Last modification: 09-Apr-2018
Author: R. Koucha

Make system() less cumbersome and more efficient

Table of Contents

1. Introduction

Several applications need to call shell commands and/or scripts of shell commands. To make it, the C library provides the **system**() service which is passed as parameter the command line to run:

```
int system(const char *command);
```

The “command” parameter may be a simple executable name or a more complex shell command line using output redirections and pipes.

system() hides a call to “/bin/sh -c” to run the command line passed as parameter.

From Linux system point of view, in the simplest case, **system**() triggers at least two pairs of **fork**()/**exec**() system calls: one for “sh -c” and another for the command line itself as depicted in Figure 1.

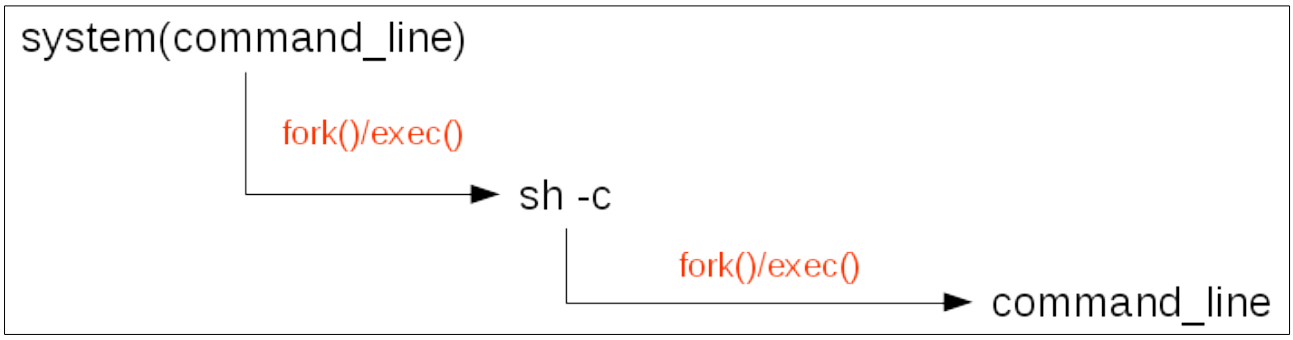

Figure 1: system() internals

Moreover, **fork**() triggers a duplication of some resources (memory, file descriptors...) of the calling process (the father) to make the forked process (the child) inherit them. Compared to original Unix implementations, Linux benefited multiple enhancements like the Copy On Write (i.e. COW) to make the **fork**() more efficient and less cumbersome. But some applications use, not to say overuse **system**() and sometimes passing to it complex command lines (i.e. with multiple commands, pipes and redirections). This may lead to a memory over consumption which triggers Linux defense mechanisms like Out Of Memory (**OOM**) killer.

In embedded environments, the cost of the hardware is an important consideration. As a consequence, the memory is often very limited. The memory as well as the CPU time are critical resources which must be used with care and as efficiently as possible not only for response time and robustness purposes but also for hardware cost reduction purposes.

This paper aims at addressing the problem of **system**() overuse with some alternate solutions to enhance existing applications in a confident way that is to say with a minimal impact on the existing source code.

2. Use of vfork()

Linux inherited **vfork**() from BSD as an optimization of the application using **fork()** directly followed by a call to **exec()**. The idea was to avoid copying to much resources from the parent process when the child address space is immediately replaced by a new program. Compared to **fork()**, **vfork()** creates a new process without copying the page tables of the parent process. It is useful in performance-sensitive applications. As **system**() is merely a call to **fork**() immediately followed by a call to **exec**(), it is possible to rewrite it by replacing the call to **fork**() by **vfork**(). § A.1 shows a source code proposal for this solution. It is inspired by the original **system**() source code in the **GLIBC** library.

Such a solution makes **system**() slightly more efficient: it is exactly the same behaviour except that **vfork**() is supposed to be more efficient than **fork**(). As a consequence, we save memory demand and CPU time.

It is possible to go farther by eliminating the step which consists to run and terminate a shell (i.e. “sh -c”) in order to save more CPU time as described in the following paragraph.

3. Remanent shell

As some applications need to call **system**() very often, it means that “sh -c” is run very often. Moreover, the execution and termination of multiple shells by several concurrent applications sucks CPU time and memory resources. It is possible to plan a solution where a shell is executed once and stays ready to use in any application needing to run commands.

The idea consists to start one (or more ?) background shell(s) at application startup. We don’t use the “-c” option which runs one command line and then makes the shell exit. The shell must lives in background during the application lifetime even after command execution. Each time the application needs to run a command, it submits it to the background shell. This saves the CPU time and memory needed to start and stop the shell. Figure 2 depicts the principle.

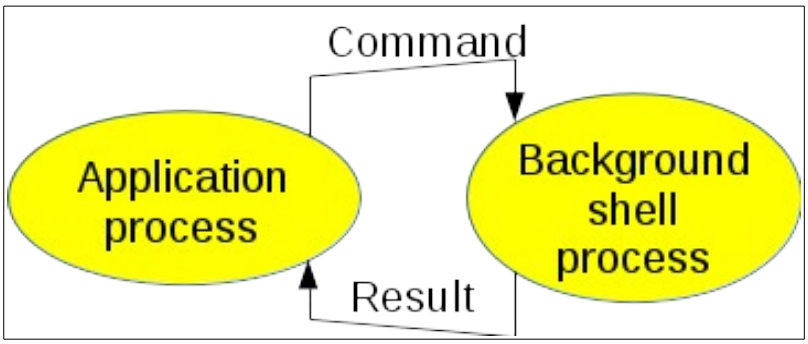

Figure 2: Background shell

Without “-c” option, the shell is interactive. In other words, it needs to be in front of a terminal. Linux provides the pseudo-terminal (i.e. **PTY**) concept to manage this kind of needs. The **PTY** is

setup between the application process (master side) and the background shell process (slave side). The latter believes that it is interacting with an operator through a real terminal whereas the operator is nothing else than the application process: cf. Figure 3.

