readme

1. rpcgen -a task.x

task.h (header file to be included)
task_xdr.c (XDR: External Data Representation, to be used by client/server stub)
task_clnt.c (client stub)
task_svc.c (server stub)

task_client.c (client program template)
task_server.c (server program template)

2. implement task_server.c and task_client.c

3. compile
gcc task_server.c task_svc.c task_xdr.c -o server -lnsl
gcc task_client.c task_clnt.c task_xdr.c -o client -lnsl

4. Excecute:

on goliath:
server &

on another machine:
client goliath

```
const MAX=10000;


struct programarg{
       char file[MAX];
       int length;
};

struct taskarg{
       char file[MAX];
       int length;
       int id;
};


program EXECTASK {
version VERSION {

        int         SET_TASK(programarg)=1;
        taskarg     EXEC_NEXT_TASK()=2;

        }=1;
}=9999;
```

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _TASK_H_RPCGEN
#define     _TASK_H_RPCGEN

#include <rpc/rpc.h>
#define     MAX 10000

struct programarg {
      char file[MAX];
      int length;
};
typedef struct programarg programarg;

struct taskarg {
      char file[MAX];
      int length;
      int id;
};
typedef struct taskarg taskarg;

#define     EXECTASK    9999
#define     VERSION     1
#define     SET_TASK    1
extern  int * set_task_1();
#define     EXEC_NEXT_TASK    2
extern  taskarg * exec_next_task_1();
extern int exectask_1_freeresult();

/* the xdr functions */
extern bool_t xdr_programarg();
extern bool_t xdr_taskarg();

#endif /* !_TASK_H_RPCGEN */
```

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "task.h"

bool_t
xdr_programarg(xdrs, objp)
      register XDR *xdrs;
      programarg *objp;
{

#if defined(_LP64) || defined(_KERNEL)
      register int *buf;
#else
      register long *buf;
#endif

      int i;
      if (!xdr_vector(xdrs, (char *)objp->file, MAX,
            sizeof (char), (xdrproc_t) xdr_char))
            return (FALSE);
      if (!xdr_int(xdrs, &objp->length))
            return (FALSE);
      return (TRUE);
}

bool_t
xdr_taskarg(xdrs, objp)

      register XDR *xdrs;
      taskarg *objp;
{

#if defined(_LP64) || defined(_KERNEL)
      register int *buf;
#else
      register long *buf;
#endif

      int i;
      if (!xdr_vector(xdrs, (char *)objp->file, MAX,
            sizeof (char), (xdrproc_t) xdr_char))
            return (FALSE);
      if (!xdr_int(xdrs, &objp->length))
            return (FALSE);
      if (!xdr_int(xdrs, &objp->id))
            return (FALSE);
      return (TRUE);
}
```

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "task.h"
#ifndef _KERNEL
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#endif /* !_KERNEL */

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
set_task_1(argp, clnt)
      programarg *argp;
      CLIENT *clnt;
{
      static int clnt_res;

      memset((char *)&clnt_res, 0, sizeof (clnt_res));
      if (clnt_call(clnt, SET_TASK,
            (xdrproc_t) xdr_programarg, (caddr_t) argp,
            (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
            TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
      }
      return (&clnt_res);
}

taskarg *
exec_next_task_1(argp, clnt)
      void *argp;
      CLIENT *clnt;
{
      static taskarg clnt_res;

      memset((char *)&clnt_res, 0, sizeof (clnt_res));
      if (clnt_call(clnt, EXEC_NEXT_TASK,
            (xdrproc_t) xdr_void, (caddr_t) argp,
            (xdrproc_t) xdr_taskarg, (caddr_t) &clnt_res,
            TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
      }
      return (&clnt_res);
}
```

