WIKIPEDIA

setcontext

setcontext is one of a family of C library functions (the others being **getcontext**, **makecontext** and **swapcontext**) used for context control. The `setcontext` family allows the implementation in C of advanced control flow patterns such as iterators, fibers, and coroutines. They may be viewed as an advanced version of setjmp/longjmp; whereas the latter allows only a single non-local jump up the stack, `setcontext` allows the creation of multiple cooperative threads of control, each with its own stack.

Contents

Specification
Definitions
Example
References
External links

Specification

`setcontext` was specified in POSIX.1-2001 and the Single Unix Specification, version 2, but not all Unix-like operating systems provide them. POSIX.1-2004 obsoleted these functions, and in POSIX.1-2008 they were removed, with POSIX Threads indicated as a possible replacement. Citing IEEE Std 1003.1, 2004 Edition[1]:

> With the incorporation of the ISO/IEC 9899:1999 standard into this specification it was found that the ISO C standard (Subclause 6.11.6) specifies that the use of function declarators with empty parentheses is an obsolescent feature. Therefore, using the function prototype:
>
> ```
> void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
> ```
>
> is making use of an obsolescent feature of the ISO C standard. Therefore, a strictly conforming POSIX application cannot use this form. Therefore, use of getcontext(), makecontext(), and swapcontext() is marked obsolescent.
>
> There is no way in the ISO C standard to specify a non-obsolescent function prototype indicating that a function will be called with an arbitrary number (including zero) of arguments of arbitrary types (including integers, pointers to data, pointers to functions, and composite types).

Definitions

The functions and associated types are defined in the `ucontext.h` system header file. This includes the `ucontext_t` type, with which all four functions operate:

```
typedef struct {
    ucontext_t *uc_link;
    sigset_t    uc_sigmask;
    stack_t     uc_stack;
    mcontext_t  uc_mcontext;
    ...
} ucontext_t;
```

uc_link points to the context which will be resumed when the current context exits, if the context was created with makecontext (a secondary context). uc_sigmask is used to store the set of signals blocked in the context, and uc_stack is the stack used by the context. uc_mcontext stores execution state, including all registers and CPU flags, the instruction pointer, and the stack pointer; mcontext_t is an opaque type.

The functions are:

- `int setcontext(const ucontext_t *ucp)`

  This function transfers control to the context in ucp. Execution continues from the point at which the context was stored in ucp. setcontext does not return.

- `int getcontext(ucontext_t *ucp)`

  Saves current context into ucp. This function returns in two possible cases: after the initial call, or when a thread switches to the context in ucp via setcontext or swapcontext. The getcontext function does not provide a return value to distinguish the cases (its return value is used solely to signal error), so the programmer must use an explicit flag variable, which must not be a register variable and must be declared volatile to avoid constant propagation or other compiler optimisations.

- `void makecontext(ucontext_t *ucp, void *func(), int argc, ...)`

  The makecontext function sets up an alternate thread of control in ucp, which has previously been initialised using getcontext. The ucp.uc_stack member should be pointed to an appropriately sized stack; the constant SIGSTKSZ is commonly used. When ucp is jumped to using setcontext or swapcontext, execution will begin at the entry point to the function pointed to by func, with argc arguments as specified. When func terminates, control is returned to ucp.uc_link.

- `int swapcontext(ucontext_t *oucp, ucontext_t *ucp)`

  Transfers control to ucp and saves the current execution state into oucp.

