Protothreads

The Protothreads Library 1.4 Reference Manual

October 2006

Adam Dunkels
adam@sics.se

Swedish Institute of Computer Science

Contents

1 The Protothreads Library

Author:
Adam Dunkels <adam@sics.se>

Protothreads are a type of lightweight stackless threads designed for severly memory constrained systems 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.

Protothreads are a extremely lightweight, stackless type of threads that 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 blocking wait without full multi-threading or stack-switching

Examples applications:

- Memory constrained systems
- Event-driven protocol stacks

- Deeply embedded systems
- Sensor network nodes

See also:
Example programs
Protothreads API documentation

The protothreads library is released under a BSD-style license that allows for both non-commercial and commercial usage. The only requirement is that credit is given.

More information and new version of the code can be found at the Protothreads homepage:

http://www.sics.se/~adam/pt/

1.1 Authors

The protothreads library was written by Adam Dunkels <adam@sics.se> with support from Oliver Schmidt <ol.sc@web.de>.

1.2 Using protothreads

Using 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 all files that should use protothreads.

1.3 Protothreads

Protothreads are a extremely lightweight, stackless threads that 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 using complex state machines or full multi-threading. Protothreads provides conditional blocking inside a C function.

In memory constrained systems, such as deeply embedded systems, traditional multi-threading may have a too large memory overhead. In traditional multi-threading, each thread requires its own stack, that typically is over-provisioned. The stacks may use large parts of the available memory.

The main advantage of protothreads over ordinary threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. A protothread only requires only two bytes of memory per protothread. Moreover, protothreads are implemented in pure C and do not require any machine-specific assembler code.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead made by spawning a separate protothread for each potentially blocking function. The advantage of this approach is that blocking is explicit: the programmer knows exactly which functions that block that which functions the never blocks.

Protothreads are similar to asymmetric co-routines. The main difference is that co-routines uses a separate stack for each co-routine, whereas protothreads are stackless. The most similar mechanism to protothreads are Python generators. These are also stackless constructs, but have a different purpose. Protothreads provides blocking contexts inside a C function, whereas Python generators provide multiple exit points from a generator function.

1.4 Local variables

Note:

Because protothreads do not save the stack context across a blocking call, local variables are not preserved when the protothread blocks. This means that local variables should be used with utmost care - if in doubt, do not use local variables inside a protothread!

1.5 Scheduling

A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

1.6 Implementation

Protothreads are implemented using local continuations. A local continuation represents the current state of execution at a particular place in the program, but does not provide any call history or local variables. A local continuation can be set in a specific function to capture the state of the function. After a local continuation has been set can be resumed in order to restore the state of the function at the point where the local continuation was set.

Local continuations can be implemented in a variety of ways:

1. by using machine specific assembler code,
2. by using standard C constructs, or
3. by using compiler extensions.

The first way works by saving and restoring the processor state, except for stack pointers, and requires between 16 and 32 bytes of memory per protothread. The exact amount of memory required depends on the architecture.

The standard C implementation requires only two bytes of state per protothread and utilizes the C switch() statement in a non-obvious way that is similar to Duff's device. This implementation does, however, impose a slight restriction to the code that uses protothreads: a protothread cannot perform a blocking wait (PT_WAIT_UNTIL() or PT_YIELD()) inside a switch() statement.

Certain compilers has C extensions that can be used to implement protothreads. GCC supports label pointers that can be used for this purpose. With this implementation, protothreads require 4 bytes of RAM per protothread.

2 The Protothreads Library 1.4 Module Index

2.1 The Protothreads Library 1.4 Modules

Here is a list of all modules:

3 The Protothreads Library 1.4 Hierarchical Index

3.1 The Protothreads Library 1.4 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 The Protothreads Library 1.4 Data Structure Index

4.1 The Protothreads Library 1.4 Data Structures

Here are the data structures with brief descriptions:

5 The Protothreads Library 1.4 File Index

5.1 The Protothreads Library 1.4 File List

Here is a list of all documented files with brief descriptions:

6 The Protothreads Library 1.4 Module Documentation

6.1 Protothreads

6.1.1 Detailed Description

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

This file in turn includes the actual implementation of local continuations, which typically also is contained in a single header file.

