NAME

PCRE2 - Perl-compatible regular expressions (revised API)

PCRE2 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, so 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 even for one-off matches. JIT support is available for all of the 8-bit, 16-bit and 32-bit PCRE2 libraries.

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.

AVAILABILITY OF JIT SUPPORT

JIT support is an optional feature of PCRE2. The "configure" option --enable-jit (or equivalent CMake option) must be set when PCRE2 is built if you want to use JIT. The support is limited to the following hardware platforms:

ARM 32-bit (v5, v7, and Thumb2)
ARM 64-bit
IBM s390x 64 bit
Intel x86 32-bit and 64-bit
MIPS 32-bit and 64-bit
Power PC 32-bit and 64-bit
SPARC 32-bit

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

A program can tell if JIT support is available by calling **pcre2_config()** with the PCRE2_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 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.

SIMPLE USE OF JIT

To make use of the JIT support in the simplest way, all you have to do is to call **pcre2_jit_compile()** after successfully compiling a pattern with **pcre2_compile()**. This function has two arguments: the first is the compiled pattern pointer that was returned by **pcre2_compile()**, and the second is zero or more of

the following option bits: PCRE2_JIT_COMPLETE, PCRE2_JIT_PARTIAL_HARD, or PCRE2_JIT_PARTIAL_SOFT.

If JIT support is not available, a call to **pcre2_jit_compile()** does nothing and returns PCRE2_ERROR_JIT_BADOPTION. 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, but yields exactly the same results. The returned value from **pcre2_jit_compile()** is zero on success, or a negative error code.

There is a limit to the size of pattern that JIT supports, imposed by the size of machine stack that it uses. The exact rules are not documented because they may change at any time, in particular, when new optimizations are introduced. If a pattern is too big, a call to **pcre2_jit_compile()** returns PCRE2_ERROR_NOMEMORY.

PCRE2_JIT_COMPLETE requests the JIT compiler to generate code for complete matches. If you want to run partial matches using the PCRE2_PARTIAL_HARD or PCRE2_PARTIAL_SOFT options of **pcre2_match()**, you should set one or both of the other options as well as, or instead of PCRE2_JIT_COMPLETE. The JIT compiler generates different optimized code for each of the three modes (normal, soft partial, hard partial). When **pcre2_match()** is called, the appropriate code is run if it is available. Otherwise, the pattern is matched using interpretive code.

You can call **pcre2_jit_compile()** multiple times for the same compiled pattern. It does nothing if it has previously compiled code for any of the option bits. For example, you can call it once with PCRE2_JIT_COMPLETE and (perhaps later, when you find you need partial matching) again with PCRE2_JIT_COMPLETE and PCRE2_JIT_PARTIAL_HARD. This time it will ignore PCRE2_JIT_COMPLETE and just compile code for partial matching. If **pcre2_jit_compile()** is called with no option bits set, it immediately returns zero. This is an alternative way of testing whether JIT is available.

At present, it is not possible to free JIT compiled code except when the entire compiled pattern is freed by calling **pcre2_code_free()**.

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

There are some **pcre2_match()** options that are not supported by JIT, and there are also some pattern items that JIT cannot handle. Details are given below. In both cases, matching 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 match-time 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 matching is available after compiling a pattern by calling **pcre2_pattern_info()** with the PCRE2_INFO_JITSIZE option. A non-zero result means that JIT compilation was successful. A result of 0 means that JIT support is not available, or the pattern was not processed by **pcre2_jit_compile()**, or the JIT compiler was not able to handle the pattern.

MATCHING SUBJECTS CONTAINING INVALID UTF

When a pattern is compiled with the PCRE2_UTF option, subject strings are normally expected to be a valid sequence of UTF code units. By default, this is checked at the start of matching and an error is generated if invalid UTF is detected. The PCRE2_NO_UTF_CHECK option can be passed to **pcre2_match()** to skip the check (for improved performance) if you are sure that a subject string is valid. If this option is used with an invalid string, the result is undefined.

However, a way of running matches on strings that may contain invalid UTF sequences is available. Calling **pcre2_compile()** with the PCRE2_MATCH_INVALID_UTF option has two effects: it tells the interpreter in **pcre2_match()** to support invalid UTF, and, if **pcre2_jit_compile()** is called, the compiled JIT code also supports invalid UTF. Details of how this support works, in both the JIT and the interpretive cases, is given in the **pcre2unicode** documentation.

There is also an obsolete option for **pcre2_jit_compile()** called PCRE2_JIT_INVALID_UTF, which currently exists only for backward compatibility. It is superseded by the **pcre2_compile()** option PCRE2_MATCH_INVALID_UTF and should no longer be used. It may be removed in future.

UNSUPPORTED OPTIONS AND PATTERN ITEMS

The **pcre2_match()** options that are supported for JIT matching are PCRE2_COPY_MATCHED_SUBJECT, PCRE2_NOTBOL, PCRE2_NOTEOL, PCRE2_NOTEMPTY, PCRE2_NOTEMPTY_ATSTART, PCRE2_NO_UTF_CHECK, PCRE2_PARTIAL_HARD, and PCRE2_PARTIAL_SOFT. The PCRE2_ANCHORED and PCRE2_ENDANCHORED options are not supported at match time.

If the PCRE2_NO_JIT option is passed to **pcre2_match()** it disables the use of JIT, forcing matching by the interpreter code.

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 MATCHING

When a pattern is matched using JIT matching, the return values are the same as those given by the interpretive **pcre2_match()** code, with the addition of one new error code: PCRE2_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.

The error code PCRE2_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 PCRE2_ERROR_DEPTHLIMIT error code is never returned when JIT matching is used.

