NAME

PCRE - Perl-compatible regular expressions

PCRE JUST-IN-TIME COMPILER SUPPORT

Just-in-time compiling is a heavyweight optimization that can greatly speed up pattern matching. However, it comes at the cost of extra processing before the match is performed. Therefore, it is of most benefit when the same pattern is going to be matched many times. This does not necessarily mean many calls of a matching function; if the pattern is not anchored, matching attempts may take place many times at various positions in the subject, even for a single call. Therefore, if the subject string is very long, it may still pay to use JIT for one-off matches.

JIT support applies only to the traditional Perl-compatible matching function. It does not apply when the DFA matching function is being used. The code for this support was written by Zoltan Herczeg.

8-BIT, 16-BIT AND 32-BIT SUPPORT

JIT support is available for all of the 8-bit, 16-bit and 32-bit PCRE libraries. To keep this documentation simple, only the 8-bit interface is described in what follows. If you are using the 16-bit library, substitute the 16-bit functions and 16-bit structures (for example, *pcre16_jit_stack* instead of *pcre_jit_stack*). If you are using the 32-bit library, substitute the 32-bit functions and 32-bit structures (for example, *pcre32_jit_stack* instead of *pcre_jit_stack*).

AVAILABILITY OF JIT SUPPORT

JIT support is an optional feature of PCRE. The "configure" option --enable-jit (or equivalent CMake option) must be set when PCRE is built if you want to use JIT. The support is limited to the following hardware platforms:

ARM v5, v7, and Thumb2
Intel x86 32-bit and 64-bit
MIPS 32-bit
Power PC 32-bit and 64-bit
SPARC 32-bit (experimental)

If --enable-jit is set on an unsupported platform, compilation fails.

A program that is linked with PCRE 8.20 or later can tell if JIT support is available by calling **pcre_config()** with the PCRE_CONFIG_JIT option. The result is 1 when JIT is available, and 0 otherwise. However, a simple program does not need to check this in order to use JIT. The normal API is implemented in a way that falls back to the interpretive code if JIT is not available. For programs that need the best possible performance, there is also a "fast path" API that is JIT-specific.

If your program may sometimes be linked with versions of PCRE that are older than 8.20, but you want to use JIT when it is available, you can test the values of PCRE_MAJOR and PCRE_MINOR, or the existence of a JIT macro such as PCRE_CONFIG_JIT, for compile-time control of your code.

SIMPLE USE OF JIT

You have to do two things to make use of the JIT support in the simplest way:

(1) Call **pcre_study()** with the PCRE_STUDY_JIT_COMPILE option for each compiled pattern, and pass the resulting **pcre_extra** block to **pcre_exec()**.

(2) Use **pcre_free_study()** to free the **pcre_extra** block when it is no longer needed, instead of just freeing it yourself. This ensures that any JIT data is also freed.

For a program that may be linked with pre-8.20 versions of PCRE, you can insert

```
#ifndef PCRE_STUDY_JIT_COMPILE
#define PCRE_STUDY_JIT_COMPILE 0
#endif
```

so that no option is passed to **pcre_study()**, and then use something like this to free the study data:

```
#ifdef PCRE_CONFIG_JIT
  pcre_free_study(study_ptr);
#else
  pcre_free(study_ptr);
#endif
```

PCRE_STUDY_JIT_COMPILE requests the JIT compiler to generate code for complete matches. If you want to run partial matches using the PCRE_PARTIAL_HARD or PCRE_PARTIAL_SOFT options of **pcre_exec()**, you should set one or both of the following options in addition to, or instead of, PCRE_STUDY_JIT_COMPILE when you call **pcre_study()**:

```
PCRE_STUDY_JIT_PARTIAL_HARD_COMPILE
PCRE_STUDY_JIT_PARTIAL_SOFT_COMPILE
```

The JIT compiler generates different optimized code for each of the three modes (normal, soft partial, hard partial). When **pcre_exec()** is called, the appropriate code is run if it is available. Otherwise, the pattern is matched using interpretive code.

In some circumstances you may need to call additional functions. These are described in the section entitled "Controlling the JIT stack" below.

If JIT support is not available, PCRE_STUDY_JIT_COMPILE etc. are ignored, and no JIT data is created. Otherwise, the compiled pattern is passed to the JIT compiler, which turns it into machine code that executes much faster than the normal interpretive code. When **pcre_exec()** is passed a **pcre_extra** block containing a pointer to JIT code of the appropriate mode (normal or hard/soft partial), it obeys that code instead of running the interpreter. The result is identical, but the compiled JIT code runs much faster.

There are some **pcre_exec()** options that are not supported for JIT execution. There are also some pattern items that JIT cannot handle. Details are given below. In both cases, execution automatically falls back to the interpretive code. If you want to know whether JIT was actually used for a particular match, you should arrange for a JIT callback function to be set up as described in the section entitled "Controlling the JIT stack" below, even if you do not need to supply a non-default JIT stack. Such a callback function is called whenever JIT code is about to be obeyed. If the execution options are not right for JIT execution, the callback function is not obeyed.