Files

- file pt.h

  Protothreads implementation.

Modules

- Protothread semaphores

  This module implements counting semaphores on top of protothreads.

- Local continuations

  Local continuations form the basis for implementing protothreads.

Data Structures

- struct pt

Initialization

- #define PT_INIT(pt)

  Initialize a protothread.

Declaration and definition

- #define PT_THREAD(name_args)

  Declaration of a protothread.

- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the C function implementing the protothread.

- #define PT_END(pt)

  Declare the end of a protothread.

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.

Calling a protothread

- #define PT_SCHEDULE(f)

  Schedule a 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.

Defines

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

6.1.2 Define Documentation

6.1.2.1 #define PT_BEGIN(pt)

Declare the start of a protothread inside the C function implementing the protothread.

This macro is used to declare the starting point of a protothread. It should be placed at the start of the function in which the protothread runs. All C statements above the PT_BEGIN() invokation will be executed each time the protothread is scheduled.

Parameters:
pt A pointer to the protothread control structure.

Definition at line 115 of file pt.h.

6.1.2.2 #define PT_END(pt)

Declare the end of a protothread.

This macro is used for declaring that a protothread ends. It must always be used together with a matching PT_BEGIN() macro.

Parameters:
pt A pointer to the protothread control structure.

Definition at line 127 of file pt.h.

6.1.2.3 #define PT_EXIT(pt)

Exit the protothread.

This macro causes the protothread to exit. If the protothread was spawned by another protothread, the parent protothread will become unblocked and can continue to run.

Parameters:
pt A pointer to the protothread control structure.

Definition at line 246 of file pt.h.

6.1.2.4 #define PT_INIT(pt)

Initialize a protothread.

Initializes a protothread. Initialization must be done prior to starting to execute the protothread.

Parameters:
pt A pointer to the protothread control structure.

See also:
PT_SPAWN()

Definition at line 80 of file pt.h.

6.1.2.5 #define PT_RESTART(pt)

Restart the protothread.

This macro will block and cause the running protothread to restart its execution at the place of the PT_BEGIN() call.

Parameters:
pt A pointer to the protothread control structure.

Definition at line 229 of file pt.h.

6.1.2.6 #define PT_SCHEDULE(f)

Schedule a protothread.

This function shedules a protothread. The return value of the function is non-zero if the protothread is running or zero if the protothread has exited.

Parameters:

f The call to the C function implementing the protothread to be scheduled

Definition at line 271 of file pt.h.

6.1.2.7 #define PT_SPAWN(pt, child, thread)

Spawn a child protothread and wait until it exits.

This macro spawns a child protothread and waits until it exits. The macro can only be used within a protothread.

Parameters:

pt A pointer to the protothread control structure.

child A pointer to the child protothread's control structure.

thread The child protothread with arguments

Definition at line 206 of file pt.h.

6.1.2.8 #define PT_THREAD(name_args)

Declaration of a protothread.

This macro is used to declare a protothread. All protothreads must be declared with this macro.

Parameters:

name_args The name and arguments of the C function implementing the protothread.

Definition at line 100 of file pt.h.

6.1.2.9 #define PT_WAIT_THREAD(pt, thread)

Block and wait until a child protothread completes.

This macro schedules a child protothread. The current protothread will block until the child protothread completes.

Note:

The child protothread must be manually initialized with the PT_INIT() function before this function is used.

Parameters:

pt A pointer to the protothread control structure.

thread The child protothread with arguments

See also:

PT_SPAWN()

Definition at line 192 of file pt.h.

6.1.2.10 #define PT_WAIT_UNTIL(pt, condition)

Block and wait until condition is true.

This macro blocks the protothread until the specified condition is true.

Parameters:

pt A pointer to the protothread control structure.

condition The condition.

Definition at line 148 of file pt.h.

6.1.2.11 #define PT_WAIT_WHILE(pt, cond)

Block and wait while condition is true.

This function blocks and waits while condition is true. See PT_WAIT_UNTIL().