Example

The example below demonstrates an iterator using setcontext.

```
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>

/* The three contexts:
 *    (1) main_context1 : The point in main to which loop will return.
 *    (2) main_context2 : The point in main to which control from loop will
 *                        flow by switching contexts.
 *    (3) loop_context  : The point in loop to which control from main will
 *                        flow by switching contexts. */
ucontext_t main_context1, main_context2, loop_context;

/* The iterator return value. */
volatile int i_from_iterator;

/* This is the iterator function. It is entered on the first call to
 * swapcontext, and loops from 0 to 9. Each value is saved in i_from_iterator,
 * and then swapcontext used to return to the main loop.  The main loop prints
 * the value and calls swapcontext to swap back into the function. When the end
 * of the loop is reached, the function exits, and execution switches to the
 * context pointed to by main_context1. */
void loop(
    ucontext_t *loop_context,
    ucontext_t *other_context,
    int *i_from_iterator)
{
    int i;

    for (i=0; i < 10; ++i) {
        /* Write the loop counter into the iterator return location. */
        *i_from_iterator = i;

        /* Save the loop context (this point in the code) into ''loop_context'',
         * and switch to other_context. */
        swapcontext(loop_context, other_context);
    }

    /* The function falls through to the calling context with an implicit
     * ''setcontext(&loop_context->uc_link);'' */
}

int main(void)
{
    /* The stack for the iterator function. */
    char iterator_stack[SIGSTKSZ];

    /* Flag indicating that the iterator has completed. */
    volatile int iterator_finished;

    getcontext(&loop_context);
    /* Initialise the iterator context. uc_link points to main_context1, the
     * point to return to when the iterator finishes. */
    loop_context.uc_link          = &main_context1;
    loop_context.uc_stack.ss_sp   = iterator_stack;
    loop_context.uc_stack.ss_size = sizeof(iterator_stack);

    /* Fill in loop_context so that it makes swapcontext start loop. The
     * (void (*)(void)) typecast is to avoid a compiler warning but it is
     * not relevant to the behaviour of the function. */
    makecontext(&loop_context, (void (*)(void)) loop,
        3, &loop_context, &main_context2, &i_from_iterator);

    /* Clear the finished flag. */
    iterator_finished = 0;

    /* Save the current context into main_context1. When loop is finished,
     * control flow will return to this point. */
    getcontext(&main_context1);

    if (!iterator_finished) {
        /* Set iterator_finished so that when the previous getcontext is
         * returned to via uc_link, the above if condition is false and the
         * iterator is not restarted. */
        iterator_finished = 1;

        while (1) {
            /* Save this point into main_context2 and switch into the iterator.
             * The first call will begin loop.  Subsequent calls will switch to
             * the swapcontext in loop. */
            swapcontext(&main_context2, &loop_context);
            printf("%d\n", i_from_iterator);
        }
    }

    return 0;
}
```

NOTE: this example is not correct[1], but may work as intended in some cases. The function `makecontext` requires additional parameters to be type `int`, but the example passes pointers. Thus, the example may fail on 64-bit machines (specifically LP64-architectures, where `sizeof(void*) > sizeof(int)`). This problem can be worked around by breaking up and reconstructing 64-bit values, but that introduces a performance penalty.

> On architectures where int and pointer types are the same size (e.g., x86-32, where both
>
> types are 32 bits), you may be able to get away with passing pointers as arguments to makecontext() following argc. However, doing this is not guaranteed to be portable, is undefined according to the standards, and won't work on architectures where pointers are larger than ints. Nevertheless, starting with version 2.8, glibc makes some changes to `makecontext(3) (http://man7.org/linux/man-pages/man3/makecontext.3.html)`, to permit this on some 64-bit architectures (e.g., x86-64).

For get and set context, a smaller context can be handy:

```
#include <stdio.h>
#include <ucontext.h>
#include <unistd.h>

int main(int argc, const char *argv[]){
    ucontext_t context;

    getcontext(&context);
    puts("Hello world");
    sleep(1);
    setcontext(&context);
    return 0;
}
```

This makes an infinite loop because context holds the program counter.

References

1. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition [1] (http://www.opengroup.org/onlinepubs/009695399/functions/makecontext.html)

External links

- System V Contexts (https://www.gnu.org/software/libc/manual/html_mono/libc.html#System-V-contexts) - The GNU C Library Manual
- `setcontext(3) (http://man7.org/linux/man-pages/man3/setcontext.3.html)`: get/set current user context – Linux Programmer's Manual – Library Functions
- setcontext - get/set current user context (http://www.freebsd.org/cgi/man.cgi?query=setcontext&manpath=FreeBSD+7.0-RELEASE) FreeBSD man page.