ProtoThreads

Report

Protothreads (PT) are a type of lightweight stackless threads designed for low memory constrained systems (Micro Controllers) such as deeply embedded systems or sensor network nodes. **Protothreads provides linear code execution for event-driven systems implemented in C**. (Protothreads can be used with or without an RTOS). **PT provides a blocking context on top of an event-driven system**, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. **Protothreads provides conditional blocking inside C functions.**

Main features:

- No machine specific code - the **protothreads library is pure C**
- Does not use error-prone functions such as longjmp()
- **Very small RAM overhead - only two bytes per protothread**
- **Can be used** with or **without an OS**
- Provides conditional blocking (inside the C function) wait without full multi-threading or stack-switching

Examples applications:

- **Memory constrained systems**
- **Event-driven protocol stacks**
- **Deeply embedded systems**
- **Sensor network nodes**

To use protothreads in a project is easy: simply copy the files **pt.h, lc.h** and **lc-switch.h** into the **include files directory of the project**, and **#include "pt.h"** in files that you use with protothreads.

Protothreads do not save the stack context across a blocking call, **local variables are not kept when the protothread blocks.** This means that local variables should be used really carefully - if you are in doubt, do not use local variables inside a protothread!

The protothread function will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

Usually Threads require per-thread stack memory . On the other hand Protothreads like events require one stack for all the event-handlers .

Actually Protothreads can run on top of existing event-based kernels without modifications to the underlying event-driven system (event dispatching system).

The Protothreads Library File Index:

lc-addrlabels.h
(Implementation of local continuations based on the "Labels as values")

lc-switch.h
(Implementation of local continuations based on switch() statement)

lc.h (Local continuations)

pt-sem.h
(Counting semaphores implemented on protothreads)

pt.h (Protothreads implementation)

Protothreads are implemented in a single header file, pt.h, which includes the local continuations header file, lc.h.

Modules :

- Protothread Semaphores
- Local Continuations
- Examples on pt refman

Protothread file structure :

Data Structure : struct pt

Initialization : #define PT_INIT(pt) // Initialise protothread

Declaration & definitions :

#define PT_THREAD(name_args) //Declaration of pt

#define PT_BEGIN(pt)
//Declare the start of pt inside C func implementing pt

#define PT_END(pt)
//Declare the end of pt

Blocked wait :

#define PT_WAIT_UNTIL(pt, condition)
//Block and wait until condition is true

#define PT_WAIT_WHILE(pt, cond)
//Block and wait while condition is true

Hierarchical protothreads :

#define PT_WAIT_THREAD(pt, thread)
//Block and wait until a child protothread completes

#define PT_SPAWN(pt, child, thread)
//Spawn a child protothread and wait until it exits

Exiting and restarting :

#define PT_RESTART(pt)
// Restart the protothread

#define PT_EXIT(pt)
// Exit the protothread

Yielding from a protothread :

#define PT_YIELD(pt)
//Yield from the current protothread

#define PT_YIELD_UNTIL(pt, cond)
//Yield from the protothread until a condition occurs

Calling Protothreads :

#define PT_SCHEDULE(f) // Schedule a protothread

The protothread runs on a C function, protothread scheduling is done by invoking its function. We invoke protothreads as blocking event handlers. In fact we can invoke protothreads with other protothreads using wait until a child protothread completes (Hierarchical protothreads).

Defines :

```
#define PT_WAITING 0
#define PT_YIELDED 1
#define PT_EXITED 2
#define PT_ENDED 3
```

Example of Protothread Implementation :

