects are **stateful and connectionless**

Disadvantages

- IPC objects consume system resources and cannot be automatically garbage collected
 - ▶ hence the need of enforcing **limits** on their quantity
- it's hard to determine when it is safe to delete an object

System V object deletion

Each System V IPC offers a **control syscall** for generic object manipulation

- `msgctl`, `semctl`, `shmctl`

IPC control syscalls are typical dispatcher syscalls that expect an object identifier, a command, and extra command-specific arguments.

Among other things, IPC control syscalls are used to **delete IPC objects**:

- passing the `IPC_RMID` command
- and no extra arguments (i.e. a trailing `NULL`)

Example

```
/* delete shared memory region */  
if (shmctl(id, IPC_RMID, NULL) < 0)  
    err_sys("shmctl error");
```


Shell manipulation of IPC objects

Shell utilities are available to manipulate System V IPC objects:

`ipcs` lists available System V IPC objects

- *which* objects are shown (i.e. all we can read, all we are owner of, etc.) is implementation dependent
- provide extra info: key, id, owner, permissions, etc.
- `ipcs -l` show **system limits** on IPC object counts

`ipcrm` allows to delete IPC objects (we own)

`ipcmk` (non portable) allows to create IPC objects

On Linux, `/proc/sysvipc/` provides a view on *all* existing IPC objects, in a format easier to parse than `ipcs`.

Shell manipulation of IPC objects

Shell utilities are available to manipulate System V IPC objects:

`ipcs` lists available System V IPC objects

- *which* objects are shown (i.e. all we can read, all we are owner of, etc.) is implementation dependent
- provide extra info: key, id, owner, permissions, etc.
- `ipcs -l` show **system limits** on IPC object counts

`ipcrm` allows to delete IPC objects (we own)

`ipcmk` (non portable) allows to create IPC objects

On Linux, `/proc/sysvipc/` provides a view on *all* existing IPC objects, in a format easier to parse than `ipcs`.

Demo

System V IPC keys

System V IPC keys are represented as **key_t integer values**. They get translated to *unique* object identifiers by get syscalls.

The kernel maintains a **key↔identifier association** and guarantees that processes getting IPC objects with the same key:

- will get the **same identifier**
- will **not clash** with identifiers returned for other keys

But we have just *shifted* the problem!

A related problem is: how can processes **choose a common key**, so that they don't end up obtaining the identifier of an IPC object used by other applications?

Assigning System V IPC keys

There are three main strategies to **assign IPC keys to applications**:

- 1 choose an **arbitrary key value** once and for all and distribute it via a common header file

Assigning System V IPC keys

There are three main strategies to **assign IPC keys to applications**:

- 1 choose an **arbitrary key value** once and for all and distribute it via a common header file
- 2 pass the special **constant `IPC_PRIVATE`** as key to get syscalls, which always result in the **creation of a fresh IPC object**
 - ▶ e.g. `id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);`
 - ▶ the creating process will then need to **transmit the object identifier** to collaborating processes by other means

ftok

- 3 use the `ftok` syscall to generate a unique key from the name of an existing file and an arbitrary integer
 - ▶ i.e. `delegate uniqueness` guarantees to the filesystem namespace

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

```
key_t ftok(char *pathname, int proj);
```

Returns: *key value if OK, -1 on error*

- `pathname` points to a file that will be subject to `stat`
- `proj` allows to have different contexts for the same file

Behind the scenes, the 8 least significant bits of `proj` and of the file `i-node` will be used to build a unique key.

ftok — example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include "helpers.h"

int main(int argc, char **argv) {
    key_t key;

    if (argc != 3)
        err_quit("Usage: ftok PATH PROJNO");
    if ((key = ftok(argv[1], atoi(argv[2]))) < 0)
        err_sys("ftok error");
    printf("key=%d\n", key);

    exit(EXIT_SUCCESS);
} /* ftok.c */
```

Demo

Notes:

- ftok is a function in the mathematical sense: equal values map to equal results
 - ▶ the function is not 100% granted to be **injective** though, as only the least significant bits are considered
- the key **depends on the i-node**, i.e. we can't cheat with links
- **directory traversal permissions** can be used to enforce **private contexts** via keys that can't be obtained by 3rd party programs that lack the appropriate permissions

POSIX IPC

POSIX.1b introduced 3 IPC mechanisms equivalent to System V IPC:
POSIX message queues, semaphores, and shared memory.

pros

- simpler API than System V IPC
- consistent naming and usage of IPC objects
 - ▶ /foo/bar/baz instead of ftok paraphernalia
 - ▶ open-, close-, unlink-like syscalls
- reference counting for IPC objects
 - ▶ ease garbage collection

cons

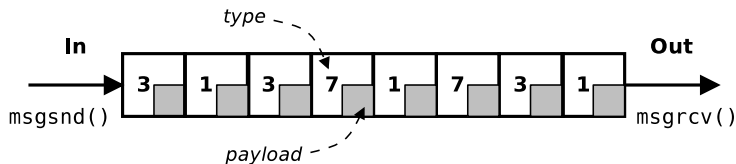
- less portability:
 - ▶ POSIX IPC are optional SUSv3 features
 - ★ *all* implemented in Linux since 2004 “only”
 - ▶ non portable object creation (e.g. would /object work?)
 - ▶ no portable cmdline tools to list/remove IPC objects

Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues**
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

System V message queues

System V **message queues** are IPC objects used for data transfer among unrelated processes.



- communication **granularity** are individual messages
 - ▶ no need to handle message boundaries explicitly as it happened with byte stream
- communication discipline is **first-in, first-out** ...
- ... but messages are **typed** with integer values and can be extracted using **type predicates**
 - ▶ i.e. a queue can be seen as **multiple independent queues**, one per message type, as well as a **priority queue**
- each message carries a **payload** of arbitrary data

Message queues limits & blocking behavior

Message queues are **limited**: in their **total number**, in **message size**, and in the **number of messages**.

```
$ ipcs -l -q
----- Messages Limits -----
max queues system wide = 7786
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
$
```

By default:

- **sending a message** is *not blocking* unless the queue is full, otherwise it's blocking
- **receiving a message** is *blocking* unless a message for the receiver is already available

Overview of System V Message Queue API

header file	<code><sys/msg.h></code>	
get queue	<code>msgget</code>	
send message	<code>msgsnd</code>	
receive message	<code>msgrcv</code>	
data structure	<code>msqid_ds</code>	note the “q”
control	<code>msgctl</code>	

msgget

An existing message queue can be opened or a new one created using the message queue get syscall **msgget**:

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

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

Returns: *message queue ID if OK, -1 on error*

- flags supports IPC_CREAT, IPC_EXCL, and permissions as usual
- the return message queue ID should be stored and used for subsequent usage of the message queue

Reminder: as with all other System V IPC objects, get is not mandatory; message queue IDs are stable and can be passed around by other means.

Messages

A **message** is a structure conforming to the following structure:

