SWI-Prolog C-library

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: jan@swi.psy.uva.nl

March 3, 2000

Abstract

This document describes commonly used foreign language extensions to SWI-Prolog distributed as a package known under the name *clib*

Contents

1 Introduction

Many useful facilities offered by one or more of the operating systems supported by SWI-Prolog are not supported by the SWI-Prolog kernel distribution. Including these would enlarge the *footprint* and complicate portability matters while supporting only a limited part of the user-community.

This document describes library(`unix`) to deal with the Unix process API, library(`socket`) to deal with inet-domain stream-sockets, library(`cgi`) to deal with getting CGI form-data if SWI-Prolog is used as a CGI scripting language and library(`crypt`) to provide access to Unix password encryption.

2 Unix Process manipulation library

The library(`unix`) library provides the commonly used Unix primitives to deal with process management. These primitives are useful for many tasks, including server management, parallel computation, exploiting and controlling other processes, etc.

The predicatea are modelled closely after their native Unix counterparts. Higher-level primitives, especially to make this library portable to non-Unix systems are desirable. Using these primitives and considering that process manipulation is not a very time-critical operation we anticipate these libraries to be developed in Prolog.

fork(-*Pid*)

Clone the current process into two branches. In the child, *Pid* is unified to `child`. In the original process, *Pid* is unified to the process identifier of the created child. Both parent and child are fully functional Prolog processes running the same program. The processes share open I/O streams that refer to Unix native streams, such as files, sockets and pipes. Data is not shared, though on most Unix systems data is initially shared and duplicated only if one of the programs attempts to modify the data.

Unix `fork()` is the only way to create new processes and `fork/2` is a simple direct interface to it.

exec(+*Command(...Args...)*)

Replace the running program by starting *Command* using the given commandline arguments. Each command-line argument must be atomic and is converted to a string before passed to the Unix call `execvp()`.

Unix `exec()` is the only way to start an executable file executing. It is commonly used together with `fork/1`. For example to start `netscape` on an URL in the background, do:

```
run_netscape(URL) :-
        (    fork(child),
             exec(netscape(URL))
        ;    true
        ).
```

Using this code, netscape remains part of the process-group of the invoking Prolog process and Prolog does not wait for netscape to terminate. The predicate `wait/2` allows waiting for a child, while `detach_IO/0` disconnects the child as a deamon process.

wait(*-Pid, -Status*)

Wait for a child to change status. Then report the child that changed status as well as the reason. *Status* is unified with `exited`(*ExitCode*) if the child with pid *Pid* was terminated by calling `exit()` (Prolog `halt/[0,1]`). *ExitCode* is the return=status. *Status* is unified with `signaled`(*Signal*) if the child died due to a software interrupt (see `kill/2`). *Signal* contains the signal number. Finally, if the process suspended execution due to a signal, *Status* is unified with `stopped`(*Signal*).

kill(*+Pid, +Signal*)

Deliver a software interrupt to the process with identifier *Pid* using software-interrupt number *Signal*. See also `on_signal/2`. The meaning of the signal numbers can be found in the Unix manual.[1].

pipe(*-InSream, -OutStream*)

Create a communication-pipe. This is normally used to make a child communicate to its parent. After `pipe/2`, the process is cloned and, depending on the desired direction, both processes close the end of the pipe they do not use. Then they use the remaining stream to communicate. Here is a simple example:

```
pipe_demo(Result) :-
        pipe(Read, Write),
        fork(Pid),
        (   Pid == child
        ->  close(Read),
            format(Write, '~w.~n', [hello(world)]),
            halt
        ;   close(Write),
            read(Read, Result),
            close(Read)
        ).
```

dup(*+FromStream, +ToStream*)

Interface to Unix dup2(), copying the underlying filedescriptor and thus making both streams point to the same underlying object. This is normally used together with `fork/1` and `pipe/2` to talk to an external program that is designed to communicate using standard I/O.

detach_IO

This predicate is intended to create Unix deamon-processes. It preforms two actions. First of all, the I/O streams `user_input`, `user_output` and `user_error` are closed and rebound to a Prolog stream that returns end-of-file on any attempt to read and starts writing to a file named `/tmp/pl-out.\bnfmeta{pid}` on any attempt to write. This file is opened only if there is data available. This is intended for debugging purposes.[2] Finally, the process is detached from the current process-group and its controlling terminal.