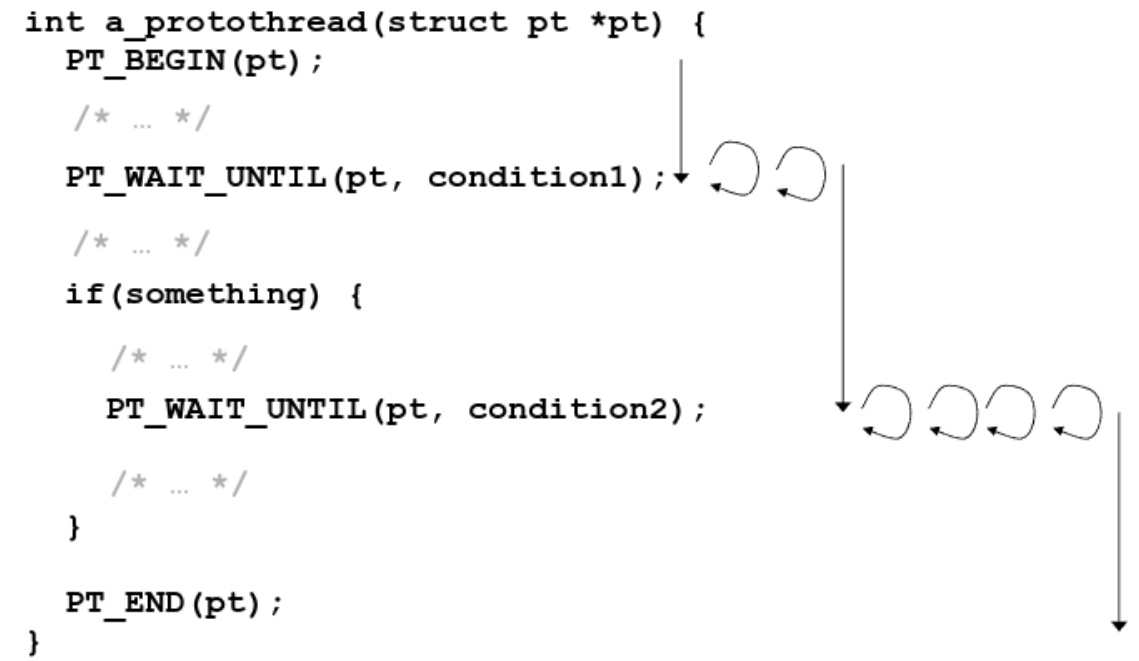
```
#include “pt.h”
#include <stdio.h>

static int counter;
static struct pt example_protothread;

static int example_protothread_function(struct pt *pt){

        PT_BEGIN(pt);
        while(1){
                PT_WAIT_UNTIL(pt, counter == 100);
                printf(“Limit reached”);
                //print function depends on the environment you want to use
                counter = 0;
        }
        PT_END(pt);
}

int main(void){
        counter = 0;
        PT_INIT(&example_protothread);
```

```
        while(1){
                example_protothread_function(&example_protothread);
                counter++;
        }
        return 0;
}
```

About the example :

The protothread on the example above wait for the counter to reach a certain value this is done by **PT_WAIT_UNTIL(pt, counter == 100);** then the protothread prints out a message and resets the counter. On the main of the program we use **while(1)** so it can loop forever and the counter is increased as well there **.**

Another example :



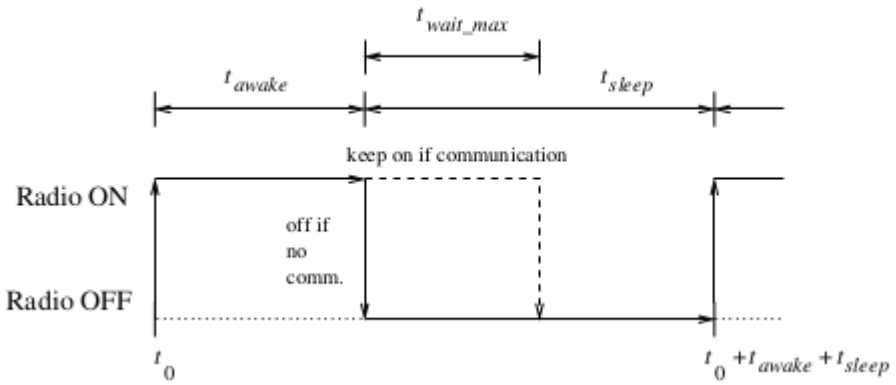
Implementation Notes :

Proof-of-concept implementation is Pure ANSI C, no changes to compiler, no special preprocessor, no assembly language. Deviations on protothreads, **automatic variables are not saved across a blocked wait** (you may use global variables), restrictions on switch() statements.

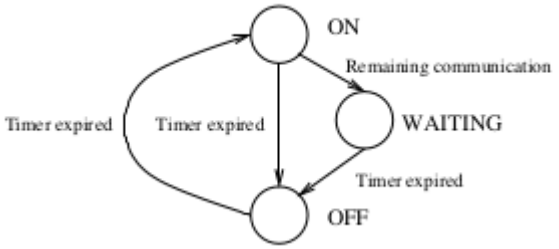
Radio Sleep Cycle implemented with protothreads, in pseudocode.

```
radio_wake_protothread:
  PT_BEGIN
  while (true)
    radio_on()
    timer ← t_awake
    PT_WAIT_UNTIL(expired(timer))
    timer ← t_sleep
    if (not communication_complete())
      wait_timer ← t_wait_max
      PT_WAIT_UNTIL(communication_complete() or
                    expired(wait_timer))
    radio_off()
    PT_WAIT_UNTIL(expired(timer))
  PT_END
```

Hypothetical sensor network MAC protocol.



State Machine realisation of the sleep cycle of the example MAC protocol.