```
struct mymsg {  
    long mtype;           /* message type */  
    char mtext[];        /* message payload */  
}
```

- the **payload size** is arbitrary: all message exchanges will specify a (maximum) size
 - ▶ generally, applications decide on a specific message structure and all involved programs stick to it
- as a consequence, **extra trailing payload fields** can be added
- also, `mtext` can have **size 0**
 - ▶ e.g. payload-less messages where all conveyed information is the message type

msgsnd

Once you have a message `msgp`, you can send it through a given message queue `msqid` using `msgsnd`:

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

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);  
Returns: 0 if OK, -1 on error
```

- `msgsz` is the *size of the payload* (i.e. what follows `mtype`)
 - ▶ it will depend on your specific *instantiation* of the `msg`. struct
- `msgflg` supports a single flag:

IPC_NOWAIT | **non-blocking send**

if given, `IPC_NOWAIT` will have `msgsnd` fail with `EAGAIN` if the target queue is full, instead of blocking

msgrcv

The dual of `msgsnd` is `msgrcv`, that is used to receive a message from the queue `msqid` and copy it to `msgp`:

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

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

Returns: *0 if OK, -1 on error*

- `maxmsgsz` is the maximum *payload* size
 - ▶ N.B.: `msgtyp` size is not included in `maxmsgsz`
- `msgtyp` is the **type predicate** that identifies the message we want to receive
 - `msgtyp == 0` first message in the queue
 - `msgtyp > 0` first message `mtype == msgtyp`
 - `msgtyp < 0` first message with the lowest `mtype < abs(msgtyp)`

msgrcv (cont.)

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

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

Returns: *0 if OK, -1 on error*

- msgflg support the following flags:

IPC_NOWAIT

MSG_EXCEPT

MSG_NOERROR

non-blocking receive

complement message selection for msgtyp > 0

allow to remove over-size messages

Message queue associated data structure

To each System V IPC object, the kernel associates a data structure that can be retrieved and set using the appropriate control syscall.

For message queues the data structure is `msqid_ds`, which contains a lot of **accounting information** as well as common properties such as permissions:

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /* ownership and permissions */
    time_t          msg_stime;     /* time of last msgsnd() */
    time_t          msg_rtime;     /* time of last msgrcv() */
    time_t          msg_ctime;     /* time of last change */
    unsigned long   __msg_cbytes; /* n. of bytes in queue */
    msgqnum_t       msg_qnum;      /* n. of messages in queue */
    msglen_t        msg_qbytes;    /* max n. of bytes in queue */
    pid_t           msg_lspid;     /* PID of last msgsnd() */
    pid_t           msg_lrpid;     /* PID of last msgrcv() */
};
```

msgctl

The control syscall of message queues is `msgctl`, a typical dispatcher syscall:

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

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

Returns: *0 if OK, -1 on error*

Allowed commands for `cmd` are:

IPC_RMID	destroy queue; all pending messages are lost
IPC_STAT	get queue data structure into buf
IPC_SET	set queue data structure from buf

Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues**
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

Client/server message queues

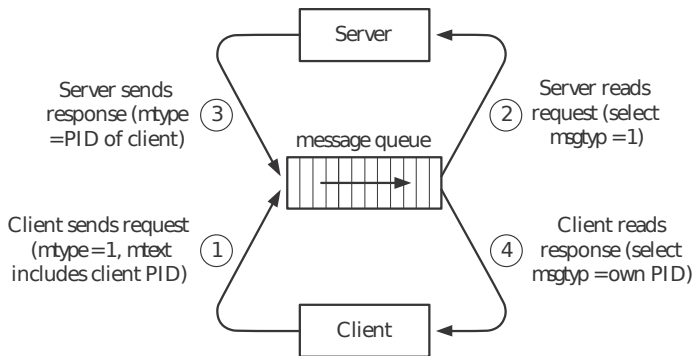
Message queues can be used to implement host-local **client/server architectures**.

Several arrangements are possible:

- 1 single request/response queue
 - ▶ this was not possible with FIFO-based client/server architectures, as there was no way to **route responses** to the appropriate client
- 2 one request queue + one response queue
- 3 one request queue + one response queue *per-client*

Single request/response queue

- we use a **single message queue**
 - ▶ all clients send **requests** there
 - ▶ the server send there **responses** to all clients
- then, to ensure proper **message routing**:
 - ▶ **requests** have **type 1** and contain client PID
 - ▶ **responses** have **type equal to the PID** of the target client



TLPI, Figure 46-2

Single request/response queue — discussion

- it is usually safe to use 1 (or 0) as **server message type** (it's `init`'s PID)
 - ▶ alternatively we can use server's PID, but it is difficult for client to get to know it
- queue **limited capacity** might cause **denial of service** (DoS):
 - 1 many clients might fill the queue blocking the system, as not even the server will be able to write responses
 - 2 a malicious/misbehaving client might fail to retrieve its response; many will block the system

One request queue + one response queue

As a first improvement, we can use two queues:

- client send their requests to a **single request queue**
- server send responses for all clients to a **single response queue**

This arrangement **solves problem (1)**—many clients blocking the queue:

- when the request queue is full, new client will block, but the server will still be able to process requests and send responses without blocking

It **does not solves problem (2)**—clients failing to retrieve responses.

Per client response queue

To improve over problem (2) we need a more complex arrangement.

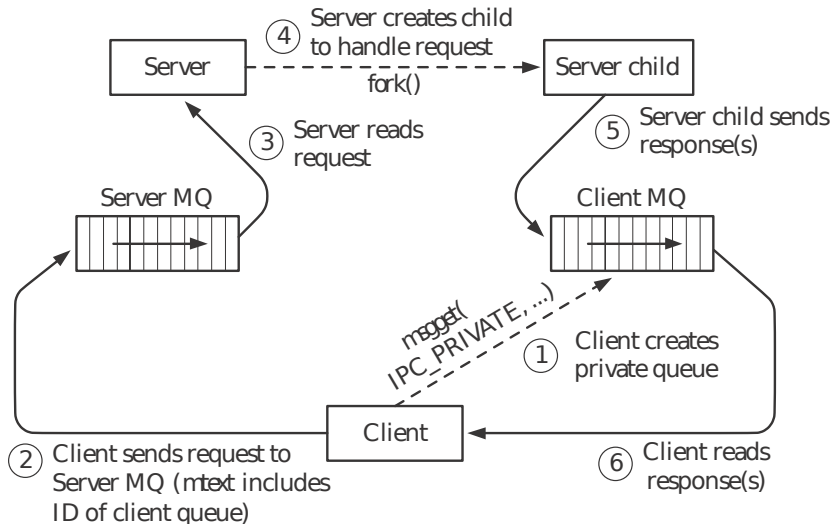
- a single request queue, as before
- one response queue per client
 - ▶ a client creates its own private message queue
 - ▶ the client queue identifier is included in requests
 - ▶ the server retrieves the queue identifier and write responses to the corresponding queue

Per client response queue

To improve over problem (2) we need a more complex arrangement.

- a single request queue, as before
- one response queue per client
 - ▶ a client creates its own private message queue
 - ▶ the client queue identifier is included in requests
 - ▶ the server retrieves the queue identifier and write responses to the corresponding queue
- to avoid blocking on full response queues (e.g. created on purpose by malicious clients), the server fork a child to handle each request

Per client response queue (cont.)



TLPI, Figure 46-3

Per client response queue — example

As an example of the per client response queue architecture we provide an implementation of a:

Example (stat server)

- clients send to the server a `pathname` of a file they want to be stat-ed
 - ▶ possibly, the file is accessible to the server but not to the clients
- the server `stat` the file and return its information to the client
 - ▶ in the example, we return only file size

We will use one shared queue for requests + one private queue per client for responses.

Per client response queue — example (protocol)

```
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define SRV_FTOK_PATH    "/tmp/stat-server" /* dummy value */
#define SRV_FTOK_PROJ   1

#define RES_OK           1      /* OK, checksum returned */
#define RES_ERR         2      /* error, no checksum */
```

Per client response queue — example (protocol) (cont.)