If the JIT compiler finds an unsupported item, no JIT data is generated. You can find out if JIT execution is available after studying a pattern by calling **pcre_fullinfo()** with the PCRE_INFO_JIT option. A result of 1 means that JIT compilation was successful. A result of 0 means that JIT support is not available, or the pattern was not studied with PCRE_STUDY_JIT_COMPILE etc., or the JIT compiler was not able to handle the pattern.

Once a pattern has been studied, with or without JIT, it can be used as many times as you like for matching different subject strings.

UNSUPPORTED OPTIONS AND PATTERN ITEMS

The only **pcre_exec()** options that are supported for JIT execution are PCRE_NO_UTF8_CHECK, PCRE_NO_UTF16_CHECK, PCRE_NO_UTF32_CHECK, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NOTEMPTY_ATSTART, PCRE_PARTIAL_HARD, and PCRE_PARTIAL_SOFT.

The only unsupported pattern items are \C (match a single data unit) when running in a UTF mode, and a callout immediately before an assertion condition in a conditional group.

RETURN VALUES FROM JIT EXECUTION

When a pattern is matched using JIT execution, the return values are the same as those given by the interpretive **pcre_exec()** code, with the addition of one new error code: PCRE_ERROR_JIT_STACKLIMIT. This means that the memory used for the JIT stack was insufficient. See "Controlling the JIT stack" below for a discussion of JIT stack usage. For compatibility with the interpretive **pcre_exec()** code, no more than two-thirds of the *ovector* argument is used for passing back captured substrings.

The error code PCRE_ERROR_MATCHLIMIT is returned by the JIT code if searching a very large pattern tree goes on for too long, as it is in the same circumstance when JIT is not used, but the details of exactly what is counted are not the same. The PCRE_ERROR_RECURSIONLIMIT error code is never returned by JIT execution.

SAVING AND RESTORING COMPILED PATTERNS

The code that is generated by the JIT compiler is architecture-specific, and is also position dependent. For those reasons it cannot be saved (in a file or database) and restored later like the bytecode and other data of a compiled pattern. Saving and restoring compiled patterns is not something many people do. More detail about this facility is given in the **pcreprecompile** documentation. It should be possible to run **pcre_study()** on a saved and restored pattern, and thereby recreate the JIT data, but because JIT compilation uses significant resources, it is probably not worth doing this; you might as well recompile the original pattern.

CONTROLLING THE JIT STACK

When the compiled JIT code runs, it needs a block of memory to use as a stack. By default, it uses 32K on the machine stack. However, some large or complicated patterns need more than this. The error PCRE_ERROR_JIT_STACKLIMIT is given when there is not enough stack. Three functions are provided for managing blocks of memory for use as JIT stacks. There is further discussion about the use of JIT stacks in the section entitled "JIT stack FAQ" below.

The **pcre_jit_stack_alloc()** function creates a JIT stack. Its arguments are a starting size and a maximum size, and it returns a pointer to an opaque structure of type **pcre_jit_stack**, or NULL if there is an error. The **pcre_jit_stack_free()** function can be used to free a stack that is no longer needed. (For the technically minded: the address space is allocated by mmap or VirtualAlloc.)

JIT uses far less memory for recursion than the interpretive code, and a maximum stack size of 512K to 1M should be more than enough for any pattern.

The **pcre_assign_jit_stack()** function specifies which stack JIT code should use. Its arguments are as follows:

```
pcre_extra        *extra
pcre_jit_callback  callback
void              *data
```

The *extra* argument must be the result of studying a pattern with PCRE_STUDY_JIT_COMPILE etc. There are three cases for the values of the other two options:

(1) If *callback* is NULL and *data* is NULL, an internal 32K block
    on the machine stack is used.

(2) If *callback* is NULL and *data* is not NULL, *data* must be
    a valid JIT stack, the result of calling **pcre_jit_stack_alloc()**.

(3) If *callback* is not NULL, it must point to a function that is
    called with *data* as an argument at the start of matching, in

order to set up a JIT stack. If the return from the callback function is NULL, the internal 32K stack is used; otherwise the return value must be a valid JIT stack, the result of calling **pcre_jit_stack_alloc()**.

A callback function is obeyed whenever JIT code is about to be run; it is not obeyed when **pcre_exec()** is called with options that are incompatible for JIT execution. A callback function can therefore be used to determine whether a match operation was executed by JIT or by the interpreter.

You may safely use the same JIT stack for more than one pattern (either by assigning directly or by callback), as long as the patterns are all matched sequentially in the same thread. In a multithread application, if you do not specify a JIT stack, or if you assign or pass back NULL from a callback, that is thread-safe, because each thread has its own machine stack. However, if you assign or pass back a non-NULL JIT stack, this must be a different stack for each thread so that the application is thread-safe.

Strictly speaking, even more is allowed. You can assign the same non-NULL stack to any number of patterns as long as they are not used for matching by multiple threads at the same time. For example, you can assign the same stack to all compiled patterns, and use a global mutex in the callback to wait until the stack is available for use. However, this is an inefficient solution, and not recommended.