Protothread-based implementation is shorter :

```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```

Yielding Protothreads :

PT_YIELD() performs a single unconditional blocking wait that temporary blocks the protothread until the next time the protothread is invoked, by the next invocation the protothread continues executing the code following(below) the PT_YIELD() statement.

Hierarchical Protothreads :

More complex operations may need a hierarchical fashion decomposition, This operation is supported by protothreads with PT_SPAWN() operation, it initializes a child protothread and blocks the current protothread until the child protothread has ended PT_END or exited with PT_EXIT, the scheduling is done by the parent protothread.

Hypothetical data collection protocol implemented with hierarchical protothreads, in pseudocode.

```
reliable_send(message):
  rxtimer: timer
  PT_BEGIN
  do
    rxtimer ← t_retransmission
    send(message)
    PT_WAIT_UNTIL(ack_received() or expired(rxtimer))
  until (ack_received())
  PT_END

data_collection_protocol
  child_state: protothread_state
  PT_BEGIN
  while (running)
    while (interests_left_to_relay())
      PT_WAIT_UNTIL(interest_message_received())
      send_ack()
      PT_SPAWN(reliable_send(interest), child_state)
    while (data_left_to_relay())
      PT_WAIT_UNTIL(data_message_received())
      send_ack()
      PT_SPAWN(reliable_send(data), child_state)
  PT_END
```

The program consists of a main protothread, data collection protocol, that invokes a child protothread, reliable send, to do transmission of the data.

Local Continuations :

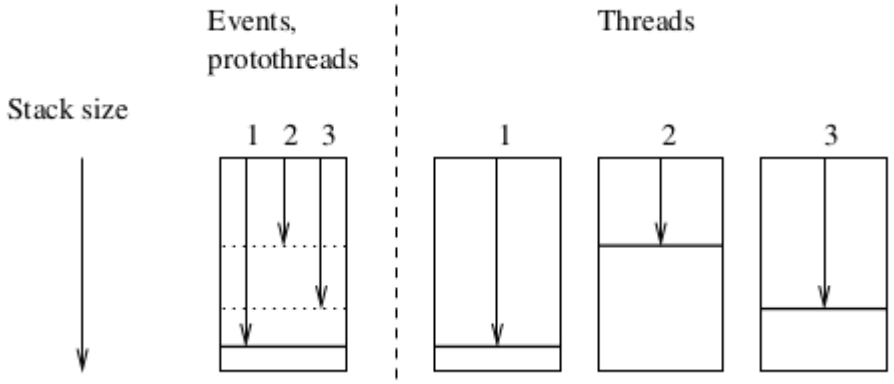
Local Continuations are the low-level mechanism that underpins protothreads. When a protothread blocks, the state of the protothread is stored in a local continuation. A local continuation does not capture the program stack. Rather, a local continuation only captures the state of execution inside a single function.a local continuation does not capture the program stack. Rather, a local continuation only captures the state of execution inside a single function.

Local Continuation has two operations : **set** and **resume** .

When the LC is set the state execution is store in local continuation, the state can be later restored with resume operation. The local continuation does not contain the stack, but only the state of the current function.

About Memory :

The three event handlers are rewritten with protothreads, and the equivalent functions running in three threads. Event handlers and protothreads run on the same stack, whereas each thread runs on a stack of its own.



Expanded C code with local continuations implemented with the C switch statement.

```
1 int sender(pt) {
2   PT_BEGIN(pt);
3
4   /* ... */
5   do {
6
7     PT_WAIT_UNTIL(pt,
8                   cond1);
9
10  } while(cond);
11  /* ... */
12  PT_END(pt);
13
14 }
```

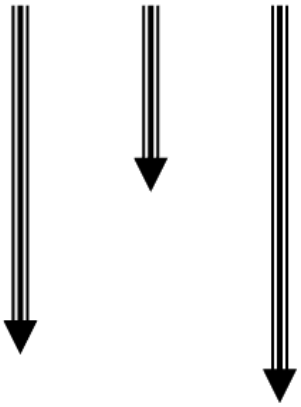
```
int sender(pt) {
  switch(pt->lc) {
  case 0:
    /* ... */
    do {
      pt->lc = 8;
  case 8:
      if(!cond1)
      return PT_WAITING;
    } while(cond);
    /* ... */
  }
  return PT_ENDED;
}
```

Protothreads Benefits :

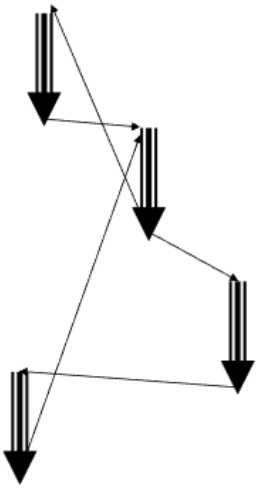
- Reduction in code complexity over state machines
- Code size
- Execution time overheap
- Memory requirements are low 2 bytes per protothread, no stacks
- Tested and recommended

Threads vs events…

Threads: sequential code flow



Events: unstructured code flow



Event non blocking wait ->