```
struct request_msg {
    long mtype;           /* not used */
    int cli_id;          /* client queue id */
    char pathname[PATH_MAX];
};

struct response_msg {
    long mtype;          /* one of RES_* values */
    size_t size;         /* size (if mtype==RES_OK) */
    /* ... */
};

#define REQ_SIZ (sizeof(struct request_msg) - sizeof(long))
#define RES_SIZ (sizeof(struct response_msg) - sizeof(long))

/* stat-common.h */
```

Per client response queue — example (server)

```
#include "stat-common.h"

static int srv_id;
int handle_request(const struct request_msg *req);
void rm_queue(int signo) {
    msgctl(srv_id, IPC_RMID, NULL);
}
void dezombie(int signo) {
    waitpid(-1, NULL, WNOHANG);
}
int main(void) {
    int msglen;
    key_t srv_key;
    pid_t pid;
    struct request_msg req;

    if ((srv_key = ftok(SRV_FTOK_PATH, SRV_FTOK_PROJ)) < 0)
        err_sys("ftok error");
    if ((srv_id = msgget(srv_key, IPC_CREAT | IPC_EXCL
                        | S_IRUSR | S_IWUSR | S_IWGRP)) < 0)
        err_sys("msgget error");
    if (signal(SIGCHLD, dezombie) == SIG_ERR /* better cleanup needed...
        || signal(SIGINT, rm_queue) == SIG_ERR)
        err_sys("signal error");
```


Per client response queue — example (server) (cont.)

```
for (;;) {
    if ((msglen = msgrcv(srv_id, &req, REQ_SIZ, 0, 0)) < 0) {
        if (errno == EINTR)
            continue;
        break;
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    if (pid == 0) { /* child */
        if (handle_request(&req) < 0)
            err_msg("handle_request error");
        exit(EXIT_SUCCESS);
    }
    /* parent continues to handle next request */
}

rm_queue(-1);
exit(EXIT_SUCCESS);
}
```

Per client response queue — example (server) (cont.)

```
int handle_request(const struct request_msg *req) {
    struct stat finfo;
    struct response_msg res;

    if (stat(req->pathname, &finfo) < 0) {
        res.mtype = RES_ERR;
    } else {
        res.mtype = RES_OK;
        res.size = finfo.st_size;
    }
    return msgsnd(req->cli_id, &res, RES_SIZ, 0);
}

/* stat-server.c */
```

Per client response queue — example (client)

```
#include "stat-common.h"
static int cli_id;
void rm_queue(void) {
    msgctl(cli_id, IPC_RMID, NULL);
}

int main(int argc, char **argv) {
    int srv_id;
    key_t srv_key;
    struct request_msg req;
    struct response_msg res;

    if (argc != 2)
        err_quit("Usage: stat-client PATH");

    if ((srv_key = ftok(SRV_FTOK_PATH, SRV_FTOK_PROJ)) < 0)
        err_sys("ftok error");
    if ((srv_id = msgget(srv_key, 0)) < 0)
        err_sys("msgget error (server queue)");
    if ((cli_id = msgget(IPC_PRIVATE,
                        S_IRUSR | S_IWUSR | S_IWGRP)) < 0)
        err_sys("msgget error (client queue)");
    if (atexit(rm_queue) != 0)
        err_sys("atexit error");
}
```

Per client response queue — example (client) (cont.)

```
req.mtype = 1; /* unused */
req.cli_id = cli_id;
strncpy(req.pathname, argv[1], sizeof(req.pathname) - 1);
req.pathname[sizeof(req.pathname) - 1] = '\0'; /* safeguard */
if (msgsnd(srv_id, &req, REQ_SIZ, 0) < 0)
    err_sys("msgsnd error");

