Tutorial 3 – Contiki Process and Events

Aim

Contiki is an event based operating system that designed to operate on resource scarce and low power embedded system platforms. This tutorial shows you how to create a process and to use events.

- **Contiki Concepts**

1. Process
2. Events

Example code:

The hello-world example code consists of 1 process.
Location: 'contiki-examples/hello-world

Contiki Process

A contiki process is a program functional unit that executes a thread (like other operating systems). One of the key advantages of processes is multitasking and concurrent execution. For example, different functions, such as processing sensor values and transmitting sensor values, can be done simultaneously, from the program's point of view.

Process Declaration

Processes are declared as functions at the top of your program, with the `PROCESS` definition.

```
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
```

Processes must have the `PROCESS_BEGIN()` `PROCESS_END()` declarations at the beginning and end of the process. Refer to example ex1_helloworld.

```
PROCESS_THREAD(hello_world_process, ev, data) {

        PROCESS_BEGIN();

        printf("Hello, world\n");

        PROCESS_END();
}
```

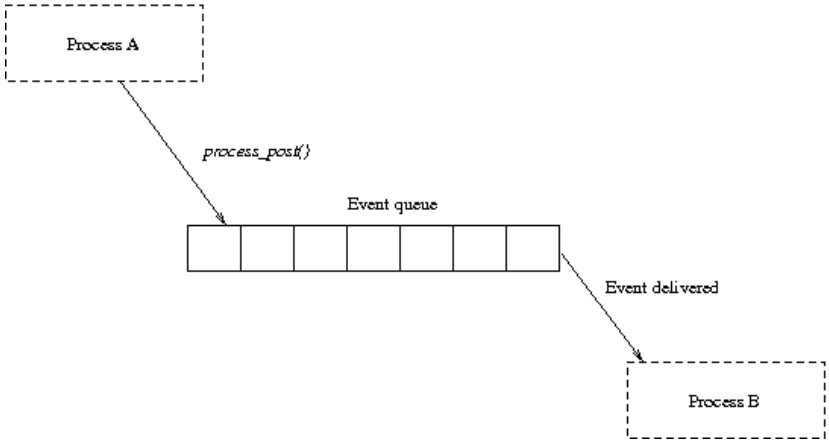
To automatically start a process when the platform turns on, the `AUTOSTART_PROCESSES` function must be called at the beginning.

```
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
```

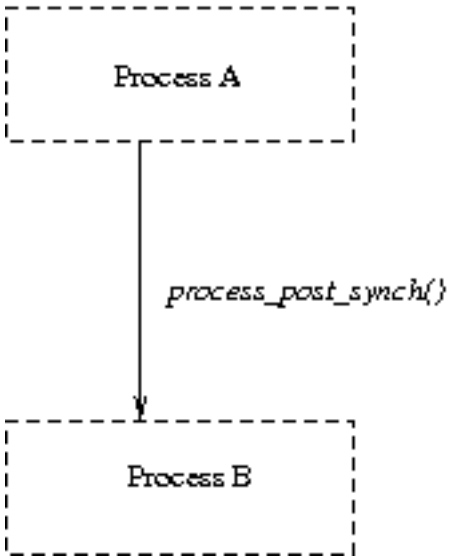
Refer to ex1_helloworld example and to the Contiki Process documentation for more information.

Contiki Event

In Contiki, a process is run when it receives an event. There are two types of events: asynchronous events and synchronous events.



When an asynchronous events is posted, the event is put on the kernel's event queue and delivered to the receiving process at some later time.



When a synchronous event is posted, the event is immediately delivered to the receiving process.

Asynchronous Events

Asynchronous events are delivered to the receiving process some time after they have been posted. Between their posting and their delivery, the asynchronous events are held

on an event queue inside the Contiki kernel.

The events on the event queue are delivered to the receiving process by the kernel. The kernel loops through the event queue and delivers the events to the processes on the queue by invoking the processes.

The receiver of an asynchronous event can either be a specific process, or all running processes. When the receiver is a specific process, the kernel invokes this process to deliver the event. When the receiver of an event is set to be all processes in the system, the kernel sequentially delivers the same event to all processes, one after another.

Asynchronous events are posted with the *process_post()* function. The internals of the process_post() function is simple. It first checks the size of the current event queue to determine if there is room for the event on the queue. If not, the function returns an error. If there is room for the event on the queue, the function inserts the event at the end of the event queue and returns.

Synchronous Events

Unlike asynchronous events, synchronous events are delivered directly when they are posted. Synchronous events can only be posted to a specific processes.

Because synchronous events are delivered immediately, posting a synchronous event is functionally equivalent to a function call: the process to which the event is delivered is directly invoked, and the process that posted the event is blocked until the receiving process has finished processing the event. The receiving process is, however, not informed whether the event was posted synchronously or asynchronously. The function: process_post_synch() is used.

Polling

A poll request is a special type of event. A process is polled by calling the function process_poll(). Calling this function on a process causes the process to be scheduled as quickly as possible. The process is passed a special event that informs the process that it has been polled.

Polling is the way to make a process run from an interrupt. The process_poll() function is the only function in the process module that is safe to call from preemptive mode.

Event Identifiers

Events are identified by an event identifier. The event identifier is an 8-bit number that is passed to the process that receives an event. The event identifier allows the receiving process to perform different actions depending on what type of event that was received.

Event identifiers below 127 can be freely used within a user process, whereas event identifiers above 128 are intended to be used between different processes. Identifiers above 128 are managed by the kernel.

The first numbers over 128 are statically allocated by the kernel, to be used for a range of different purposes. The definition for these event identifiers are shown in below.

Event identifiers reserved by the Contiki kernel:

- PROCESS_EVENT_NONE
- PROCESS_EVENT_INIT

- PROCESS_EVENT_POLL
- PROCESS_EVENT_CONTINUE //Used for waiting processes
- PROCESS_EVENT_MSG
- PROCESS_EVENT_EXITED
- PROCESS_EVENT_TIMER

Example showing process waiting for event to be given or posted

```
#include "contiki.h"

PROCESS(example_process, "Example process");
AUTOSTART_PROCESSES(&example_process);

PROCESS_THREAD(example_process, ev, data)  {
        PROCESS_BEGIN();
        while(1) {
                PROCESS_WAIT_EVENT();
                printf("Got event number %d\n", ev);
        }
        PROCESS_END();
}
```

Example of posting an event

```
post_process_synch(&example_process, PROCESS_EVENT_CONTINUE, NULL);
```

See Example: contiki-examples/process_events.

Conclusion

This lesson has introduced the process concept of the Contiki operating system. Refer to ex1_helloworld example and to the Contiki documentation for more information.

Related Documentation

- Contiki Wiki
- Contiki Processes Wiki