This is a suggestion for how a multithreaded program that needs to set up non-default JIT stacks might operate:

```
During thread initalization
  thread_local_var = pcre_jit_stack_alloc(...)

During thread exit
  pcre_jit_stack_free(thread_local_var)

Use a one-line callback function
  return thread_local_var
```

All the functions described in this section do nothing if JIT is not available, and **pcre_assign_jit_stack()** does nothing unless the **extra** argument is non-NULL and points to a **pcre_extra** block that is the result of a successful study with PCRE_STUDY_JIT_COMPILE etc.

JIT STACK FAQ

(1) Why do we need JIT stacks?

PCRE (and JIT) is a recursive, depth-first engine, so it needs a stack where the local data of the current node is pushed before checking its child nodes. Allocating real machine stack on some platforms is difficult. For example, the stack chain needs to be updated every time if we extend the stack on PowerPC. Although it is possible, its updating time overhead decreases performance. So we do the recursion in memory.

(2) Why don't we simply allocate blocks of memory with **malloc()**?

Modern operating systems have a nice feature: they can reserve an address space instead of allocating memory. We can safely allocate memory pages inside this address space, so the stack could grow without moving memory data (this is important because of pointers). Thus we can allocate 1M address space, and use only a single memory page (usually 4K) if that is enough. However, we can still grow up to 1M anytime if needed.

(3) Who "owns" a JIT stack?

The owner of the stack is the user program, not the JIT studied pattern or anything else. The user program must ensure that if a stack is used by **pcre_exec()**, (that is, it is assigned to the pattern currently running), that stack must not be used by any other threads (to avoid overwriting the same memory area). The best

practice for multithreaded programs is to allocate a stack for each thread, and return this stack through the JIT callback function.

(4) When should a JIT stack be freed?

You can free a JIT stack at any time, as long as it will not be used by **pcre_exec()** again. When you assign the stack to a pattern, only a pointer is set. There is no reference counting or any other magic. You can free the patterns and stacks in any order, anytime. Just *do not* call **pcre_exec()** with a pattern pointing to an already freed stack, as that will cause SEGFAULT. (Also, do not free a stack currently used by **pcre_exec()** in another thread). You can also replace the stack for a pattern at any time. You can even free the previous stack before assigning a replacement.

(5) Should I allocate/free a stack every time before/after calling **pcre_exec()**?

No, because this is too costly in terms of resources. However, you could implement some clever idea which release the stack if it is not used in let's say two minutes. The JIT callback can help to achieve this without keeping a list of the currently JIT studied patterns.

(6) OK, the stack is for long term memory allocation. But what happens if a pattern causes stack overflow with a stack of 1M? Is that 1M kept until the stack is freed?

Especially on embedded sytems, it might be a good idea to release memory sometimes without freeing the stack. There is no API for this at the moment. Probably a function call which returns with the currently allocated memory for any stack and another which allows releasing memory (shrinking the stack) would be a good idea if someone needs this.

(7) This is too much of a headache. Isn't there any better solution for JIT stack handling?

No, thanks to Windows. If POSIX threads were used everywhere, we could throw out this complicated API.

EXAMPLE CODE

This is a single-threaded example that specifies a JIT stack without using a callback.

```
int rc;
int ovector[30];
pcre *re;
pcre_extra *extra;
pcre_jit_stack *jit_stack;

re = pcre_compile(pattern, 0, &error, &erroffset, NULL);
/* Check for errors */
extra = pcre_study(re, PCRE_STUDY_JIT_COMPILE, &error);
jit_stack = pcre_jit_stack_alloc(32*1024, 512*1024);
/* Check for error (NULL) */
pcre_assign_jit_stack(extra, NULL, jit_stack);
rc = pcre_exec(re, extra, subject, length, 0, 0, ovector, 30);
/* Check results */
pcre_free(re);
pcre_free_study(extra);
pcre_jit_stack_free(jit_stack);
```

JIT FAST PATH API

Because the API described above falls back to interpreted execution when JIT is not available, it is convenient for programs that are written for general use in many environments. However, calling JIT via **pcre_exec()** does have a performance impact. Programs that are written for use where JIT is known to be

available, and which need the best possible performance, can instead use a "fast path" API to call JIT execution directly instead of calling **pcre_exec()** (obviously only for patterns that have been successfully studied by JIT).

The fast path function is called **pcre_jit_exec()**, and it takes exactly the same arguments as **pcre_exec()**, plus one additional argument that must point to a JIT stack. The JIT stack arrangements described above do not apply. The return values are the same as for **pcre_exec()**.

When you call **pcre_exec()**, as well as testing for invalid options, a number of other sanity checks are performed on the arguments. For example, if the subject pointer is NULL, or its length is negative, an immediate error is given. Also, unless PCRE_NO_UTF[8|16|32] is set, a UTF subject string is tested for validity. In the interests of speed, these checks do not happen on the JIT fast path, and if invalid data is passed, the result is undefined.

Bypassing the sanity checks and the **pcre_exec()** wrapping can give speedups of more than 10%.

SEE ALSO

pcreapi(3)

AUTHOR

Philip Hazel (FAQ by Zoltan Herczeg)
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 17 March 2013
Copyright (c) 1997-2013 University of Cambridge.