Netpbm subroutine library: pm_system() subroutine

pm_system()

Updated: 26 August 2006

Name

pm_system - run a Netpbm program with program input and output

Synopsis

```
#include <netpbm/pm_system.h>
pm_system(void                stdinFeeder(int, void *),
        void *        const feederParm,
        void              stdoutAccepter(int, void *),
        void *        const accepterParm,
        const char *  const shellCommand);
```

Example

This simple example converts a PNM image on Standard Input to a JFIF (JPEG) image on Standard Output. In this case, **pm_system()** is doing no more than **system()** would do.

```
  pm_system(NULL, NULL, NULL, NULL, "pnmtojpeg");
```

This example does the same thing, but moves the data through memory buffers to illustrate use with memory buffers, and we throw in a stage to shrink the image too:

```
#include <netpbm/pm_system.h>
char            pnmData[100*1024];   /* Input file better be < 100K */
char            jfifData[100*1024];
struct bufferDesc pnmBuffer;
struct bufferDesc jfifBuffer;
unsigned int    jfifSize;
pnmBuffer.size = fread(pnmData, 1, sizeof(pnmData), stdin);
pnmBuffer.buffer = pnmData;
pnmBuffer.bytesTransferredP = NULL;
jfifBuffer.size = sizeof(jfifData);
jfifBuffer.buffer = jfifData;
jfifBuffer.bytesTransferredP = &jfifSize;
pm_system(&pm_feed_from_memory, &pnmBuffer,
        &pm_accept_to_memory, &jfifBuffer,
        "pamscale .5 | pnmtojpeg");
fwrite(jfifData, 1, jfifSize, stdout);
```

This example reads an image into libnetpbm PAM structures, then brightens it, then writes it out, to illustrate use of **pm_system** with PAM structures.

```
#include <netpbm/pam.h>
#include <netpbm/pm_system.h>
struct pam       inpam;
struct pam       outpam;
tuples **        inTuples;
tuples **        outTuples;
struct pamtuples inPamtuples;
struct pamtuples outPamtuples;
inTuples = pnm_readpam(stdin, &inpam, sizeof(inpam));
outpam = inpam;
inPamtuples.pamP = &inpam;
inPamtuples.tuplesP = &inTuples;
outPamtuples.pamP = &outpam;
outPamtuples.tuplesP = &outTuples;
pm_system(&pm_feed_from_pamtuples, &inPamtuples,
        &pm_accept_to_pamtuples, &outPamtuples,
        "ppmbrighten -v 100");
outpam.file = stdout;
pnm_writepam(&outpam, outTuples);
```

DESCRIPTION

This library function is part of Netpbm.

pm_system() is a lot like the standard C library **system()** subroutine. It runs a shell and has that shell execute a shell command that you specify. But **pm_system()** gives you more control over the Standard Input and Standard Output of that shell command than **system()**. **system()** passes to the shell command as Standard Input and Output whatever is the Standard Input and Output of the process that calls **system()**. But with **pm_system()**, you specify as arguments subroutines to execute to generate the shell command's Standard Input stream and to process the shell command's Standard Output stream.

Your Standard Input feeder subroutine can generate the stream in limitless ways. **pm_system()** gives it a file descriptor of a pipe to which to write the stream it generates. **pm_system()** hooks up the other end of that pipe to the shell command's Standard Input.

Likewise, your Standard Output accepter subroutine can do anything it wants with the stream it gets. **pm_system()** gives it a file descriptor of a pipe from which to read the stream. **pm_system()** hooks up the other end of that pipe to the shell command's Standard Output.

The argument *stdinFeeder* is a function pointer that identifies your Standard Input feeder subroutine. **pm_system()** runs it in a child process and waits for that process to terminate (and accepts its completion status) before returning. *feederParm* is the argument that **pm_system()** passes to the subroutine; it is opaque to **pm_system()**.

If you pass *stdinFeeder* = NULL, **pm_system()** simply passes your current Standard Input stream to the shell command (as **system()** would do), and does not create a child process.

The argument *stdoutAccepter* is a function pointer that identifies your Standard Output accepter subroutine. **pm_system()** calls it in the current process. *accepterParm* is an argument analogous to *feederParm*.

If you pass *stdoutAccepter* = NULL, **pm_system()** simply passes your current Standard Output stream to the shell command (as **system()** would do.

The argument *shellCommand* is a null-terminated string containing the shell command that the shell is to execute. It can be any command that means something to the shell and can take a pipe for Standard Input and Output. Example:

```
ppmbrighten -v 100 | pamdepth 255 | pamscale .5
```

pm_system() creates a child process to run the shell and waits for that process to terminate (and accepts its

completion status) before returning.