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 32KiB on the machine stack. However, some large or complicated patterns need more than this. The error PCRE2_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 **pcre2_jit_stack_create()** function creates a JIT stack. Its arguments are a starting size, a maximum size, and a general context (for memory allocation functions, or NULL for standard memory allocation). It returns a pointer to an opaque structure of type **pcre2_jit_stack**, or NULL if there is an error. The **pcre2_jit_stack_free()** function is used to free a stack that is no longer needed. If its argument is NULL, this function returns immediately, without doing anything. (For the technically minded: the address space is allocated by mmap or VirtualAlloc.) A maximum stack size of 512KiB to 1MiB should be more than enough for any pattern.

The **pcre2_jit_stack_assign()** function specifies which stack JIT code should use. Its arguments are as follows:

```
pcre2_match_context  *mcontext
pcre2_jit_callback    callback
void                 *data
```

The first argument is a pointer to a match context. When this is subsequently passed to a matching function, its information determines which JIT stack is used. If this argument is NULL, the function returns immediately, without doing anything. There are three cases for the values of the other two options:

(1) If *callback* is NULL and *data* is NULL, an internal 32KiB block
on the machine stack is used. This is the default when a match
context is created.

(2) If *callback* is NULL and *data* is not NULL, *data* must be a pointer to a valid JIT stack, the result of calling **pcre2_jit_stack_create()**.

(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 32KiB stack is used; otherwise the return value must be a valid JIT stack, the result of calling **pcre2_jit_stack_create()**.

A callback function is obeyed whenever JIT code is about to be run; it is not obeyed when **pcre2_match()** is called with options that are incompatible for JIT matching. 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 matched sequentially in the same thread. Currently, the only way to set up non-sequential matches in one thread is to use callouts: if a callout function starts another match, that match must use a different JIT stack to the one used for currently suspended match(es).

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 a match context that is used by any number of patterns, as long as they are not used for matching by multiple threads at the same time. For example, you could use the same stack in all compiled patterns, with 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 initialization
  thread_local_var = pcre2_jit_stack_create(...)

During thread exit
  pcre2_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.

JIT STACK FAQ

(1) Why do we need JIT stacks?

PCRE2 (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 1MiB address space, and use only a single memory page (usually 4KiB) if that is enough. However, we can still grow up to 1MiB 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 being used by **pcre2_match()**, (that is, it is assigned to a match context that is passed 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 **pcre2_match()** again. When you assign the stack to a match context, only a pointer is set. There is no reference counting or any other magic. You can free compiled patterns, contexts, and stacks in any order, anytime. Just *do not* call **pcre2_match()** with a match context pointing to an already freed stack, as that will cause SEGFAULT. (Also, do not free a stack currently used by **pcre2_match()** in another thread). You can also replace the stack in a context at any time when it is not in use. You should free the previous stack before assigning a replacement.

(5) Should I allocate/free a stack every time before/after calling **pcre2_match()**?

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 patterns.

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

FREEING JIT SPECULATIVE MEMORY

void pcre2_jit_free_unused_memory(pcre2_general_context ***gcontext*);**

The JIT executable allocator does not free all memory when it is possible. It expects new allocations, and keeps some free memory around to improve allocation speed. However, in low memory conditions, it might be better to free all possible memory. You can cause this to happen by calling pcre2_jit_free_unused_memory(). Its argument is a general context, for custom memory management, or NULL for standard memory management.

EXAMPLE CODE

This is a single-threaded example that specifies a JIT stack without using a callback. A real program should include error checking after all the function calls.

```
int rc;
pcre2_code *re;
pcre2_match_data *match_data;
pcre2_match_context *mcontext;
pcre2_jit_stack *jit_stack;

re = pcre2_compile(pattern, PCRE2_ZERO_TERMINATED, 0,
  &errornumber, &erroffset, NULL);
```

```
rc = pcre2_jit_compile(re, PCRE2_JIT_COMPLETE);
mcontext = pcre2_match_context_create(NULL);
jit_stack = pcre2_jit_stack_create(32*1024, 512*1024, NULL);
pcre2_jit_stack_assign(mcontext, NULL, jit_stack);
match_data = pcre2_match_data_create(re, 10);
rc = pcre2_match(re, subject, length, 0, 0, match_data, mcontext);
/* Process result */

pcre2_code_free(re);
pcre2_match_data_free(match_data);
pcre2_match_context_free(mcontext);
pcre2_jit_stack_free(jit_stack);
```

JIT FAST PATH API

Because the API described above falls back to interpreted matching when JIT is not available, it is convenient for programs that are written for general use in many environments. However, calling JIT via **pcre2_match()** 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 matching directly instead of calling **pcre2_match()** (obviously only for patterns that have been successfully processed by **pcre2_jit_compile()**).

The fast path function is called **pcre2_jit_match()**, and it takes exactly the same arguments as **pcre2_match()**. However, the subject string must be specified with a length; PCRE2_ZERO_TERMINATED is not supported. Unsupported option bits (for example, PCRE2_ANCHORED, PCRE2_ENDANCHORED and PCRE2_COPY_MATCHED_SUBJECT) are ignored, as is the PCRE2_NO_JIT option. The return values are also the same as for **pcre2_match()**, plus PCRE2_ERROR_JIT_BADOPTION if a matching mode (partial or complete) is requested that was not compiled.

When you call **pcre2_match()**, 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 but the length is non-zero, an immediate error is given. Also, unless PCRE2_NO_UTF_CHECK 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 **pcre2_match()** wrapping can give speedups of more than 10%.

SEE ALSO

pcre2api(3)

AUTHOR

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

REVISION

Last updated: 30 November 2021