[1] `kill/2` should support interrupt-names as well

[2] More subtle handling of I/O, especially for debugging is required: communicate with the syslog deamon and optionally start a debugging dialog on a newly created (X-)terminal should be considered.

3 Socket library

The library(()socket) library provides TCP inet-domain sockets from SWI-Prolog, both client and server-side communication. The interface of this library is very close to the Unix socket interface, also supported by the MS-Windows *winsock* API.

In the future we hope to provide a more high-level socket interface defined in Prolog and based on these primitives.

tcp_socket(*-SocketId***)**
Creates an `INET`-domain stream-socket and unifies an identifier to it with *SocketId*. On MS-Windows, if the socket library is not yet initialised, this will also initialise the library.

tcp_close_socket(*+SocketId***)**
Closes the indicated socket, making *SocketId* invalid. Normally, sockets are closed by closing both stream handles returned by `open_socket/3`. There are two cases where `close_socket/1` is used because there are no stream-handles:

- After `tcp_accept/3`, the server does a `fork/1` to handle the client in a sub-process. In this case the accepted socket is not longer needed from the main server and must be discarded using `tcp_close_socket/1`.
- If, after discovering the connecting client with `tcp_accept/3`, the server does not want to accept the connection, it should discard the accepted socket immediately using `tcp_close_socket/1`.

tcp_open_socket(*+SocketId, -InStream, -OutStream***)**
Open two SWI-Prolog I/O-streams, one to deal with input from the socket and one with output to the socket. If `tcp_bind/2` has been called on the socket. *OutSream* is useless and will not be created. After closing both *InStream* and *OutSream*, the socket itself is discarded.

tcp_bind(*+Socket, +Port***)**
Bind the socket to *Port* on the current machine. This operation, together with `tcp_listen/2` and `tcp_accept/3` implement the *server*-side of the socket interface.

tcp_listen(*+Socket, +Backlog***)**
Tells, after `tcp_bind/2`, the socket to listen for incomming requests for connections. *Backlog* indicates how many pending connection requests are allowed. Pending requests are requests that are not yet acknowledged using `tcp_accept/3`. If the indicated number is exceeded, the requesting client will be signalled that the service is currently not available. A suggested default value is 5.

tcp_accept(*+Socket, -Slave, -Peer***)**
This predicate waits on a server socket for a connection request by a client. On success, it creates a new socket for the client and binds the identifier to *Slave*. *Peer* is bound to the IP-address of the client.

tcp_connect(*C***)**
lient-interface to connect a socket to a given *Port* on a given *Host*. After successfull completion, `tcp_open_socket/3` can be used to create I/O-Streams to the remote socket.

tcp_fcntl(*+Stream, +Action, ?Argument*)

Interface to the Unix `fcntl()` call. Currently only suitable to deal switch stream to non-blocking mode using:

```
...
tcp_fcntlStream, setfl. nonblock),
...
```

As of SWI-Prolog 3.2.4, handling of non-blocking stream is supported. An attempt to read from a non-blocking stream returns -1 (or `end_of_file` for `read/1`), but `at_end_of_stream/1` fails. On actual end-of-input, `at_end_of_stream/1` succeeds.

tcp_host_to_address(*?HostName, ?Address*)

Translate between a machines host-name and it's (IP-)address. If *HostName* is an atom, it is resolved using `gethostbyname()` and the IP-number is unified to *Address* using a term of the format `ip`(*Byte1, Byte2, Byte3, Byte4*). Otherwise, if *Address* is bound to a `ip/4` term, it is resolved by `gethostbyaddr()` and the canonical hostname is unified with *HostName*.

3.1 Server applications

The typical sequence for generating a server application is defined below:

```
create_server(Port) :-
        tcp_socket(Socket),
        tcp_bind(Socket, Port),
        tcp_listen(Socket, 5),
        tcp_open_socket(Socket, AcceptFd, _),
        <dispatch>
```

There are various options for ⟨*dispatch*⟩. One is to keep track of active clients and server-sockets using `wait_for_input/3`. If input arrives at a server socket, use `tcp_accept/3` and add the new connection to the active clients. Otherwise deal with the input from the client. Another is to use (Unix) `fork/1` to deal with the client in a separate process.