```
1   /*
2    * Please do not edit this file.
3    * It was generated using rpcgen.
4    */
5
6   #include "task.h"
7   #include <stdio.h>
8   #include <stdlib.h> /* getenv, exit */
9   #include <signal.h>
10  #include <sys/types.h>
11  #include <memory.h>
12  #include <stropts.h>
13  #include <netconfig.h>
14  #include <sys/resource.h> /* rlimit */
15  #include <syslog.h>
16
17  #ifdef DEBUG
18  #define     RPC_SVC_FG
19  #endif
20
21  #define     _RPCSVC_CLOSEDOWN 120
22  static int _rpcpmstart;       /* Started by a port monitor ? */
23
24  /* States a server can be in wrt request */
25
26  #define     _IDLE 0
27  #define     _SERVED 1
28
29  static int _rpcsvcstate = _IDLE;    /* Set when a request is serviced */
30  static int _rpcsvccount = 0;        /* Number of requests being serviced */
31
32  static
33  void _msgout(msg)
34        char *msg;
35  {
36  #ifdef RPC_SVC_FG
37        if (_rpcpmstart)
38              syslog(LOG_ERR, msg);
39        else
40              (void) fprintf(stderr, "%s\n", msg);
41  #else
42        syslog(LOG_ERR, msg);
43  #endif
44  }
45
46  static void
47  closedown(sig)
48        int sig;
49  {
50        if (_rpcsvcstate == _IDLE && _rpcsvccount == 0) {
51              int size;
52              int i, openfd = 0;
53
54              size = svc_max_pollfd;
55              for (i = 0; i < size && openfd < 2; i++)
56                    if (svc_pollfd[i].fd >= 0)
```

```
57                          openfd++;
58              if (openfd <= 1)
59                    exit(0);
60        } else
61              _rpcsvcstate = _IDLE;
62
63        (void) signal(SIGALRM, (void(*)()) closedown);
64        (void) alarm(_RPCSVC_CLOSEDOWN/2);
65  }
66
67  static void
68  exectask_1(rqstp, transp)
69        struct svc_req *rqstp;
70        register SVCXPRT *transp;
71  {
72        union {
73              programarg set_task_1_arg;
74        } argument;
75        char *result;
76        bool_t (*_xdr_argument)(), (*_xdr_result)();
77        char *(*local)();
78
79        _rpcsvccount++;
80        switch (rqstp->rq_proc) {
81        case NULLPROC:
82              (void) svc_sendreply(transp, xdr_void,
83                    (char *)NULL);
84              _rpcsvccount--;
85              _rpcsvcstate = _SERVED;
86              return;
87
88        case SET_TASK:
89              _xdr_argument = xdr_programarg;
90              _xdr_result = xdr_int;
91              local = (char *(*)()) set_task_1;
92              break;
93
94        case EXEC_NEXT_TASK:
95              _xdr_argument = xdr_void;
96              _xdr_result = xdr_taskarg;
97              local = (char *(*)()) exec_next_task_1;
98              break;
99
100       default:
101             svcerr_noproc(transp);
102             _rpcsvccount--;
103             _rpcsvcstate = _SERVED;
104             return;
105       }
106       (void) memset((char *)&argument, 0, sizeof (argument));
107       if (!svc_getargs(transp, _xdr_argument, (caddr_t) &argument)) {
108             svcerr_decode(transp);
109             _rpcsvccount--;
110             _rpcsvcstate = _SERVED;
111             return;
112       }
113       result = (*local)(&argument, rqstp);
```