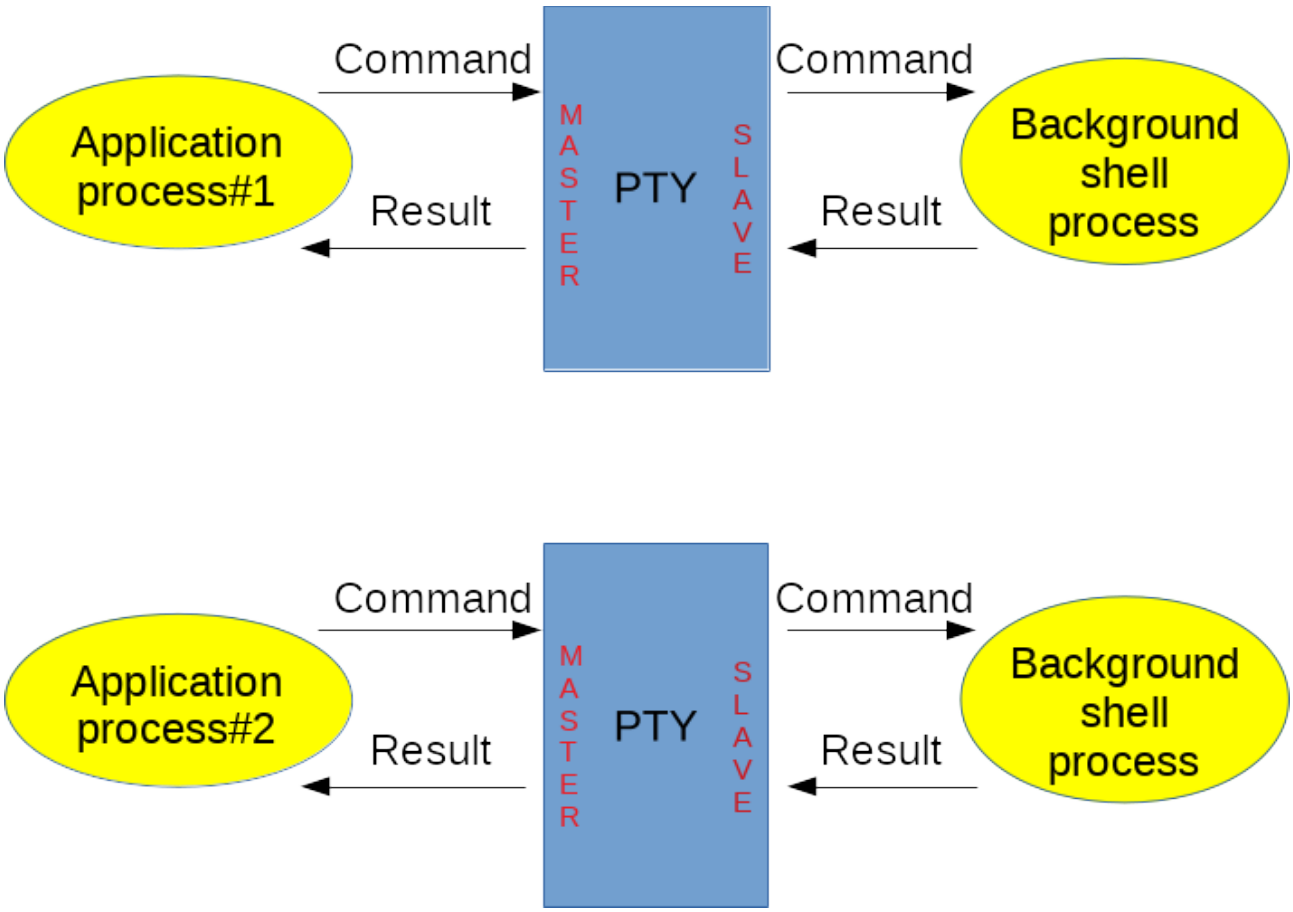

Figure 3: Pseudo-terminal

As the shell is in interactive mode, it displays a prompt to wait for a command. It gets the command, executes it and displays a new prompt at the end of the command to wait for another one. By default, the application process would need to do some tricky work to parse the displays from the shell and get the result of the command. To make it simple, it is possible to use **PDIP** (i.e. Programmed Dialogs with Interactive Programs). This is an open source (https://sourceforge.net/projects/pdip/). The package is fully documented with online manuals, html pages (http://pdip.sourceforge.net/) and examples. It is an expect-like[1] tool but much more simple to use which provides the ability to pilot interactive programs. It comes in two flavors: a command named **pdip** which is used to control interactive programs from a shell script and an C language API offered by a shared library called **libpdip.so** to control interactive programs from a C/C++ language program. The latter is interesting to implement the current solution.

1 **Expect** is a famous tool very well known by testers as it permits to automate interactive programs: https://en.wikipedia.org/wiki/Expect

In the source tree, the **isys** sub-directory contains a variant of **system**() using the above principle (cf. **isys.c** embedded in a shared library called **libisys.so**). § A.2 presents some details about this library.

With **libisys.so,** the application process calls an API named **isystem**() which behaves the same as **system**() but actually it hides the **PTY** and the running background shell described above (cf. Figure 4).