Using `fork/1`, ⟨*dispatch*⟩ may be implemented as:

```
dispatch(AcceptFd) :-
        tcp_accept(AcceptFd, Socket, _Peer),
        fork(Pid)
        (   Pid == child
        ->  tcp_open_socket(Socket, In, Out),
            handle_service(In, Out),
            close(In),
            close(Out),
            halt
        ;   close_socket(Socket)
        ),
        dispatch(AcceptFd).
```

3.2 Client applications

The skeleton for client-communication is given below.

```
create_client(Host, Port) :-
        tcp_socket(Socket),
        tcp_connect(Socket, Host:Port),
        tcp_open_socket(Socket, ReadFd, WriteFd),
        <handle I/O using the two streams>
        close(ReadFd),
        close(WriteFd).
```

To deal with timeouts and multiple connections, `wait_for_input/3` and/or non-blocking streams (see `tcp_fcntl/3`) can be used.

4 CGI Support library

This is currently a very simple library, providing support for obtaining the form-data for a CGI script:

cgi_get_form(-*Form*)
: Decodes standard input and the environment variables to obtain a list of arguments passed to the CGI script. This predicate both deals with the CGI **GET** method as well as the **POST** method. If the data cannot be obtained, an `existence_error` exception is raised.

Below is a very simple CGI script that prints the passed parameters. To test it, compile this program using the command below, copy it to your cgi-bin directory (or make it otherwise known as a CGI-script) and make the query `http://myhost.mydomain/cgi-bin/cgidemo?hello=world`

```
% pl -o cgidemo --goal=main --toplevel=halt -c cgidemo.pl


:- use_module(library(cgi)).

main :-
        cgi_get_form(Arguments),
        format('Content-type: text/html~n~n', []),
        format('<HTML>~n', []),
        format('<HEAD>~n', []),
        format('<TITLE>Simple SWI-Prolog CGI script</TITLE>~n', []),
        format('</HEAD>~n~n', []),
        format('<BODY>~n', []),
        format('<P>', []),
        print_args(Arguments),
        format('<BODY>~n</HTML>~n', []).

print_args([]).
```

```
print_args([A0|T]) :-
        A0 =.. [Name, Value],
        format('<B>~w</B>=<EM>~w</EM><BR>~n', [Name, Value]),
        print_args(T).
```

4.1 Some considerations

Printing an HTML document using `format/2` is not really a neat way of producing HTML. We are considering two alternatives. One is to define a library that provides for macro-expansion as well as keeping track of open environments. This makes the code more robust and allows for writing generic infrastructure to finish a page in a neat way should anything go wrong during its construction. We have used this in some internal projects with a certain degree of success. The above code would look like this:

```
main :-
        cgi_get_form(Arguments),
        #open_document('text/html'),
        #head([#title('Simple SWI-Prolog CGI script output')]),
        ...
```

Recently, we are considering to represent the HTML document as a complex Prolog term and provide a library for decent printing thereof.

5 Unix password encryption library

The library(`crypt`) library defines `crypt/2` for encrypting and testing Unix passwords:

crypt(*+Plain, ?Encrypted***)**

This predicate can be used in three modes. If *Encrypted* is unbound, it will be unified to a string (list of character-codes) holding a random encryption of *Plain*. If *Encrypted* is bound to a list holding 2 characters and an unbound tail, these two character are used for the *salt* of the encryption. Finally, if *Encrypted* is instantiated to an encrypted password the predicate succeeds iff *Encrypted* is a valid encryption of *Plain*.

Plain is either an atom, SWI-Prolog string, list of characters or list of character-codes. It is not adviced to use atoms, as this implies the password will be available from the Prolog heap as a defined atom.

6 Installation

6.1 Unix systems

Installation on Unix system uses the commonly found *configure*, *make* and *make install* sequence. SWI-Prolog should be installed before building this package. If SWI-Prolog is not installed as `pl`, the environment variable `PL` must be set to the name of the SWI-Prolog executable. Installation is now accomplished using:

```
% ./configure
% make
% make install
```

This installs the foreign libraries in `\$PLBASE/lib/\$PLARCH` and the Prolog library files in `\$PLBASE/library`, where `\$PLBASE` refers to the SWI-Prolog 'home-directory'.