if (msgrcv(cli_id, &res, RES_SIZ, 0, 0) < 0)
    err_sys("msgrcv error");
if (res.mtype == RES_ERR)
    exit(EXIT_FAILURE);
else {
    printf("%ld\n", res.size);
    exit(EXIT_SUCCESS);
}
} /* stat-client.c */
```

Demo

Notes:

- clients cannot DoS the server queue
 - ▶ they can still DoS the system by filling up the number of available queues; this is an intrinsic problem of System V IPC
- clients and servers take good care of queue cleanup
- we don't use message types, but they can be used to provide **Quality of Service** (QoS), by prioritizing requests

Demo

Notes:

- clients cannot DoS the server queue
 - ▶ they can still DoS the system by filling up the number of available queues; this is an intrinsic problem of System V IPC
- clients and servers take good care of queue cleanup
- we don't use message types, but they can be used to provide **Quality of Service** (QoS), by prioritizing requests

Exercise

*Adapt the example to become a **file server**, where the server sends to clients the content of the requested file.*

Disadvantages of System V message queues

Common System V IPC issues:

- object identifiers, rather than FDs
- keys, rather than filenames
- no accounting of object users; hard to delete safely
- limits

Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues
 - Client/server message queues
- 3 System V Semaphores**
- 4 System V Shared Memory
 - Data structures in shared memory

Semaphores

A **semaphore** is an abstract data type whose instances can be used by processes that want to **coordinate access to shared resources**. A semaphore is equipped with operations that allow coordination without **race conditions** between testing the value of the semaphore and changing it.

In its general form, a semaphore is an **integer variable** that can never descend below 0.

Intuitively, such a semaphore counts the **number of available units** of a shared resource.

A special, yet very common, case is that of a **binary semaphore** used to guarantee mutual exclusion when accessing a **single shared resource**. A binary semaphore has only two possible values:

- 1 the resource is available
- 0 the resource is taken



Edsger W. Dijkstra

Cooperating Sequential Processes

Programming Languages, Academic Press, New York. 1968

<http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html>

Semaphore operations

Two operations are defined on a semaphore:

- P** (for *probeer te verlagen*, “try to decrease” in Dutch)
try to decrease semaphore value and blocks if that would result in a negative value; when the operation returns the semaphore value has been decreased by one
intuition: ask for exclusive access to a resource unit
- V** (for *verhogen*, “increase” in Dutch)
increase semaphore value
intuition: signal that a resource unit has been made available to others who are waiting for it

Semaphore — example

You want to coordinate **access to a shared memory area** among multiple processes ensuring **memory coherence**.

More precisely, each process wants to perform an **increment action** on the `int` counter variable stored into shared memory. The increment operation is defined as: read counter and write it back incremented by one (i.e. `counter++`). The desired property is that if n processes perform a total of k increment operations, the counter is incremented exactly by k .

Due to the **race condition** between reading counter value and writing it back incremented, it is not enough for each process to simply do `counter++`

- with more complex data structure, the race condition can also cause **data corruption** that makes impossible to read the value

Semaphore — example (cont.)

With a binary semaphore S we can solve the problem and avoid race conditions. We assume that the semaphore is initialized with $S \leftarrow 1$. Each process will implement increment as follows:

- 1 $P(S)$
- 2 counter++
- 3 $V(S)$

The P call would block if someone else is incrementing counter. Within the $P \dots V$ window a process knows that—as long as all other processes follow the same protocol—he will execute its **critical section** without interference from others.

For semaphores to work, it is essential that $P(S)$ **atomically test & decrement** (if possible) the semaphore value. Only the kernel can guarantee that.

General semaphores

In the example, there is a single resource available (the counter global variable), hence a binary semaphore is enough.

- the scheme can be **generalized** to n available resources using $S \leftarrow n$ as semaphore initialization
- we then generalize P and V to take arbitrary integer parameters
 - ▶ $V(S, n)$ increment semaphore value by n
 - ▶ $P(S, n)$ try to decrease semaphore value by n and block if that is not possible
 - ★ note: until the decrease is possible, semaphore value remains unchanged, i.e. **no partial decrease** is possible

Example

A TP room at the UFR has n available machines. In the morning, the room opens with $S \leftarrow n$. A student arrives at the door and does $P(S, 1)$ to enter. A group of m students working together on a project does $P(S, m)$. Each student leaving the room does $V(S, 1)$.

System V Semaphores

The traditional implementation of semaphores by UNIX kernels are **System V semaphores**.

- they belong to System V IPC mechanisms; the IPC object is a *set of semaphores*
- all semaphores in the set are **generalized semaphores** with arbitrary (non-negative) integer values
- each semaphore in a set support the following **operations**:
 - 1 add a (positive) value to semaphore value (generalized V)
 - 2 subtract a (positive) number from semaphore value (generalized P)
 - 3 set semaphore value to an arbitrary (non-negative) value (initialization)
 - 4 wait for semaphore value to become 0

Overview of System V semaphore API

header file	<code><sys/sem.h></code>
get semaphore set semaphore action	<code>semget</code> <code>semop</code>
data structure control	<code>semid_ds</code> <code>semctl</code>

semget

To obtain a semaphore set corresponding to a given key, the get syscall `semget` should be used:

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

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

Returns: *semaphore set ID if OK, -1 on error*

- `nsems` specifies the **number of semaphores in the set**
 - ▶ 1 is a common choice
- `semflg` supports `IPC_CREAT`, `IPC_FLAGS`, and permissions as usual

semop

All semaphore operations are requested via the `semop` syscall. The fact that the object granularity is the semaphore *set*, makes it rather cumbersome to use. . .

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

```
int semop(int semid, struct sembuf *sops, unsigned int nsops);  
Returns: 0 if OK, -1 on error
```

- `semid` is the semaphore set identifier
- `sops` points to a (non-empty) **array of operations** to be performed on individual semaphores

```
struct sembuf {  
    unsigned short sem_num;    /* semaphore number, 0-based */  
    short          sem_op;     /* operation to be performed */  
    short          sem_flg;    /* operation flags */  
};
```

- `nsops` is the number of operations in the `sops` array

semop (cont.)

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

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

Returns: 0 if OK, -1 on error

```
struct sembuf {  
    unsigned short sem_num;    /* semaphore number, 0-based */  
    short          sem_op;     /* operation to be performed */  
    short          sem_flg;    /* operation flags */  
};
```

Operations are interpreted as follows:

$\text{sem_op} > 0$	\rightarrow	$V(S_{\text{sem_num}}, \text{sem_op})$
$\text{sem_op} < 0$	\rightarrow	$P(S_{\text{sem_num}}, \text{sem_op})$
$\text{sem_op} = 0$	\rightarrow	test & wait until $S_{\text{sem_num}} = 0$

operation flags support IPC_NOWAIT with the usual semantics

Semaphore associated data structure

As with all System V IPC objects, a kernel data structure is associated to each semaphore set:

```
struct semid_ds {  
    struct ipc_perm msg_perm;      /* ownership and permissions */  
    time_t      sem_otime;        /* time of last semop() */  
    time_t      sem_ctime;        /* time of last change */  
    unsigned long sem_nsems;     /* n. of semaphores in set */  
};
```

It allows to access IPC object permissions and also provides the usual accounting information.

semctl

The semaphore set control syscall is `semctl`. Contrary to other control syscalls, `semctl` is overloaded for two different purposes:

- 1 generic control operations (access to `semid_ds`, deletion)
- 2 semaphore initialization & reset

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

```
int semctl(int semid, int semnum, int cmd, ... /*union semun arg */);  
Returns: non-negative integer if OK, -1 on error
```

Generic control operations are as usual according to `cmd`:

- `IPC_RMID` to delete the semaphore set
- `IPC_STAT/IPC_SET` to get/set the associated data structure

semctl (cont.)

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

```
int semctl(int semid, int semnum, int cmd, ... /*union semun arg */);  
Returns: non-negative integer if OK, -1 on error
```

Semaphore initialization / reset uses the following cmd-s:

- GETVAL/SETVAL: get/set the value of the setnum-th semaphore
- GETALL/SETALL: as above, but for *all* semaphores at once

Arguments and return values for all semctl commands are conveyed by semun, that must be defined by your programs as follows:

```
union semun {  
    int          val;          /* individual semaphore value */  
    struct semid_ds *buf;     /* semaphore data structure */  
    unsigned short *array;    /* multiple semaphore values */  
};
```

Semaphore initialization

- according to SUSv3, semaphores are **not initialized** to any specific value when created with `semget+IPC_CREAT`
 - ▶ Linux initializes semaphores to 0, but it's not portable
- therefore, semaphores must be explicitly initialized *after* creation with `semctl`

If multiple processes attempt to create & initialize a common semaphore you get a **race condition** for semaphore initialization.

Semaphore initialization

- according to SUSv3, semaphores are **not initialized** to any specific value when created with `semget+IPC_CREAT`
 - ▶ Linux initializes semaphores to 0, but it's not portable
- therefore, semaphores must be explicitly initialized *after* creation with `semctl`

If multiple processes attempt to create & initialize a common semaphore you get a **race condition** for semaphore initialization.

Solutions:

- 1 **avoidance**: ensure that a single process is in charge of creating & initializing the semaphore, e.g.:
 - ▶ creation at (application) boot time, possibly using `ipcmk`
 - ▶ have a parent create & initialize before spawning children
- 2 the **semotime trick** based on the historical (and now standard) fact that the field is set to 0 upon creation

Semaphore — example

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

union semun {    /* SUSv3 mandates definition in user programs */
    int          val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

Semaphore — example (cont.)

