τ IPC Manual

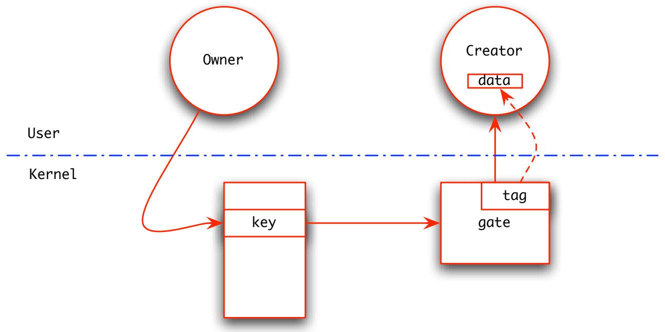
Paul Taysom

November 30, 2009 : April 25, 2005

The trick is not to make remote operations look like local operations but to make local operation look like remote operations. – Gary Sager 1979.

Messages

The τ (tau) Interprocess Communication sends small, fixed sized messages from one process to another. τIPC is patterned after Demos [BASK] and Oryx/Pecos [SAGE]. A gate controls the right to send a message to a process. Links and paths are the equivalent constructs in Demos and Oryx/Pecos. Each gate has an owner and a creator. The creator creates a gate with a specific set of attributes and gets back a key for that gate. The creator then passes the key either directly or indirectly to the new owner of the key. The owner can then send messages to the creator using the key it received for the gate.



To transfer large data areas, τ uses data move operations. Data move transfer data directly from one user process to another.

message – format of messages used by τ

```
typedef u32     ki_t;
typedef struct sys_s {
        void    *q_tag;         /* User defined value */
        ki_t    q_passed_key;   /* Key passed to/from another process */
        u16     q_type;         /* Type of key received */
        u16     q_unused;       /* Number of data areas */
        void    *q_start;       /* Start of data area */
        u64     q_length;       /* Length of data area */
        u64     q_offset;       /* Offset into data area */
} sys_s;

typedef struct msg_s {
        sys_s   q;
        union {
                u8      m_method;
                u8      b_body[64];
                u64     i[8];
                u64     cr_id;
        };
} msg_s;
```

call – synchronously send a message with a reply key

Synopsis:

```
int call_tau (ki_t key, void *msg);
```

Description:

A synchronous operation, **call_tau** waits for a response to the message sent. Each thread of a process can be blocked on its own **call_tau** without affecting the other threads in the process.

change_index – change key index to a standard key index location.

Synopsis:

```
int change_index_tau (ki_t key,ki_t std_key);
```

Description:

Key indexes 1–4 are reserved. The **change_index** will change the key index to a standard index. Key index 1 is used by the switchboard libraries.

Returns:

Returns 0 on success
EINUSE – The target index is currently in use.

create_gate – create a gate to allow passing of messages to the creator

Synopsis:

```
int create_gate_tau (void *msg);
```

Description:

The process creating the gate is called the creator. The process holding the key is called the owner.

Fields that can be set:

q_tag – a user defined value, typically, a pointer to an object. This value will be in **q_tag** of any message received through this gate.

q_type – type of gate to be created.

- REPLY – One and only one message is allowed to be sent through this gate. That message may be a destruction notification.
- REQUEST – The gate can be used to send multiple messages. However, no destruction notification is generated when the key is destroyed. Used to advertise services. Can be replicated.
- RESOURCE – The gate can be used to send multiple messages. Destruction notification is generated for key destruction. Used when a resource is tied to the gate by the tag.
- PASS_REPLY – The gate allows a reply key to be passed with the message.
- PASS_OTHER – The gate allows passing either request or resource key with the message.
- READ_DATA – The gate provides read access to an area of data.
- WRITE_DATA – The gate provides write access to an area of data.

q_start – start address of a data area, only needs to be set if gate type is READ_DATA or WRITE_DATA.

q_length – length of the data area.

q_passed_key – returns the key index to the created gate.

Returns:

Returns 0 on success.

q_passed_key contains the **key** for the newly created **gate**. The key can be passed to an other processes using **send_key** to let those processes send messages to the creator.

msg.cr_id contains the id for the gate. The creator can break the gate with **destroy_gate_tau**(msg.cr_id).

Errors:

EBADGATE – the type supplied in the msg.q.q_type field is invalid.

EDATA – one or more of the addresses that are part of the data move area is invalid. It may be the ending address is out of range.

Example:

```
key_t replygate ()
{
        msg_s   msg;
        int     rc;

        msg.q.q_tag = &MyData;
        msg.q.q_type = REPLY | PASS_OTHER;
        rc = create_gate_tau( &msg);
        if (rc) {
                error(rc);
                return 0;
        }
        return msg.q.q_passedkey;
}
```

destroy_gate – destroys the specified gate.

Synopsis:

```
int destroy_gate_tau (u64 gate_id);
```

Description:

When a gate is created, a value is returned that is the gate's id, *msg.cr_id*. The creator of the gate can then destroy it and prevent the owner of the key from using it.

destroy_key – given a key, destroy the corresponding gate

Synopsis:

```
int destroy_key_tau (ki_t key);
```

Description:

Destroy the key. If it is a resource or reply gate, the creator receives a destruction notification.

duplicate_key – given a key, make a new key for accessing the same gate.

Synopsis:

```
int duplicate_key_tau (ki_t key);
```

Description:

Duplicate a REQUEST key.