```
114         if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
115               svcerr_systemerr(transp);
116         }
117         if (!svc_freeargs(transp, _xdr_argument, (caddr_t) &argument)) {
118               _msgout("unable to free arguments");
119               exit(1);
120         }
121         _rpcsvccount--;
122         _rpcsvcstate = _SERVED;
123         return;
124   }
125
126   main()
127   {
128         pid_t pid;
129         int i;
130
131         (void) sigset(SIGPIPE, SIG_IGN);
132
133         /*
134          * If stdin looks like a TLI endpoint, we assume
135          * that we were started by a port monitor. If
136          * t_getstate fails with TBADF, this is not a
137          * TLI endpoint.
138          */
139         if (t_getstate(0) != -1 || t_errno != TBADF) {
140               char *netid;
141               struct netconfig *nconf = NULL;
142               SVCXPRT *transp;
143               int pmclose;
144
145               _rpcpmstart = 1;
146               openlog("task", LOG_PID, LOG_DAEMON);
147
148               if ((netid = getenv("NLSPROVIDER")) == NULL) {
149               /* started from inetd */
150                     pmclose = 1;
151               } else {
152                     if ((nconf = getnetconfigent(netid)) == NULL)
153                           _msgout("cannot get transport info");
154
155                     pmclose = (t_getstate(0) != T_DATAXFER);
156               }
157               if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
158                     _msgout("cannot create server handle");
159                     exit(1);
160               }
161               if (nconf)
162                     freenetconfigent(nconf);
163               if (!svc_reg(transp, EXECTASK, VERSION, exectask_1, 0)) {
164                     _msgout("unable to register (EXECTASK, VERSION).");
165                     exit(1);
166               }
167               if (pmclose) {
168                     (void) signal(SIGALRM, (void(*)()) closedown);
169                     (void) alarm(_RPCSVC_CLOSEDOWN/2);
170               }
```

```
171               svc_run();
172               exit(1);
173               /* NOTREACHED */
174         }     else {
175   #ifndef RPC_SVC_FG
176               int size;
177               struct rlimit rl;
178               pid = fork();
179               if (pid < 0) {
180                     perror("cannot fork");
181                     exit(1);
182               }
183               if (pid)
184                     exit(0);
185               rl.rlim_max = 0;
186               getrlimit(RLIMIT_NOFILE, &rl);
187               if ((size = rl.rlim_max) == 0)
188                     exit(1);
189               for (i = 0; i < size; i++)
190                     (void) close(i);
191               i = open("/dev/null", 2);
192               (void) dup2(i, 1);
193               (void) dup2(i, 2);
194               setsid();
195               openlog("task", LOG_PID, LOG_DAEMON);
196   #endif
197         }
198         if (!svc_create(exectask_1, EXECTASK, VERSION, "netpath")) {
199               _msgout("unable to create (EXECTASK, VERSION) for netpath.");
200               exit(1);
201         }
202
203         svc_run();
204         _msgout("svc_run returned");
205         exit(1);
206         /* NOTREACHED */
207   }
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
```

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "task.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */


void
exectask_1(host)
      char *host;
{
      CLIENT *clnt;
      int  *result_1;
      programarg  set_task_1_arg;
      taskarg  *result_2;
      char *  exec_next_task_1_arg;

#ifndef     DEBUG
      clnt = clnt_create(host, EXECTASK, VERSION, "netpath");
      if (clnt == (CLIENT *) NULL) {
            clnt_pcreateerror(host);
            exit(1);
      }
#endif      /* DEBUG */

      result_1 = set_task_1(&set_task_1_arg, clnt);
      if (result_1 == (int *) NULL) {
            clnt_perror(clnt, "call failed");
      }
      result_2 = exec_next_task_1((void *)&exec_next_task_1_arg, clnt);
      if (result_2 == (taskarg *) NULL) {
            clnt_perror(clnt, "call failed");
      }
#ifndef     DEBUG
      clnt_destroy(clnt);
#endif            /* DEBUG */
}


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

      if (argc < 2) {
            printf("usage:  %s server_host\n", argv[0]);
            exit(1);
      }
      host = argv[1];
      exectask_1(host);
}
```

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "task.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>

int *
set_task_1(argp, rqstp)
      programarg *argp;
      struct svc_req *rqstp;
{
      static int  result;

      /*
       * insert server code here
       */

      return (&result);
}

taskarg *
exec_next_task_1(argp, rqstp)
      void *argp;
      struct svc_req *rqstp;
{
      static taskarg  result;

      /*
       * insert server code here
       */

      return (&result);
}
```