```
int main(int argc, char **argv) {
    int semid;

    if (argc < 2 || argc > 3)
        err_quit("Usage: svsem_demo [ SEM_ID ] SEM_OP");

    if (argc == 2) {          /* create & init semaphore */
        union semun arg;

        if ((semid = semget(IPC_PRIVATE, 1,
                            S_IRUSR | S_IWUSR)) < 0)
            err_sys("semid error");
        arg.val = atoi(argv[1]);

        if (semctl(semid, 0, SETVAL, arg) < 0)
            err_sys("semctl error");

        printf("semaphore ID = %d\n", semid);
    }
}
```

Semaphore — example (cont.)

```
else {          /* act on first semaphore */
    struct sembuf sop;

    semid = atoi(argv[1]);
    sop.sem_num = 0;          /* act on first semaphore */
    sop.sem_op = atoi(argv[2]);
    sop.sem_flg = 0;          /* no special flags */

    printf("%d: about to semop\n", getpid());
    if (semop(semid, &sop, 1) < 0)
        err_sys("semop error");

    printf("%d: semop completed\n", getpid());
}

exit(EXIT_SUCCESS);
}
/* svsem_demo.c. Based on TLPI's svsem_demo.c.
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL-3+ */
```

Demo

Semaphore limits

As other System V IPC objects, semaphores are **limited** along many traits, many of which can be customized:

```
$ ipcs -l -s
----- Semaphore Limits -----
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767
$
```

Disadvantages of System V semaphores

Common System V IPC issues:

- object identifiers, rather than FDs
- keys, rather than filenames
- no accounting of object users; hard to delete safely
- limits

Semaphore-specific issues:

- initialization race condition
- overly complex API, “thanks” to the set granularity

Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

System V Shared Memory

System V **shared memory** is an IPC mechanism that allows unrelated processes to share memory areas. In the System V shared memory jargon, shared memory areas are called **segments**.

As with other System V IPC mechanisms, segments are kernel-persistent and should be explicitly created and destroyed.

System V Shared Memory

System V **shared memory** is an IPC mechanism that allows unrelated processes to share memory areas. In the System V shared memory jargon, shared memory areas are called **segments**.

As with other System V IPC mechanisms, segments are kernel-persistent and should be explicitly created and destroyed.

For many purposes, System V shared memory is similar to **memory mappings** that we already discussed:

- different processes access virtual pages pointing to the **same memory frames**
- communication happens via **direct memory access**
- however, segments can be shared among **unrelated processes without a file** as backing store
 - ▶ with memory mappings either you map a file, or you can share only among related processes (via anonymous mappings)

Shared memory protocol

The “**protocol**” to use shared memory segments, unlike other System V mechanisms, require more than one step before actual IPC.

- 1 obtain the **identifier** of a shared memory segment
 - ▶ this can happen via a get syscall (shmget) that either create a segment or retrieve its identifier
 - ▶ or via communication of the identifier by other means
- 2 **attach the segment** to the process address space
 - ▶ as a result of attaching, the caller obtain the **starting memory address** of the shared segment
- 3 use the attached segment, as if it were native process memory
 - ▶ no mediation by the syscall API is needed
 - ▶ as with memory mappings, synchronization among sharing processes is an important concern here

Shared memory protocol (cont.)

- 4 once a process is done using shared memory, it should **detach the segment** from its address space
 - ▶ once detached, access to the memory corresponding to the (now detached) segment will cause SIGSEGV
- 5 once all processes are done using shared memory, a process should **delete the segment** using the control syscall with `IPC_RMID`
 - ▶ until this step is done, the shared memory segment will continue to exist (and to occupy system resources)

Overview of System V Shared Memory API

header file	<code><sys/shm.h></code>
get segment	<code>shmget</code>
attach segment	<code>shmat</code>
detach segment	<code>shmdt</code>
data structure	<code>shmid_ds</code>
control	<code>shmctl</code>

shmget

shmget is the get syscall for shared memory segments:

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

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

Returns: *shared memory segment id if OK, -1 on error*

- key is the System V key, obtained by the usual means
- size is the size of the desired memory segment, in bytes
 - ▶ the kernel will return a segment of size multiple to system page size, **roundup** to the next such multiple will happen
- shmflg is a bitwise OR of the usual System V IPC flags: IPC_CREAT, IPC_EXCL, permission bits
- on success, shmget returns a shared memory segment identifier that can be stored and passed to others

shmat

Once a process has obtained a segment identifier, it should use it to attach the segment to its address space using `shmat`, which behaves similarly to `mmap`:

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

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

Returns: *base address of the attached segment if OK, -1 on error*

- `shmid` is the identifier of the segment to attach
- `shmaddr` is the address **where to attach the segment**
 - ▶ it is **optional**, if omitted the kernel will choose a suitable address
 - ▶ contrary to `mmap`, **it is not a hint**
 - ★ either attachment at `shmaddr` is possible
 - ★ or `shmat` will fail with `EINVAL`
- on success, `shmat` will return the starting address of the attached segment

Where to attach segments?

The main reason to force attachment at a specific address is to **ensure consistency** throughout all processes that share the segment.

Example

A group of processes want to collaborate on a big array of struct `mystruct` in shared memory.

To do so, they all attach a suitably-sized segment starting at address `0x6ed000`.

Once done, all processes know that the 1234-th array entry can be found at address:

$$0x6ed000 + \text{sizeof}(\text{struct mystruct}) * (1234 - 1)$$

Otherwise, addresses pointing to specific array entries in different processes might be different.

Where to attach segments? (cont.)

Attaching segments at fixed addresses using `shmaddr` comes with its own problems though

- at compile time, when we write the program, we often **don't know if `shmaddr`** will be available at runtime for attaching the segment; in particular:
 - ▶ we don't know if another segment will be attached at `shmaddr`
 - ▶ we don't know if at `shmaddr` there will be enough available **contiguous memory** to fit the segment+roundup
- it **reduces portability**, as an address valid on some UNIX machine will not necessarily be valid on different UNIX-es
- `shmaddr` is less flexible than `mmap`'s `addr` hint, as `mmap` is able to fallback to a difference address
 - ▶ unless you requested `MAP_FIXED`

As a best practice, *never* use `shmaddr` and rely on the kernel to find a suitable address for you.

(Stay tuned for how to solve the uniform memory address problem.)

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

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

Returns: *base address of the attached segment if OK, -1 on error*

Flags is a bitwise OR of flags that include:

- SHM_RDONLY: attach segment read-only
- SHM_REMAP: replace already existing mapping at shmaddr

shmdt

Once done working on a shared memory segment, a process can detach it from its address space using **shmdt**:

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

