



## Kernel: process management

So, you need a kernel? In this tutorial you will learn how to create and execute processes in BeRTOS and what synchronization facilities you have in your toolbox

### Processes

If you have programmed with threads in a desktop operating system, you will find some similarities with BeRTOS processes. Every process is defined by:

- ▶ a function that is executed when the process is running;
- ▶ some user data that the process can use to communicate with other processes;
- ▶ a memory area used for the stack (remember that BeRTOS has a fully static memory model).

Unlike threads, however, processes have are independent from the parent process, they can have a priority (if enabled) and they can be monitored to detect stack overflows.

### First example

Enough talking. In this section we are going to create three processes (one the main process plus two extra) that control some LEDs.

First, initialize the scheduler calling `proc_init()`, then create two processes using `proc_new()`. Remember that you need to provide some stack space to the process; usually `KERN_MINSTACKSIZE` should be enough, but read carefully the documentation of this variable if you're using a 16-bit architecture.

You can allocate the stack for both processes with the following lines:

```
PROC_DEFINE_STACK(stack1, KERN_MINSTACKSIZE);
PROC_DEFINE_STACK(stack2, KERN_MINSTACKSIZE);
```

This macro defines a memory buffer. The first parameter is the buffer name, while the second is the size in bytes.

#### NOTE:

It's not possible to determine the optimal stack size for a generic process before running it. `KERN_MINSTACKSIZE` is a good value to start working, but it's not the best value for every use case. For example, a process which uses `kprintf()` must have a bigger stack, at least  $(KERN\_MINSTACKSIZE * 2)$ , because formatting requires lots of memory.

Then let's define the entry points for the two processes:

```
void proc1(void)
{
    int times = ;
    bool light_on = false;
    while (true)
    {
        light_on = !light_on;
```

```

// light on or off led1
TURN_LED_ON(1, !light_on);

++times;
timer_delay(20);
if (times > 30)
    return;
}
}

// proc2 is similar to proc1
void proc2(void)
{
    int times = ;
    bool light_on = false;
    while (true)
    {
        light_on = !light_on;
        // light on or off led2
        TURN_LED_ON(2, !light_on);
        ++times;
        timer_delay(40);
        if (times > 30)
            return;
    }
}

```

As you can see, you can define variables inside the process and they will be automatically saved: that's why we defined a stack space a bit larger. Of course you may need more stack space in your application, so feel free to create a larger stack for your processes. You can also create a larger stack for one process only, if you need that.

The macro `TURN_LED_ON(led, value)` turns the led `led` on or off depending on `value`. This macro must be defined in the `hw/` directory, which contains all hardware specific macros and functions, in the file `hw_leds.h`. See [Program Configuration](#) for further explanations.

### REMEBER:

if the entry point of the process reaches the end of the function, the process will quit (just like any plain program). Thus, if you want to create a process that will run forever in your application, you must enclose the function with a neverending loop like `while (true) { ... }` above. You can still exit the process under certain conditions by using `return`.

Now in our main program we just need to create the two processes and set the priority for each of them.

```

void main(void)
{
    proc_init()
    // other initializations...
    // ...
    Process *p = proc_new(proc1, NULL, sizeof(stack1), stack1);
    proc_setPri(p, -5);

    p = proc_new(proc2, NULL, sizeof(stack2), stack2);
}

```

```

proc_setPri(p, 10);

// now wait for both processes to complete
// note that "main" is itself a process
ticks_t start = timer_clock();
while (timer_clock() - start < ms_to_ticks(3000))
{
    kprintf("main\n");
    timer_delay(500);
}
}

```

We create two processes with `proc_new()`, which takes the entry point, some user data (NULL in this case) and the stack to operate on.

We also define a *priority* for each of the two processes. Priorities are signed integers, where positive numbers have a higher priority over negative numbers. You should set priorities for your processes in the range -10,+10 (inclusive) to avoid interfering with other system tasks.

We're done. If you run the example, you will note that the second led will light before the first at the start. You can also add some debug prints to see process scheduling.

## Process monitor

This process monitors all the other process to check wheter a stack overflow has occurred; in this case you get a warning on the debug serial port.

The monitor is enabled with `CONFIG_KERN_MONITOR` flag in `cfg/cfg_monitor.h`, so you can enable it at debug time and remove it when releasing the software.

## Synchronization example

You can synchronize processes using semaphores. In this example we will create one process that reads commands from the serial port and puts them in a queue and one process that reads such commands and lights the LEDs.

A common way to synchronize processes is to use semaphores. In our example you would lock the FIFO queue with the semaphore and then access it to read (or write) commands.

```

static void serialRead(void)
{
    while (1)
    {
        // read data from serial line...
        size_t count = kfile_read(&ser_port.fd, buf, sizeof(buf));
        // ...put the data in the queue
        // ...
    }
}

```

The code for the second process (the "worker") has the same structure, but instead of reading from serial line, it will get some data from the queue, interpret it and it will finally light the LEDs.

As you can see, thanks to the cooperative kernel we know each time that memory access is exclusive, so there's no need to lock shared data structures. You can safely access each data without the risk of race conditions.

I would like to stress that all BeRTOS modules are designed to **automatically switch context** when there is a wait in

progress, so there's no need to worry to release the CPU manually. In the above example, the function `kfile_read()` automatically releases the CPU when there are no more input chars on serial port. If you need to write new code which requires lots of CPU power, you must remember to release the CPU from time to time using the `cpu_relax()` function.