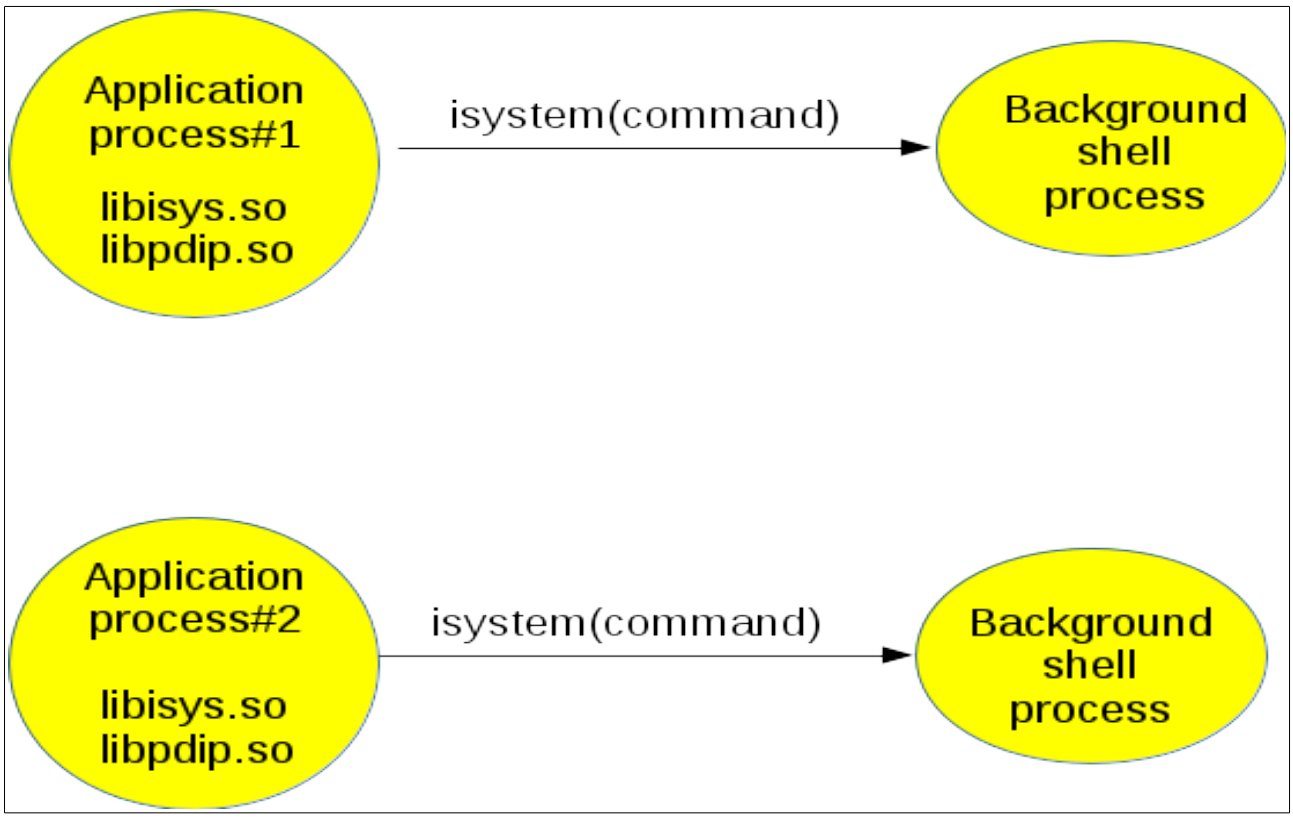

Figure 4: Use of PDIP library

The solution described in this chapter saves the **fork**()/**exec**() of "sh -c" by keeping at least one running background shell per application process. Depending on the application's behaviour, it may be useful to keep at least a running shell. But it may be cumbersome from a memory point of view if the application calls to **isystem**() are rare. It is possible to enhance this implementation to reduce the number of running background shells by sharing them with all the running applications as proposed in the following chapter.

4. Remote shell

To go farther in the preceding implementation, we propose to share running shells with all the application processes. The principle consists to setup a daemon process managing one or more background shells (static configuration or dynamic setting on demand for example). Let's call it **rsystemd** (i.e. rsystem daemon) to comply with Unix naming scheme. It is started before any application (at system startup for example) and waits for commands to run on a named socket. It

submits the command to the shells that it manages and reports the result to the originating application processes. To make it, **rsystemd** relies on **libpdip.so** to interact with the shells as explained in § 3. On application process side, an API named **rsystem**() behaves the same as **system**() but actually it hides the interaction with **systemd** through the named socket: the command line passed as argument is written into the socket to make **rsystemd** run it and return the displays and the command status. The principle is depicted in Figure 5.