```
int shmdt(const void *shmaddr);
```

Returns: *0 if OK, -1 on error*

Note that detaching is different than deleting. Once detached, the memory segment (**and its content**) will still exist in kernel space until explicitly deleted.

Segment inheritance

Attached memory segments are automatically **detached upon process termination**.

Attached memory segments are **inherited through fork**— as it happens with memory mappings. That provides a(nother) handy way to do IPC between parent and child

- 1 parent creates shared memory segment using key `IPC_PRIVATE`
- 2 parent fork-s a child
- 3 child and parent can now transfer data via shared memory

Note: the same is neither possible via regular memory due to address space copying (on write), nor via `vfork` as parent is blocked until child “reset”.

Shared memory — example

As an example, we will develop a simple command line toolkit to transfer data across processes with the mediation of the kernel and without relying on the filesystem. It will work as follows:

- we create shared memory segments using `ipcmk`, and note down the shared memory identifier (`shmId`) that it returns
- we copy data from standard input to shared memory with the custom program `stdin2shm`
- we copy data from shared memory to stdout with the custom program `shm2stdout`
 - ▶ for both programs we will need to specify on the cmdline shared memory segment identifier and size
- once done, we use `ipcrm` to free the used segment

For simplicity, copied data (in both directions) will be `trimmed` to

$\min(\textit{stdin size}, \textit{segment size})$

Shared memory — example (protocol)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <unistd.h>
#include "helpers.h"

#define min(x,y)      (x > y ? y : x)

static int shmids = -1;
static int shmsize = -1;

struct shmseg {
    int size;      /* used data in buf */
    char buf[];   /* actual data */
};
```

Shared memory — example (protocol) (cont.)

```
/* parse cmdline and fill: shmid, shmsize */
void parse_cmdline(int argc, char **argv) {
    char msg[1024] = "";

    if (argc != 3) {
        snprintf(msg, 1024, "Usage: %s SHMID SIZE", argv[0]);
        err_quit(msg);
    }
    shmid = atoi(argv[1]);
    shmsize = atoi(argv[2]);
}

/* shm-common.h */
```

Shared memory — example (stdin2shm)

```
#include "shm-common.h"
```

```
int main(int argc, char **argv) {  
    int n, bufsize;  
    struct shmseg *shmp;  
    parse_cmdline(argc, argv);  
  
    if ((shmp = shmat(shmid, NULL, 0)) < 0)  
        err_sys("shmat error");  
    fprintf(stderr, "attached segment at: %p\n", shmp);  
  
    bufsize = shmsize - sizeof(int);  
    shmp->size = 0; /* running count of read bytes */  
    while (bufsize - shmp->size >= 0  
           && (n = read(STDIN_FILENO, shmp->buf + shmp->size,  
                       bufsize - shmp->size)) > 0)  
        shmp->size += n;  
    if (n < 0) err_sys("read error");  
    fprintf(stderr, "stdin -> shm: copied %d bytes\n", shmp->size);  
    exit(EXIT_SUCCESS);  
} /* stdin2shm.c */
```

Shared memory — example (shm2stdout)

```
#include "shm-common.h"
int main(int argc, char **argv) {
    int n, done, rem;
    struct shmseg *shmp;
    parse_cmdline(argc, argv);

    if ((shmp = shmat(shmid, NULL, 0)) < 0)
        err_sys("shmat error");
    fprintf(stderr, "attached segment at: %p\n", shmp);

    done = 0;
    rem = min(shmp->size, shmsize); /* written bytes */
    while(rem > 0
        && (n = write(STDOUT_FILENO, shmp->buf + done, rem)) > 0)
        done += n;
        rem -= n;
    }
    if (n < 0) err_sys("write error");
    fprintf(stderr, "shm -> stdout: copied %d bytes\n", done);
    exit(EXIT_SUCCESS);
} /* shm2stdout.c */
```


Demo

Notes:

- shared memory segments are stable, content copied there **persists** beyond the life of processes accessing it
- shared memory **identifiers are stable** (and readable), we can create and pass them around between different programs
- **setup and tear-down** are delegated to `ipcmk` and `ipcrm`
- we do **no synchronization** whatsoever

Exercise

Can we avoid specifying segment size on the cmdline?

Shared memory synchronization — example

Any “real” use of shared resources memory needs synchronization to avoid race conditions and memory corruptions. Consider the following use case:

Example (efficient process-to-process data transfer)

We want to transfer a big amount of data among two unrelated processes.

For maximum efficiency:

- we want to minimize the amount of memory copies and of context switches between user space and kernel space
- we do not want to use the filesystem

System V shared memory offers an interesting solution.

Shared memory synchronization — example (cont.)

- we use a **shared memory segment** as a buffer for data exchange among two processes
- two processes are involved: a **writer** (reading from stdin and writing to shared memory) + a **reader** (reading from shared memory and writing to stdout)
- as the buffer is in general not enough to hold the entire amount of data to be transferred we do **several iterations** of
 - (i) writer writes, (ii) reader reads
- therefore we need to ensure proper **synchronization**:
 - 1 *mutual exclusion*: the reader does not act when the writer does and vice-versa
 - 2 reader and writer *alternates* in accessing data
 - ★ i.e. each written chunk is read exactly once

Shared memory synchronization — example (cont.)

To do so we use an array of 2 System V semaphores:

- 1 a **write semaphore** W
 - ▶ intuition: when the semaphore is taken, writing is happening
- 2 a **read semaphore** R
 - ▶ intuition: when the semaphore is taken, reading is happening

At system bootstrap, we **initialize** the semaphores so that **only writing can happen**:

- write semaphore is initialized $W \leftarrow 1$ (available)
- read semaphore is initialized $R \leftarrow 0$ (not available)

Shared memory synchronization — example (cont.)

writer()

while true do

① $P(W)$

② $shmem \leftarrow stdin$

③ $V(R)$

reader()

while true do

① $P(R)$

② $stdout \leftarrow shmem$

③ $V(W)$

Discussion:

- at bootstrap: reader is blocked at (1); writer do a full iteration
- when writer does (3), reader is unblocked;
when writer does (1) (again), it blocks (sides exchanged)
- reader do a full iteration, unblocks writer at (3), and blocks
when doing (1) again (sides exchanged again, loop)

Shared memory sync. — example (protocol)

```
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "helpers.h"

#define SHM_KEY          0x1234  /* shmem key */
#define SEM_KEY          0x5678  /* semaphore set key */
#define OBJ_PERMS       (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
#define WRITE_SEM       0        /* write index in semaphore set */
#define READ_SEM        1        /* read index in semaphore set */
#define BUF_SIZE        10240    /* transfer buffer size */

union semun { /* SUSv3 forces definition in user programs */
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

struct shmseg {
    int cnt; /* bytes used in 'buf' */
    char buf[BUF_SIZE]; /* data */
};
```

Shared memory sync. — example (library)

```
/* init semaphore to 1 */
int sem_init_avail(int semid, int semno) {
    union semun arg;

    arg.val = 1;
    return semctl(semid, semno, SETVAL, arg);
}

/* init semaphore to 0 */
int sem_init_taken(int semid, int semno) {
    union semun arg;

    arg.val = 0;
    return semctl(semid, semno, SETVAL, arg);
}
```