Applications

The point of **pm_system()** is to allow you write a C program that uses other programs internally, as a shell script would. This is particularly desirable with Netpbm, because Netpbm consists of a lot of programs that perform basic graphic manipulations and you'd like to be able to build a program that does a more sophisticated graphic manipulation by calling the more basic Netpbm programs. These building block programs typically take input from Standard Input and write output to Standard Output.

The obvious alternative is to use a higher level language -- Bourne Shell or Perl, for example. But often you want your program to do manipulations of your graphical data that are easier and more efficient in C. Or you want to use the Netpbm subroutine library in your program. The Netpbm subroutine library is a C-linkage library; the subroutines in it are not usable from a Bourne Shell or Perl program.

A typical use of **pm_system()** is to place the contents of some graphical image file in memory, run a Netpbm program against it, and have what would ordinarily go into an output file in memory too, for further processing. To do that, you can use the memory buffer Standard Input feeder and Standard Output accepter described below.

If your program uses the Netpbm subroutine library to read, write, and manipulate images, you may have an image in an array of PAM tuples. If you want to manipulate that image with a Netpbm program (perhaps remap the colors using **pnmremap**), you can use the pamtuple Standard Input feeder and Standard Output acceptor described below.

Broken Pipe Behavior

When you set up a shell command to take input from a pipe, as you do with **pm_system()**, you need to understand how pipes work with respect to the programs at either end of the pipe agreeing to how much data is to be transferred. Here are some notes on that.

It is normal to read a pipe before the process on the other end has written the data you hope to read, and it is normal to write to a pipe before the process on the other end has tried to read your data. Writes to a pipe can be buffered until the reading end requests the data. A process reading or writing a pipe can block until the other end is ready. Or a read or write can complete with an indication that the other end is not ready at the moment and therefore no data, or less data than was requested, was transferred.

The pipe is normally controlled by the writing end. When you read from a pipe, you keep reading until the program on the other end of the pipe closes it, and then you get an end-of-file indication. You then normally close the reading end of the pipe, since it is no longer useful.

When you close the reading end of a pipe before getting the end-of-file indication and the writer subsequently tries to write to the pipe, that is an error condition for the writer. In a typical default Unix environment, that error causes the writer to receive a SIGPIP signal and that signal causes the writer process to terminate abnormally. But if, alternatively, the writer has ordered that SIGPIPE be blocked, ignored, or handled, the signal does not cause the death of the writer. Instead, the write operation simply completes with an error indication.

Standard Feeders And Acceptors

You can supply anything you like as a Standard Input feeder or Standard Output acceptor, but the Netpbm subroutine library comes with a few that perform commonly needed functions.

Memory Buffer

These routines are for when you just want to treat an area of memory as a file. If the shell command would ordinarily read a 513 byte regular file from its Standard Input, you want it to take 513 bytes from a certain address in your process' memory. Whatever bytes the shell command wants to write to its output file you want it to store at another address in your process' memory.

The Standard Input feeder for this is called **pm_feed_from_memory**. The Standard Output accepter is

pm_accept_to_memory.

For both of these, the argument is the address of a **struct bufferDesc**, which is defined as follows:

```
struct bufferDesc {
  unsigned int    size;
  unsigned char * buffer;
  unsigned int *  bytesTransferredP;
};
```

size is the size of the memory buffer and *buffer* is its location in memory (address). The Standard Input feeder will attempt to feed the entire buffer to the shell command's Standard Input; the Standard Output accepter will not accept any more data from the shell command's Standard Output than will fit in the buffer. Both return the actual amount of data read or written, in bytes, at the location identified by *bytesTransferredP*. Unless **bytesTransferredP** is NULL.

Because a process typically terminates abnormally when it is not able to write everything to a pipe that it wanted to, *bytesTransferredP* is not usually useful in the Standard Input feeder case.

Pamtuple

These routines are for when you have images in memory in the data structures used by the PAM family of subroutines in the Netpbm library -- i.e. struct PAM and an array of struct tuple. With these routines, you can run a Netpbm program against such an image just as you would against the same image in a regular file.

The Standard Input feeder for this is called **pm_feed_from_pamtuples**. The Standard Output accepter is **pm_accept_to_pamtuples**.

For both of these, the argument is the address of a **struct pamtuples**, which is defined as follows:

```
struct pamtuples {
  struct pam * pamP;
  tuple ***    tuplesP;
};
```

For the Standard Input feeder, you supply a struct pam, valid up through the *tuple_type* member (except it doesn't matter what the *file* member is) and array of tuples.

For the Standard Output Accepter, you supply only space in memory for the struct pam and the address of the tuple array. The routine fills in the struct pam up through the *tuple_type* member (except leaves the *file* member undefined) and allocates space for the tuple array with malloc(). You are responsible for freeing that memory.

HISTORY

pm_system() was introduced in Netpbm 10.13 (January 2003).