ENODUP – Attempt to duplicate a REPLY or RESOURCE key.

Used to duplicate a key and pass it to another process while still keeping the original. Used extensively by the switchboard.

getdata – a synchronous call to get (read) a chunk of data.

Synopsis:

```
int getdata_tau (ki_t key, void *msg, unint length, void *start);
```

Description:

Allows another process to write **length** bytes of data starting at **start**. **getdata** uses a reply gate.

putdata – a synchronous call to put (write) a chunk of data.

Synopsis:

```
int putdata_tau (ki_t key, void *msg, unint length, const void *start);
```

Description:

Allows another process to read **length** bytes of data starting at **start**. **putdata** uses a reply gate.

readdata – reads data from another process.

Synopsis:

```
int readdata_tau (ki_t key, int length, void *data, int offset);
```

Description:

Reads data from the area specified by the gate referenced by the key starting at offset. Returns the number of bytes read.

Area data is an asynchronous version of readdata. The user passes in the channel to use for the reply gate to use to tell the process that the data move operation is completed.

receive – receive a message.

Synopsis:

```
int receive_tau (msg_s *msg);
```

Description:

If a message is waiting, returns the message otherwise block and wait for a message.

Example:

```
for (;;) {
        rc = receive_tau( &msg);
        if (rc == 0) {
                process_message( &msg);
        } else if (rc == DESTROYED) {
                cleanup_resources( &msg);
        } else {
                fprintf(stderr, “Error=%d\n”, rc);
        }
}
```

send – send a message to the creator of the key

Synopsis:

```
int send_tau (ki_t key, msg_s *msg);
```

Description:

Send the body of the message from the owner of the key, the current process, to the creator of the gate corresponding to the key. If the key is for a REPLY gate, the key is removed from the owner’s key table and the index becomes free. If the receiving thread is blocked, send will schedule the process to be run, otherwise the message is queued. At the receiving end, q_tag is set from the gate and the q_passed_key is set to zero.

Example:

send_key – pass a key with a message to the creator of the key

Synopsis:

```
int send_key_tau (ki_t key, msg_s *msg);
```

Description:

Sends the gate specified by the passed key along with the message.

Example:

writedata – writes data to another process.

Synopsis:

```
int writedata_tau (ki_t key, void *data, int length, int offset);
```

Description:

Writes data to the area specified by the gate referenced by the key starting at offset. Returns the number of bytes written.

Simple Example

Given: The Client has a Request Gate to the Server, a simple file server.

```
ki_t    File_system_key;
char    Buf[4096];

Client (char *file_name)
{
        Message_s       msg;
        key_t           file_key;
        int             rc;

        msg.m_method = OPEN_REQEST;
        rc = putdata_tau(File_system_key, &msg, strlen(file_name), file_name);
        if (rc) return rc;
        file_key = msg.q.q_passedkey;

        msg.m_method = READ_REQUEST;
        rc = getdata_tau(file_key, &msg, sizeof(Buf), Buf);
        if (rc) {
                destroykey_tau(file_key);
                return rc;
        }
        destroykey_tau(file_key);
        return 0;
}

Server ()
{
        Message_s       msg;
        int             rc;

        for (;;) {
                rc = receive( &msg);
                if (rc == Gate_DESTROYED) {
                        close_file(msg.q.q_tag);
                        continue;
                }
                if (rc) {
                        fprintf(stderr, “Error=%d\n”, rc);
                        continue;
                }
                replykey = msg.q.q_passedkey;
                switch (msg.m_method) {
                        case OPEN_FILE:
                                readdata_tau(replykey, msg.q_length, name, 0);
                                file = open_file(name);
                                msg.q.q_tag = file;
                                msg.q.q_type = RESOURCE | PASS_REPLY;
                                rc = creategate_tau( &msg);
                                if (rc) {
                                        fprintf(stderr, “creategate failed=%d\n”, rc);
                                        destroykey_tau(replykey);
                                }
                                rc = send_key_tau(replykey, &msg);
                                if (rc) {
                                        fprintf(stderr, “sendkey failed=%d\n”, rc);
                                        destroykey_tau(msg.q.q_passedkey);
                                        destorykey_tau(replykey);
                                }
                                continue;
                        case READ_FILE:
                                buf = readfile(msg.q.q_tag);
                                rc = writedata_tau(replykey, msg.q_length, buf, 0);
                                if (rc) {
                                        fprintf(stderr, “writedata failed=%d\n”, rc);
                                        release(buf);
                                        destroykey_tau(replykey);
                                        continue;
                                }
                                rc = send_tau(replykey, &msg);
                                if (rc) {
                                        fprintf(stderr, “send failed=%d\n”, rc);
                                        destroykey_tau(replykey);
                                }
                                continue;
                        case WRITE_FILE:
                                .
                                .
                                .
                        default:
                                fprintf(stderr, “unknown request %d\n”, msg.b.i[0]);
                                destroykey_tau(replykey);
                                continue;
                }
        }
}
```

Bibliography

[BASK] Baskett, Forest, John Howard, and John Motague, *Task Communication in DEMOS*, Proceedings of Sixth ACM Symposium on Operating Principles, Nov. 77, 23-31.

[SAGE] Sager, G. R., J. A. Melber, and K. T. Fong, The Oryx/Pecos Operating System, AT&T Technical Journal, Vol. 64, No. 1 Jan. 85, 251–268.