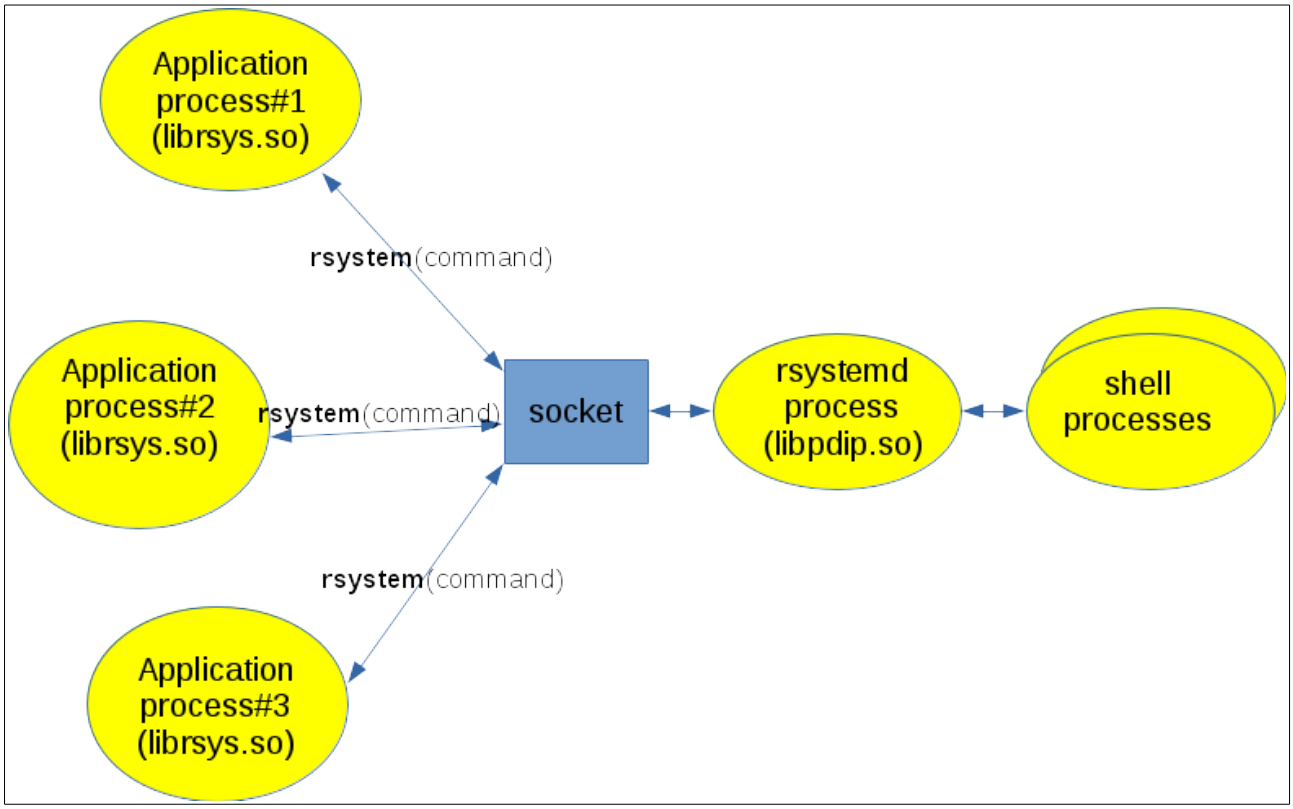

Figure 5: rsystemd

In the source tree of the **PDIP** package, the **rsys** sub-directory contains a variant of **system**() using the above principle (cf. **rsystem.c** embedded in a shared library called **librsys.so** which implements **rsystem()** API and **rsystemd.c** which implements the daemon part). § A.3 presents some details about this library.

This proposal not only saves CPU time as we do not continuously **fork**()/**exec**() and terminate shell processes but it also saves memory space as the running shells are shared with several processes. By the way, we must not forget that this solution differs from original **system**() service from a user interface point of view: as the shells are running in separate processes which are not childs of the application processes: they are childs of **rsystemd.** As a consequence, the father to child inheritance does not operate here (file descriptors, environment variables, signal disposition …). But most of the time it is not required by the users of **system**().

5. Conclusion

This paper presented various ideas to replace calls to **system**() by new simple APIs hiding mechanisms to save memory space and CPU time. This may contribute to the application robustness and efficiency. Especially at system startup time where several initialization shell scripts are triggered to setup the execution environment (server startup, log creation, network configuration, kernel modules loading…).