Parameters:

pt A pointer to the protothread control structure.

cond The condition.

Definition at line 167 of file pt.h.

6.1.2.12 #define PT_YIELD(pt)

Yield from the current protothread.

This function will yield the protothread, thereby allowing other processing to take place in the system.

Parameters:

pt A pointer to the protothread control structure.

Definition at line 290 of file pt.h.

6.1.2.13 #define PT_YIELD_UNTIL(pt, cond)

Yield from the protothread until a condition occurs.

Parameters:

pt A pointer to the protothread control structure.

cond The condition.

This function will yield the protothread, until the specified condition evaluates to true.

Definition at line 310 of file pt.h.

6.2 Examples

6.2.1 A small example

This first example shows a very simple program: two protothreads waiting for each other to toggle two flags. The code illustrates how to write protothreads code, how to initialize protothreads, and how to schedule them.

```
/**
 * This is a very small example that shows how to use
 * protothreads. The program consists of two protothreads that wait
 * for each other to toggle a variable.
 */

/* We must always include pt.h in our protothreads code. */
#include "pt.h"

#include <stdio.h> /* For printf(). */

/* Two flags that the two protothread functions use. */
static int protothread1_flag, protothread2_flag;

/**
 * The first protothread function. A protothread function must always
 * return an integer, but must never explicitly return - returning is
 * performed inside the protothread statements.
 *
 * The protothread function is driven by the main loop further down in
 * the code.
 */
static int
protothread1(struct pt *pt)
{
  /* A protothread function must begin with PT_BEGIN() which takes a
     pointer to a struct pt. */
  PT_BEGIN(pt);

  /* We loop forever here. */
  while(1) {
    /* Wait until the other protothread has set its flag. */
    PT_WAIT_UNTIL(pt, protothread2_flag != 0);
    printf("Protothread 1 running\n");

    /* We then reset the other protothread's flag, and set our own
       flag so that the other protothread can run. */
    protothread2_flag = 0;
    protothread1_flag = 1;

    /* And we loop. */
  }

  /* All protothread functions must end with PT_END() which takes a
     pointer to a struct pt. */
  PT_END(pt);
}

/**
 * The second protothread function. This is almost the same as the
 * first one.
 */
static int
protothread2(struct pt *pt)
{
  PT_BEGIN(pt);

  while(1) {
    /* Let the other protothread run. */
    protothread2_flag = 1;

    /* Wait until the other protothread has set its flag. */
    PT_WAIT_UNTIL(pt, protothread1_flag != 0);
    printf("Protothread 2 running\n");

    /* We then reset the other protothread's flag. */
    protothread1_flag = 0;
```

```
    /* And we loop. */
  }
  PT_END(pt);
}

/**
 * Finally, we have the main loop. Here is where the protothreads are
 * initialized and scheduled. First, however, we define the
 * protothread state variables pt1 and pt2, which hold the state of
 * the two protothreads.
 */
static struct pt pt1, pt2;
int
main(void)
{
  /* Initialize the protothread state variables with PT_INIT(). */
  PT_INIT(&pt1);
  PT_INIT(&pt2);

  /*
   * Then we schedule the two protothreads by repeatedly calling their
   * protothread functions and passing a pointer to the protothread
   * state variables as arguments.
   */
  while(1) {
    protothread1(&pt1);
    protothread2(&pt2);
  }
}
```

6.2.2 A code-lock

This example shows how to implement a simple code lock - the kind of device that is placed next to doors and that you have to push a four digit number into in order to unlock the door.