Semaphores can be useful in some corner cases, such as:

- ▶ two processes access the same serial: `kfile_read()` executes a context switch, so the serial must be protected with a semaphore;
- ▶ two processes access the same structure and there can be a context switch in the meanwhile. Example:

```
uint8_t buf[10];

void procl(void)
{
    // ...
    // this will switch context if there are no chars to read
    kfile_read(&ser.fd, buf, sizeof(buf));
    // ...
}

void proc2(void)
{
    // ...
    // whoops, race condition
    kfile_read(&ser.fd, buf, sizeof(buf));
}
```

Have a look at [semaphore API](#) for further explanations.

## Messages

Synchronizing processes using semaphores can be tricky, since some code paths can lead to deadlocks and race conditions. For example, if you forget to release a semaphore after doing your work, no other process can access the data the semaphore protects. Also, you must remember to explicitly release the CPU when done, which is error prone.

To avoid such problems, BeRTOS provides a very lightweight yet powerful communication method: **Message Queues**. The concept behind this tool is easy: define your custom message, declare an input port for each process and wait for signals coming from the port.

### DEFINING CUSTOM MESSAGES

The `Msg` struct is minimalistic, so you need to expand it to contain whatever data you want.

```
#include <kern/msg.h>
typedef struct Command
{
    Msg msg;
    int cmd_id;
    int parameters[MAX_PARMS_CNT];
    int result;
} Command;
```

We have defined a Command which has an id and some parameters to operate on, together with the return code. Now we can send it to a process and let it do the work.

## DECLARE THE INPUT PORT

Each process that wants to communicate with other processes must have an **input port** and, optionally, a **reply port**. In our example the input port will be used by other processes to schedule tasks on the worker process. Remember that tasks in the message queue are processed in FIFO order.

Let's declare a process with its ports:

```
MsgPort procl_in_port;
MsgPort main_reply_port;

void procl_main(void)
{
    // we will see this in a moment
    msg_initPort(...);
    while (1)
    {
        // wait for signals, see below
    }
}

int main(void)
{
    // init everything here
    //...
    // also init reply port
    msg_initPort(...);

    Command cmd;
    cmd.cmd_id = CMD_WASTE_TIME;
    cmd.parameters[0] = 1000;    //secs
    cmd.msg.replyPort = &main_reply_port;
    msg_put(&procl_in_port, &cmd.msg);
    // ...
    // wait for reply signals, see below
}
```

The main process creates a worker process then sends commands through the message port. The command struct must be filled with valid data (of course); we also fill in the reply port to get the answer. When everything is done, just put it in the port and you're (almost) done.

To init a port, you must declare which signal will be emitted when a message is received. Signals are a kernel feature which is very cheap to use but it carries almost no information; see [signals documentation](#). However, in this case signals are just enough!

So, how do we init a port? This is the code:

```
/* rename standard signal to something more useful */
#define SIG_CMD_ARRIVED SIG_USER0
```

```
msg_initPort(&procl_in_port, event_createSignal(proc_current(), SIG_CMD_ARRIVED);
```

First, rename one of the standard signals SIG\_USERn to something sensible; then init the port with the above line. Whenever a new message arrives, a signal is sent to the process. How do you wait for a signal? It's just one call:

```
// wait just one signal, but you can OR more signals at the same time
sigmask_t sigs = sig_wait(SIG_CMD_ARRIVED);
```

This call will put the process to sleep, waiting for a signal. Note that the process will **not be blocked in a spinlock**, polling the signal somewhere, **nor it will be in the ready queue** in the kernel; it will be frozen without wasting CPU resources. When the signal arrives, there's a context switch to the waiting process, which will wake up and resume execution.

Once the command is executed, you may want to return a code to let the calling process know what happened. To this end you can use the reply port we defined earlier.

```
//...
// get the first message from port
Command *cmd = (Command *)msg_get(&procl_in_port)
// execute the command
switch (cmd->cmd_id)
{
  //...
}
// res is the return code of the above commands
cmd.result = res;
msg_reply(cmd);
```

See that the message already knows which is its reply port, so you don't need to specify it. Note that when using `msg_get()` without putting the message into another port, you must take care of recycle the message, by `free()` ing it or, with static memory, by putting it again in a free message pool.

One final remark: you can put really everything inside a message, be it strings, function pointers, arrays or other structures. **There won't be any performance hit** depending on the size of message struct because messages are simply moved from one port (a linked list) to another, so moving a message between ports is just a matter of swapping a few pointers.

That's it. Look at [message documentation](#) for full API description.

#### DEVELOPER'S TRAC

On the developers website you will find all the information to contribute to BeRTOS development community, milestones, the possibility to open tickets to the community, the changelog and all the details on every feature. [dev.bertos.org](http://dev.bertos.org)

#### PREMIUM SUPPORT

By registering to Develer's advanced support services you will have online assistance, phone assistance and the possibility to report bugs directly to our BeRTOS engineering team. All services guarantee reply in 24 working hours. [bertos.org/supporto/premium/](http://bertos.org/supporto/premium/)

#### CERTIFIED TRAINING

Customized training, onsite teaching, embedded application debug and deploy assistance, peer review on source code, training classes for single programmers or working teams. [bertos.org/supporto/formazione/](http://bertos.org/supporto/formazione/)



BeRTOS website by [Develer](#).