ANNEXES

A.1. System() based on vfork()

```
int vsystem(
             const char *format,
             va_list     ap
            )
{

int               rc;
char              cmd[SH_CMD_LEN];
char             *av[4];
int               status;
struct sigaction  action, act_quit, act_int;
sigset_t          old_mask;
int               sav_errno = errno;
int               sig_is_blocked = 0;

  // Build the command line to pass to "sh -c"

  // According to the manual, if the command is NULL, a non 0 value
  // is returned if a shell is available or 0 value is returned if a
  // shell is not available
  // For example a shell might not be available after a chroot()
  if (!format)
  {
    // The trick used in GLIBC is to execute "exit 0" command
    rc = snprintf(cmd, sizeof(cmd), "exit 0");
  }
  else
  {
    rc = vsnprintf(cmd, sizeof(cmd), format, ap);
  } // End if !format

  // Check if the command is not too long
  if (rc >= sizeof(cmd))
  {
    errno = ENOSPC;
    return -1;
  }

  // Make the parameters
  av[0] = SH_NAME;
  av[1] = "-c";
  av[2] = cmd;
  av[3] = (char *)0;

  // According to the manual, the service must block SIGCHLD and ignore
  // SIGINT and SIGQUIT

  action.sa_handler = SIG_IGN;
  action.sa_flags = 0;
  sigemptyset(&(action.sa_mask));

  // The reference counter ensures that we will not get in act_int and
  // act_quit, actions inherited from concurrent calls of systemX() which
  // would set IGNORE forever...
  SYST2_LOCK();
```

```
// If the signal are not already ignored
if (0 == (syst2_ref ++))
{
  if (0 != sigaction(SIGINT, &action, &act_int))
  {
    sav_errno = errno;
    syst2_ref --;
    status = -1;
    goto out;
  } // End if sigaction !OK

  if (0 != sigaction(SIGQUIT, &action, &act_quit))
  {
    sav_errno = errno;
    syst2_ref --;
    status = -1;
    goto out_restore_sigint;
  } // End if sigaction !OK
} // End if not already ignored (former ref_count != 0)

SYST2_UNLOCK();

// For some reasons, GLIBC's system() blocks SIGCHLD outside of the critical
// section...
sigaddset(&(action.sa_mask), SIGCHLD);
if (0 != pthread_sigmask(SIG_BLOCK, &(action.sa_mask), &old_mask))
{
  sav_errno = errno;

  SYST2_LOCK();

  if (0 == (-- syst2_ref))
  {
    // Restore the former handler for SIGQUIT
    (void)sigaction(SIGQUIT, &act_quit, 0);

    // Restore the former handler for SIGINT
    (void)sigaction(SIGINT, &act_int, 0);
  }

  SYST2_UNLOCK();

  errno = sav_errno;

  return -1;
}
else
{
  sig_is_blocked = 1;
} // End if pthread_sigmask() failed

// Use the light fork
rc = vfork();
switch(rc)
{
  case -1 : // Error
  {
    sav_errno = errno;
    status = -1;
  }
  break;
```

```
    case 0 : // Child process
    {
    extern char **environ;

      // Restore the former handlers for SIGQUIT, SIGINT and signal mask
      (void)sigaction(SIGQUIT, &act_quit, 0);
      (void)sigaction(SIGINT, &act_int, 0);
      (void)pthread_sigmask(SIG_SETMASK, &old_mask, 0);

      (void)execve(SH_PATH, av, environ);
      _exit(127);
    }
    break;

    default: // Father process
    {
    pid_t pid;

      // Wait for the termination of the child
      pid = waitpid(rc, &status, 0);
      if (pid != rc)
      {
        status = -1;
      }
    }
    break;
  } // End switch

  SYST2_LOCK();

  if (0 == (-- syst2_ref))
  {

out:

    // Restore the former handler for SIGQUIT
    if (sigaction(SIGQUIT, &act_quit, 0) != 0)
    {
      sav_errno = errno;
      status = -1;
    }

out_restore_sigint:

    // Restore the former handler for SIGINT
    if (sigaction(SIGINT, &act_int, 0) != 0)
    {
      sav_errno = errno;
      status = -1;
    }
  } // End if ref_count is 0

  // Restore the signal mask (For some reasons the GLIBC restores the
  // mask under the LOCK where as the blocking was done outside
  // the lock)
  if (sig_is_blocked)
  {
    if (pthread_sigmask(SIG_SETMASK, &old_mask, 0) != 0)
    {
      sav_errno = errno;
```

```
      status = -1;
    }
  }

  SYST2_UNLOCK();

  errno = sav_errno;

  return status;
} // vsystem
```

A.2. isys library

A.2.1. Build from the sources

Unpack the source code package:

```
$ tar xvfz pdip-2.1.0.tgz
```

Go into the top level directory of the sources and trigger the build of the DEB packages:

```
$ cd  pdip-2.1.0
$ ./pdip_install -P DEB
```

A.2.2. Installation from the packages

ISYS depends on **PDIP**. So, **PDIP** must be installed prior to install **ISYS** otherwise you get the following error:

```
$ sudo dpkg -i isys_2.1.0_amd64.deb
Selecting previously unselected package isys.
(Reading database ... 218983 files and directories currently installed.)
Preparing to unpack isys_2.1.0_amd64.deb ...
Unpacking isys (2.1.0) ...
dpkg: dependency problems prevent configuration of isys:
isys depends on pdip (>= 2.0.4); however:
 Package pdip is not installed.

dpkg: error processing package isys (--install):
dependency problems - leaving unconfigured
Errors were encountered while processing:
isys
```

Install first the **PDIP** package:

```
$ sudo dpkg -i pdip_2.1.0_amd64.deb
Selecting previously unselected package pdip.
(Reading database ... 218988 files and directories currently installed.)
Preparing to unpack pdip_2.1.0_amd64.deb ...
Unpacking pdip (2.1.0) ...
Setting up pdip (2.1.0) ...
Processing triggers for man-db (2.7.5-1) ...
```

Then install the **ISYS** package:

```
$ sudo dpkg -i isys_2.1.0_amd64.deb
(Reading database ... 219040 files and directories currently installed.)
Preparing to unpack isys_2.1.0_amd64.deb ...
Unpacking isys (2.1.0) over (2.1.0) ...
Setting up isys (2.1.0) …
```

Installation from the packages is the preferred way as it is easy to get rid of the software with all the cleanups by calling:

```
$ sudo dpkg -r isys
(Reading database ... 219043 files and directories currently installed.)
Removing isys (2.1.0) …
```

To display the list of files installed by the package:

```
$ dpkg -L isys
/.
/usr
/usr/local
/usr/local/include
/usr/local/include/isys.h
/usr/local/lib
/usr/local/lib/libisys.so
/usr/local/share
/usr/local/share/man
/usr/local/share/man/man3
/usr/local/share/man/man3/isys.3.gz
/usr/local/share/man/man3/isystem.3.gz
```

A.2.3. Installation from cmake

It is also possible to trigger the installation from cmake:

```
$ tar xvfz pdip-2.1.0.tgz
$ cd  pdip-2.1.0
$ cmake .
-- The C compiler identification is GNU 6.2.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Building PDIP version 2.1.0
The user id is 1000

-- Configuring done
-- Generating done
-- Build files have been written to: /home/rachid/DEVS/PDIP
$ sudo make install
Scanning dependencies of target man
[  2%] Building pdip_en.1.gz
[  4%] Building pdip_fr.1.gz
[  6%] Building pdip_configure.
[…]
-- Installing: /usr/local/lib/librsys.so
-- Installing: /usr/local/sbin/rsystemd
-- Set runtime path of "/usr/local/sbin/rsystemd" to ""
```

A.2.4. Manual

When the **ISYS** package is installed, on line manuals are available in section 3 (API).

```
$ man 3 isystem
NAME
      isys - Interactive system() service
```

SYNOPSIS

```
#include "isys.h"

int isystem(const char *fmt, ...);

int isys_lib_initialize(void);
```

DESCRIPTION

The **ISYS** API provides a **system**(3)-like service based on a remanent background shell to save memory and CPU time in applications where **system**(3) is heavily used.

isystem() executes the shell command line formatted with fmt. The behaviour of the format is compliant with **printf**(3). Internally, the command is run by a remanent shell created by the **libisys.so** library in a child of the current process.

isys_lib_initialize() is to be called in child processes using the **ISYS** API. By default, **ISYS** API is deactivated upon **fork**(2).