Shared memory sync. — example (library) (cont.)

```
int sem_p(int semid, int semno) { /* reserve semaphore (P) */
    struct sembuf sop;

    sop.sem_num = semno;
    sop.sem_op = -1;
    sop.sem_flg = 0;
    return semop(semid, &sop, 1);
}

int sem_v(int semid, int semno) { /* release semaphore (V) */
    struct sembuf sop;

    sop.sem_num = semno;
    sop.sem_op = 1;
    sop.sem_flg = 0;
    return semop(semid, &sop, 1);
}

/* svshm-xfr.h, based on svshm_xfr.h
   Copyright (C) Michael Kerrisk 2010, License: GNU AGPL-3+ */
```


Shared memory sync. — example (writer)

```
#include "svshm-xfr.h"

int main(int argc, char **argv) {
    int semid, shmid, bytes, xfrs;
    struct shmseg *shmp;
    union semun dummy;

    if ((semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS)) < 0)
        err_sys("semget error");
    if (sem_init_avail(semid, WRITE_SEM) < 0)
        err_sys("semctl error (sem_init_avail)");
    if (sem_init_taken(semid, READ_SEM) < 0)
        err_sys("semctl error (sem_init_taken)");

    if ((shmid = shmget(SHM_KEY, sizeof(struct shmseg),
                        IPC_CREAT | OBJ_PERMS)) < 0)
        err_sys("shmget error");
    if ((shmp = shmat(shmid, NULL, 0)) < 0)
        err_sys("shmat error");
```

Shared memory sync. — example (writer) (cont.)

```
for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
    if (sem_p(semid, WRITE_SEM) < 0)
        err_sys("semop error (sem_p)");
    if ((shmp->cnt = read(STDIN_FILENO, shmp->buf,
                        BUF_SIZE)) < 0)
        err_sys("read error");
    if (sem_v(semid, READ_SEM) < 0)
        err_sys("semop error (sem_v)");

    /* Test for EOF after reader turn so that
       it can see the 0 value in shmp->cnt. */
    if (shmp->cnt == 0)
        break;
}

if (sem_p(semid, WRITE_SEM) < 0)
    err_sys("semop error (sem_p)");
```

Shared memory sync. — example (writer) (cont.)

```
if (semctl(semid, 0, IPC_RMID, dummy) < 0)
    err_sys("semctl error");
if (shmdt(shmp) < 0)
    err_sys("shmdt error");
if (shmctl(shmid, IPC_RMID, 0) < 0)
    err_sys("shmctl error");

fprintf(stderr, "sent %d bytes (%d xfrs)\n", bytes, xfrs);
exit(EXIT_SUCCESS);
}
```

```
/* svshm-writer.c, based on svshm_xfr_writer.c
   Copyright (C) Michael Kerrisk 2010, License: GNU AGPL-3+ */
```

Shared memory sync. — example (reader)

```
#include "svshm-xfr.h"

int main(int argc, char **argv) {
    int semid, shmid, xfrs, bytes;
    struct shmseg *shmp;

    if ((semid = semget(SEM_KEY, 0, 0)) < 0)
        err_sys("semget error");
    if ((shmid = shmget(SHM_KEY, 0, 0)) < 0)
        err_sys("shmget error");

    if ((shmp = shmat(shmid, NULL, SHM_RDONLY)) < 0)
        err_sys("shmat error");
```

Shared memory sync. — example (reader) (cont.)

```
for (xfrs = 0, bytes = 0; ; xfrs++) {
    if (sem_p(semid, READ_SEM) < 0)
        err_sys("semop error (sem_p)");

    if (shmp->cnt == 0)      /* writer EOF */
        break;
    bytes += shmp->cnt;

    if (write(STDOUT_FILENO, shmp->buf, shmp->cnt)
        != shmp->cnt)
        err_quit("partial/failed write");

    if (sem_v(semid, WRITE_SEM) < 0)
        err_sys("semop error (sem_v)");
}
```

Shared memory sync. — example (reader) (cont.)

```
if (shmdt(shmp) < 0)
    err_sys("shmdt error");

/* one more writer turn to cleanup */
if (sem_v(semid, WRITE_SEM) < 0)
    err_sys("releaseSem");

fprintf(stderr,
        "received %d bytes (%d xfrs)\n", bytes, xfrs);
exit(EXIT_SUCCESS);
}
```

```
/* svshm-reader.c, based on svshm_xfr_reader.c
   Copyright (C) Michael Kerrisk 2010, License: GNU AGPL-3+ */
```

Demo

Notes:

- semaphore initialization is delegated to writer
 - ▶ it should start first, otherwise reader won't find the shared memory segment (and fail)
- we need a way to inform reader of EOF; to do so we let reader read `cnt == 0`
- before writer termination (that will destroy the shared memory segment), we wait for the last reader turn to happen
 - ▶ after it, reader liberates writer using `sem_v` one last time

Shared memory associated data structure

As all System V IPC mechanisms, each shared memory segment is associated to a specific data structure, `shmid_ds` in this case:

```
struct shmid_ds {
    struct ipc_perm msg_perm;      /* ownership and permissions */
    size_t shm_segsz;             /* size of segment (bytes) */
    time_t shm_atime;             /* time of last shmat() */
    time_t shm_dtime;             /* time of last shmdt() */
    time_t shm_ctime;             /* time of last change */
    pid_t shm_cpid;               /* PID of creator */
    pid_t shm_lpid;               /* PID of last shmat/shmdt() */
    shmatt_t shm_nattach;         /* n. of attached processes */
}
```

It contains the usual ownership and accounting information.

shmctl

The control syscall for shared memory segments is `shmctl`:

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

```
shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: *0 if OK, -1 on error*

It supports the usual operations: `IPC_RMID` (removal), `IPC_STAT` (retrieve `shmid_ds`), `IPC_SET` (update `shmid_ds`).

Segment removal semantics

To avoid disruptions in process address spaces, a shared memory segment is *not* removed immediately after `IPC_RMID`. Instead, the kernel does **reference counting** on the number of attached processes and destroys the segment only when the number reaches 0.

- we can ensure proper segment cleanup by **removing the segment right after attaching it**
- ... but whether new attaches are allowed or not after `IPC_RMID` is implementation dependent
 - ▶ Linux allows it

Shared memory limits

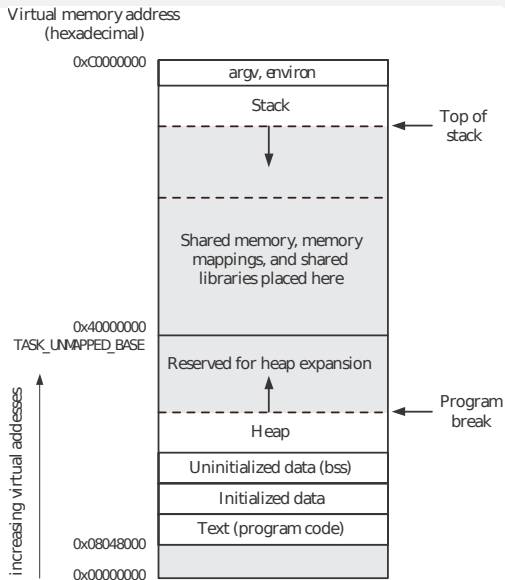
Like other System V IPC mechanisms, shared memory segments are **limited**:

```
$ ipcs -l -q
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1
$
```

Outline

- 1 Introduction to System V IPC
- 2 System V Message Queues
 - Client/server message queues
- 3 System V Semaphores
- 4 System V Shared Memory
 - Data structures in shared memory

Shared memory location



TLPI, Figure 48-2

If we let the kernel choose, all shared memory areas—memory mappings, System V shared segments, and shared libraries—will be located in a common memory area **between the stack and the heap**.

- on Linux, the starting address for shared memory areas is the compile-time constant `TASK_UNMAPPED_BASE`

Shared memory areas are per-process. There is no guarantee that co-operating processes will attach a shared area at the same address.

Data structures in shared memory

At the **conceptual level**, most non-trivial data structures are **interlinked**, i.e. parts of the structure *refers to* other parts of it.

At the **physical memory level**, we have memory cells that contains parts of a data structure (say, p) that points to other parts of it (say, $target$).

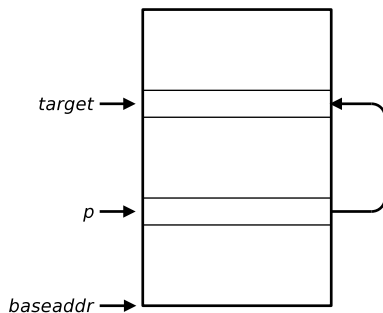


Figure: conceptual memory layout of data structure in shared memory

Data structures in shared memory (cont.)

For a memory part to reference another, we usually store the **absolute memory address** of the destination as part of the source. In presence of shared memory regions that might be attached at different addresses **this is wrong**:

- the absolute address of target might be different from process to process
 - ▶ it *might* happen that the address is the same, but there is no guarantee of it
- also, there is no way of knowing where other processes have attached a shared memory area

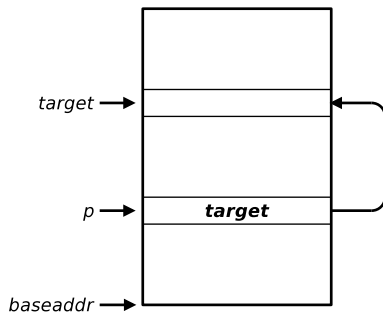


Figure: wrong/risky with shared memory

Data structures in shared memory

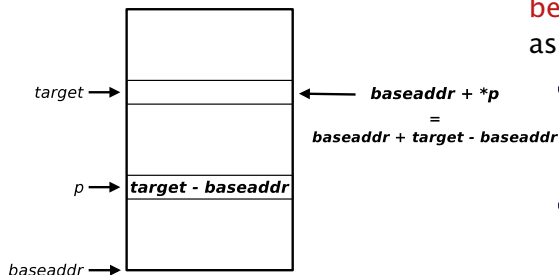


Figure: shared memory safe

The solution is to use addresses relative to the beginning of the shared area as pointers.

- keep around `baseaddr`, as returned by `mmap` or `shmat`
- to store a pointer to shared memory:
`*p = (target - baseaddr);`
- to dereference a pointer:
`target = baseaddr + *p;`

Data structures in shmem — example

Let's reconsider the ad-hoc memory allocation mechanism we implemented using mmap (shared + anonymous). That example had no problem with absolute addressing:

- the memory mapping was created by a common ancestor before fork
- due to address space copy, the absolute addresses were granted to be the same for all involved processes

We want to do the same, but this time among **unrelated processes** that **cooperate on a linked list**:

- `shm-list-writer` sets up a linked list in a shared memory segment
- `shm-list-reader` visit the list printing its content

Data structures in shmem — example (protocol)

```
#include <errno.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define SHM_KEY          0x1234 /* shmem key */
#define OBJ_PERMS       (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)

struct list {
    int val;
    struct list *next;
};

static struct list *list_bot;
static struct list *list_top;
static long list_siz;
```

Data structures in shmem — example (library)

```
struct list *list_get(void) {
    int shmid;
    if ((shmid = shmget(SHM_KEY, 0, 0)) < 0)
        return ((void *) -1);
    list_bot = shmat(shmid, NULL, 0);
    printf("list attached at %p\n", list_bot);
    return list_bot;
}

int list_init(long len) {
    int shmid;
    if ((shmid = shmget(SHM_KEY, len * sizeof(struct list),
                       IPC_CREAT | OBJ_PERMS)) < 0)
        return (-1);
    if ((list_top = shmat(shmid, NULL, 0)) < 0)
        return (-1);
    printf("list attached at %p\n", list_top);
    list_bot = list_top;
    list_siz = len;
    printf("list_init: top=%p, len=%ld\n", list_top, len);
    return 0;
}
```

Data structures in shmem — example (library) (cont.)

```
struct list *list_alloc() {  
    long siz = list_top - list_bot;  
    if (siz >= list_siz) {  
        errno = ENOMEM;  
        return NULL;  
    }  
    list_top->next = NULL;  
    printf("allocated %p (length: %ld)\n", list_top, siz + 1);  
    return list_top++;  
}  
  
struct list *list_free() {  
    /* left as an exercise */  
    return NULL;  
}
```

Data structures in shmem — example (library) (cont.)

```
struct list *list_append(struct list *l, int val) {
    struct list *elt;
    if ((elt = list_alloc()) == NULL)
        return NULL;
    elt->val = val;
    l->next = (void *) ((long) elt - (long) list_bot);
    return elt;
}
```

```
void visit_list(const char *label, struct list *l) {
    printf("[%s] visit list: ", label);
    while (l != NULL) {
        printf("%d ", l->val);
        if (l->next == NULL)
            l = NULL;
        else
            l = (void *) ((long) list_bot + (long) l->next);
    }
    printf("\n");
}
```

Data structures in shmem — example (writer)

```
#include "shm-list-common.h"

int main(void) {
    struct list *head, *l = NULL;

    if (list_init(1000) < 0) err_sys("list_init error");
    if ((l = list_alloc()) == NULL)
        err_sys("list_alloc");
    l->val = 13;
    head = l;
    if ((l = list_append(l, 17)) == NULL
        || (l = list_append(l, 42)) == NULL)
        err_sys("list_append");
    visit_list("common", head);

    exit(EXIT_SUCCESS);
} /* shm-list-writer.c */
```

Data structures in shmem — example (reader)

```
#include "shm-list-common.h"

int main(void) {
    struct list *l = NULL;

    if ((l = list_get()) < 0)
        err_sys("list_get error");

    visit_list("common", l);

    exit(EXIT_SUCCESS);
} /* shm-list-reader.c */
```

Demo

Notes:

- we have switched from a cons semantics to an **append semantics**
 - ▶ otherwise the head pointer would be lost in the data structure
 - ▶ alternatively: we could have stored a pointer to the head in `struct list` and kept it up to date
- shared memory segments are **attached to different addresses** on a regular basis!
- we need quite a bit of (scary) **cast gymnastics** to make the compiler happy (i.e. we really are on our own, beware!)