ENVIRONMENT VARIABLE

The **ISYS_TIMEOUT** environment variable specifies the maximum time in seconds to wait for data from the shell (by default, it is 10 seconds).

RETURN VALUE

isystem() returns the status of the executed command line (i.e. the last executed command). The returned value is a "wait status" that can be examined using the macros described in **waitpid**(2) (i.e. WIFEXITED(), WEXITSTATUS(), and so on).

isys_lib_initialize() returns 0 when there are no error or -1 upon error (**errno** is set).

MUTUAL EXCLUSION

The service does not support concurrent calls to **isystem**() by multiple threads. If this behaviour is needed, the application is responsible to manage the mutual exclusion on its side.

EXAMPLE

The following program receives a shell command as argument and executes it via a call to **isystem**().

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <libgen.h>
#include <stdlib.h>
#include <string.h>
#include <isys.h>

int main(int ac, char *av[])
{
int     status;
int     i;
char   *cmdline;
size_t  len;
size_t  offset;

  if (ac < 2)
```

```
        {
          fprintf(stderr, "Usage: %s cmd params...\n", basename(av[0]));
          return 1;
        }

        // Build the command line
        cmdline = (char *)0;
        len     = 1; // Terminating NUL
        offset = 0;
        for (i = 1; i < ac; i ++)
        {
          len += strlen(av[i]) + 1; // word + space
          cmdline = (char *)realloc(cmdline, len);
         assert(cmdline);
          offset += sprintf(cmdline + offset, "%s ", av[i]);
        } // End for

        printf("Running '%s'...\n", cmdline);

        status = isystem(cmdline);
        if (status != 0)
        {
          printf("Error from program (0x%x)!\n", status);
          free(cmdline);
          return  1;
        } // End if

        free(cmdline);
        return 0;
      } // main

      Build the program:

      $ gcc tisys.c -o tisys -lisys -lpdip -lpthread

      Then, run something like the following:

      $ ./tisys echo example
      Running 'echo example '...
      example

AUTHOR
      Rachid Koucha

SEE ALSO
      system(3).
```

A.2.5. Build facilities

To help people to auto-detect the location of **ISYS** stuff (libraries, include files…), the **ISYS** package installs a configuration file named **isys.pc** to make it available for **pkg-config** tool. Moreover, for **cmake** based packages, a **FindIsys.cmake** file is provided at the top level of **PDIP** source tree.

A.3. rsys library

A.3.1. Build from the sources

Unpack the source code package:

```
$ tar xvfz pdip-2.1.0.tgz
```

Go into the top level directory of the sources and trigger the build of the DEB packages:

```
$ cd  pdip-2.1.0
$ ./pdip_install -P DEB
```

A.3.2. Installation from the packages

RSYS depends on **PDIP**. So, **PDIP** must be installed prior to install **RSYS** otherwise you get the following error:

```
$ sudo dpkg -i rsys_2.1.0_amd64.deb
Selecting previously unselected package rsys.
(Reading database ... 218983 files and directories currently installed.)
Preparing to unpack rsys_2.1.0_amd64.deb ...
Unpacking rsys (2.1.0) ...
dpkg: dependency problems prevent configuration of rsys:
rsys depends on pdip (>= 2.0.4); however:
 Package pdip is not installed.

dpkg: error processing package rsys (--install):
dependency problems - leaving unconfigured
Errors were encountered while processing:
rsys
```

Install first the **PDIP** package:

```
$ sudo dpkg -i pdip_2.1.0_amd64.deb
Selecting previously unselected package pdip.
(Reading database ... 218988 files and directories currently installed.)
Preparing to unpack pdip_2.1.0_amd64.deb ...
Unpacking pdip (2.1.0) ...
Setting up pdip (2.1.0) ...
Processing triggers for man-db (2.7.5-1) ...
```

Then install the **RSYS** package:

```
$ sudo dpkg -i rsys_2.1.0_amd64.deb
(Reading database ... 219042 files and directories currently installed.)
Preparing to unpack rsys_2.1.0_amd64.deb ...
Unpacking rsys (2.1.0) over (2.1.0) ...
Setting up rsys (2.1.0) …
```

Installation from the packages is the preferred way as it is easy to get rid of the software with all the cleanups by calling:

```
$ sudo dpkg -r rsys
(Reading database ... 219043 files and directories currently installed.)
Removing rsys (2.1.0) …
```

To display the list of files installed by the package:

```
$ dpkg -L rsys
/.
/usr
/usr/local
/usr/local/include
/usr/local/include/rsys.h
/usr/local/lib
/usr/local/lib/librsys.so
/usr/local/sbin
/usr/local/sbin/rsystemd
/usr/local/share
/usr/local/share/man
/usr/local/share/man/man3
/usr/local/share/man/man3/rsys.3.gz
/usr/local/share/man/man3/rsystem.3.gz
/usr/local/share/man/man8
/usr/local/share/man/man8/rsystemd.8.gz
```

A.3.3. Installation from cmake

It is also possible to trigger the installation from cmake:

```
$ tar xvfz pdip-2.1.0.tgz
$ cd  pdip-2.1.0
$ cmake .
-- The C compiler identification is GNU 6.2.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Building PDIP version 2.1.0
The user id is 1000

-- Configuring done
-- Generating done
-- Build files have been written to: /home/rachid/DEVS/PDIP
$ sudo make install
Scanning dependencies of target man
[  2%] Building pdip_en.1.gz
[  4%] Building pdip_fr.1.gz
[  6%] Building pdip_configure.
[…]
-- Installing: /usr/local/lib/librsys.so
-- Installing: /usr/local/sbin/rsystemd
-- Set runtime path of "/usr/local/sbin/rsystemd" to ""
```

A.3.4. Manual

When the **RSYS** package is installed, on line manuals are available in sections 3 (API) and 8 (rsystemd daemon).

A.3.4.1. rsystem()

```
$ man 3 rsystem
NAME
       rsys - Remote system() service

SYNOPSIS
       #include "rsys.h"

       int rsystem(const char *fmt, ...);

       int rsys_lib_initialize(void);

DESCRIPTION
       The RSYS API provides a system(3)-like service based on shared remanent
background shells managed by rsystemd(8) daemon. This saves memory and CPU time
in applications where system(3) is heavily used.

       rsystem() executes the shell command line formatted with fmt.  The
behaviour of the format is compliant with printf(3).  Internally, the command is
run by one of the remanent shells managed by rsystemd(8).

       rsys_lib_initialize() is to be called in child processes using the RSYS
API. By default, RSYS API is deactivated upon fork(2).

ENVIRONMENT VARIABLE
       By default, the server socket pathname used for the client/server dialog
is /var/run/rsys.socket.  The RSYS_SOCKET_PATH environment variable is available
to specify an alternate socket pathname if one needs to change it for access
rights or any test purposes.

RETURN VALUE
       rsystem() returns the status of the executed command line (i.e. the last
executed command). The returned value is a "wait status" that can be examined
using the macros described in waitpid(2)  (i.e.  WIFEX-
       ITED(), WEXITSTATUS(), and so on).

       rsys_lib_initialize() returns 0 when there are no error or -1 upon error
(errno is set).

MUTUAL EXCLUSION
       The service does not support concurrent calls to rsystem() by multiple
threads.  If this behaviour is needed, the application is responsible to manage
the mutual exclusion on its side.

EXAMPLE
       The following program receives a shell command as argument and executes it
via a call to rsystem().

       #include <stdio.h>
       #include <assert.h>
       #include <stdlib.h>
       #include <libgen.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <rsys.h>

int main(int ac, char *av[])
{
int           status;
int           i;
char         *cmdline;
size_t        len;
size_t        offset;

  if (ac < 2)
  {
    fprintf(stderr, "Usage: %s cmd params...\n", basename(av[0]));
    return 1;
  }

  // Build the command line
  cmdline = 0;
  len     = 1; // Terminating NUL
  offset = 0;
  for (i = 1; i < ac; i ++)
  {
    len += strlen(av[i]) + 1; // word + space
    cmdline = (char *)realloc(cmdline, len);
    assert(cmdline);
    offset += sprintf(cmdline + offset, "%s ", av[i]);
  } // End for

  printf("Running '%s'...\n", cmdline);

  status = rsystem(cmdline);
  if (status != 0)
  {
    fprintf(stderr, "Error from program (0x%x = %d)!\n", status, status);
    free(cmdline);
    return(1);
  } // End if

  free(cmdline);

  return(0);
} // main
```

Build the program:

```
$ gcc trsys.c -o trsys -lrsys -lpdip -lpthread
```

Make sure that **rsystemd**(8) is running. Then, run something like the following:

```
$ ./trsys echo example
Running 'echo example '...
example
```

AUTHOR

Rachid Koucha

SEE ALSO

rsystemd(8), **system**(3).

A.3.4.2. rsystemd

```
$ man 8 rsystemd
NAME
      rsystemd - Remote system() daemon

SYNOPSIS
      rsystemd [-s shells] [-V] [-d level] [-D] [-h]

DESCRIPTION
      rsystemd is a daemon which manages several childs processes running
shells. It is a server for the rsystem(3) service.

OPTIONS
      -s | --shells shell_list
             Shells  to  launch along with their CPU affinity. This may be
overriden by the RSYSD_SHELLS environment variable. The content is a colon
delimited list of affinities for shells to launch. An affinity is defined as
follow:

                   * A comma separated list of fields

                   * A field is either a CPU number or an interval of
consecutive CPU numbers described with the first and last CPU numbers separated
by an hyphen.

                   * An empty field implicitely means all the active CPUs

                   * A CPU number is from 0 to the number of active CPUs minus 1


                   * If the first CPU number of an interval is empty, it is
considered to be CPU number 0

                   * If the last CPU number of an interval is empty, it is
considered to be the biggest active CPU number

             If a CPU number is bigger than the maximum active CPU number, it is
implicitely translated into the maximum active CPU number.

             If this option is not specified, the default behaviour is one shell
running on all available CPUs.

      -V | --version
             Display the daemon's version

      -D | --daemon
             Activate the daemon mode (the process detaches itself from the
current terminal and becomes a child of init, process number 1).


      -d | --debug level
             Set the debug level. The higher the value, the more traces are
displayed.

      -h | --help
```

```
            Display the help

ENVIRONMENT VARIABLE
      By default, the server socket pathname used for the client/server dialog
is /var/run/rsys.socket.  The RSYS_SOCKET_PATH environment variable is available
to specify an alternate socket pathname if one needs to change it for access
rights or any test purposes. It is advised to specify an absolute pathname
especially in daemon mode where the server changes its current directory to the
root of the filesystem. Consequently, any relative pathname will be considered
from the server's current directory.

EXAMPLES
      The following launches a shell running on CPU number 3 and CPU numbers 6
to 8. We use sudo as rsystemd creates a named socket in /var/run.

        $ sudo systemd -s 3,6-8

      The  following  launches  three shells. The first runs on CPU numbers 0 to
3, CPU number 5 and CPU number 6. The second runs on CPU number 0 and CPU
numbers 3 to the latest active CPU. The third runs on all
      the active CPUs.

        $ sudo systemd -s -3,5,6:0,3-:

      The following launches one shell through the RSYSD_SHELLS environment
variable. We pass -E option to sudo to preserve the environment otherwise
RSYSD_SHELLS would not be taken in  account.  The  environment
      variable overrides the parameter passed to rsystemd.  The affinity of the
shell are CPU number 1 and 3.

        $ export RSYSD_SHELLS=1,3
        $ sudo -E systemd -s -3,5,6:0,3-:

AUTHOR
      Rachid Koucha

SEE ALSO
      rsystem(3).
```

A.3.5. FSM of rsystemd

The finished state machine describing the main engine of rsystemd is depicted in Figure 6.

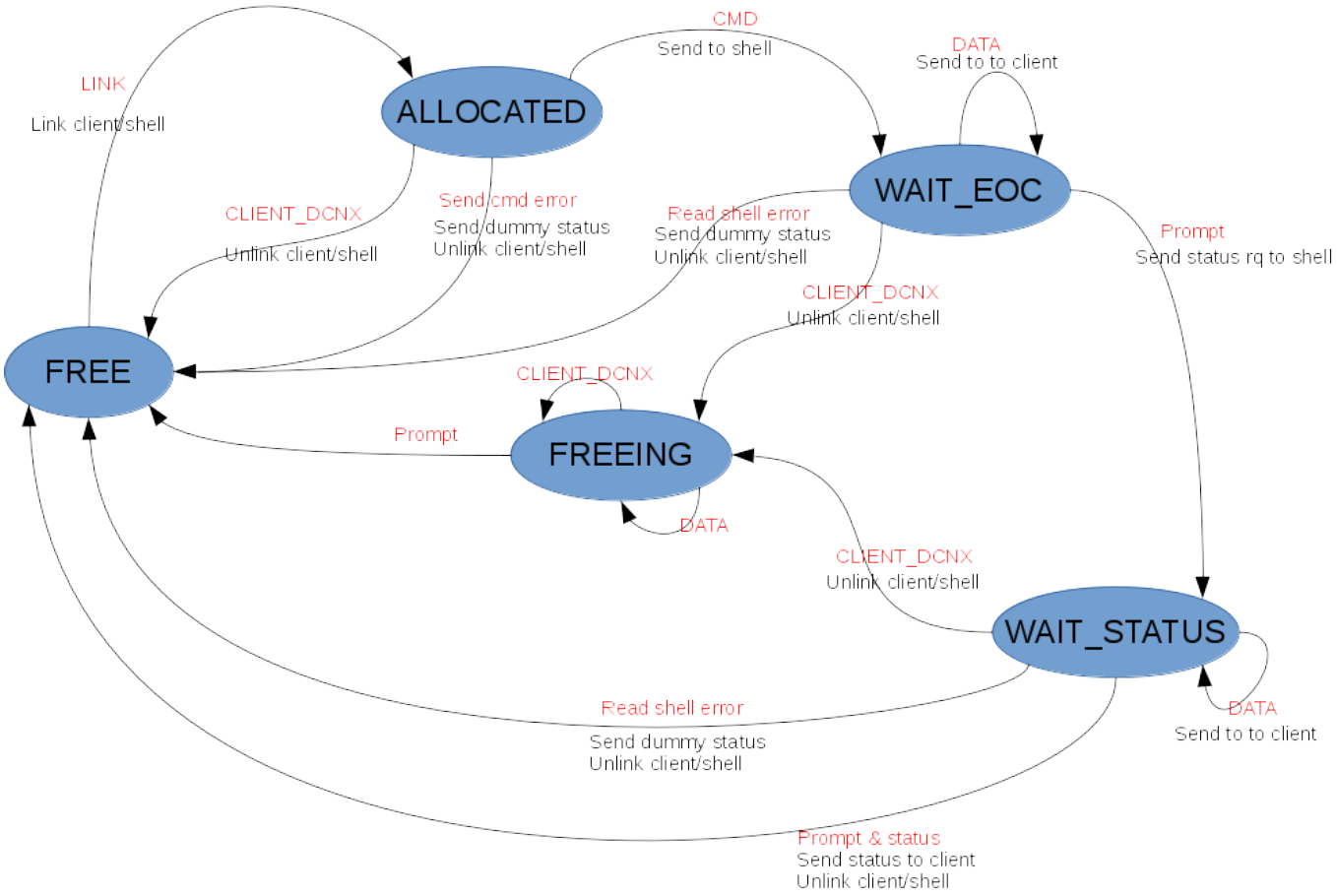

Figure 6: FSM of rsystemd

A.3.6. Build facilities

To help people to auto-detect the location of **RSYS** stuff (libraries, include files…), the **RSYS** package installs a configuration file named **rsys.pc** to make it available for **pkg-config** tool. Moreover, for **cmake** based packages, a **FindRsys.cmake** file is provided at the top level of **PDIP** source tree.