The code lock waits for key presses from a numeric keyboard and if the correct code is entered, the lock is unlocked. There is a maximum time of one second between each key press, and after the correct code has been entered, no more keys must be pressed for 0.5 seconds before the lock is opened.

```
/*
 * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

```
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the protothreads library.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 * $Id: example-codelock.c,v 1.5 2005/10/06 07:57:08 adam Exp $
 */

/*
 *
 * This example shows how to implement a simple code lock. The code
 * lock waits for key presses from a numeric keyboard and if the
 * correct code is entered, the lock is unlocked. There is a maximum
 * time of one second between each key press, and after the correct
 * code has been entered, no more keys must be pressed for 0.5 seconds
 * before the lock is opened.
 *
 * This is an example that shows two things:
 * - how to implement a code lock key input mechanism, and
 * - how to implement a sequential timed routine.
 *
 * The program consists of two protothreads, one that implements the
 * code lock reader and one that implements simulated keyboard input.
 *
 *
 */

#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#include <sys/time.h>
#endif
#include <stdio.h>

#include "pt.h"

/*---------------------------------------------------------------------------*/
/*
 * The following definitions are just for the simple timer library
 * used in this example. The actual implementation of the functions
 * can be found at the end of this file.
 */
struct timer { int start, interval; };
static int  timer_expired(struct timer *t);
static void timer_set(struct timer *t, int usecs);
/*---------------------------------------------------------------------------*/
/*
 * This example uses two timers: one for the code lock protothread and
 * one for the simulated key input protothread.
 */
static struct timer codelock_timer, input_timer;
/*---------------------------------------------------------------------------*/
/*
 * This is the code that has to be entered.
 */
static const char code[4] = {'1', '4', '2', '3'};
/*---------------------------------------------------------------------------*/
/*
 * This example has two protothread and therefor has two protothread
 * control structures of type struct pt. These are initialized with
 * PT_INIT() in the main() function below.
 */
static struct pt codelock_pt, input_pt;
/*---------------------------------------------------------------------------*/
```

```
/*
 * The following code implements a simple key input. Input is made
 * with the press_key() function, and the function key_pressed()
 * checks if a key has been pressed. The variable "key" holds the
 * latest key that was pressed. The variable "key_pressed_flag" is set
 * when a key is pressed and cleared when a key press is checked.
 */
static char key, key_pressed_flag;

static void
press_key(char k)
{
  printf("--- Key '%c' pressed\n", k);
  key = k;
  key_pressed_flag = 1;
}

static int
key_pressed(void)
{
  if(key_pressed_flag != 0) {
    key_pressed_flag = 0;
    return 1;
  }
  return 0;
}
/*---------------------------------------------------------------------------*/
/*
 * Declaration of the protothread function implementing the code lock
 * logic. The protothread function is declared using the PT_THREAD()
 * macro. The function is declared with the "static" keyword since it
 * is local to this file. The name of the function is codelock_thread
 * and it takes one argument, pt, of the type struct pt.
 *
 */
static
PT_THREAD(codelock_thread(struct pt *pt))
{
  /* This is a local variable that holds the number of keys that have
   * been pressed. Note that it is declared with the "static" keyword
   * to make sure that the variable is *not* allocated on the stack.
   */
  static int keys;

  /*
   * Declare the beginning of the protothread.
   */
  PT_BEGIN(pt);

  /*
   * We'll let the protothread loop until the protothread is
   * explitiсly exited with PT_EXIT().
   */
  while(1) {

    /*
     * We'll be reading key presses until we get the right amount of
     * correct keys.
     */
    for(keys = 0; keys < sizeof(code); ++keys) {

      /*
       * If we haven't gotten any keypresses, we'll simply wait for one.
       */
      if(keys == 0) {

        /*
```

```
      * The PT_WAIT_UNTIL() function will block until the condition
      * key_pressed() is true.
      */
     PT_WAIT_UNTIL(pt, key_pressed());
   } else {

     /*
      * If the "key" variable was larger than zero, we have already
      * gotten at least one correct key press. If so, we'll not
      * only wait for the next key, but we'll also set a timer that
      * expires in one second. This gives the person pressing the
      * keys one second to press the next key in the code.
      */
     timer_set(&codelock_timer, 1000);

     /*
      * The following statement shows how complex blocking
      * conditions can be easily expressed with protothreads and
      * the PT_WAIT_UNTIL() function.
      */
     PT_WAIT_UNTIL(pt, key_pressed() || timer_expired(&codelock_timer));

     /*
      * If the timer expired, we should break out of the for() loop
      * and start reading keys from the beginning of the while(1)
      * loop instead.
      */
     if(timer_expired(&codelock_timer)) {
       printf("Code lock timer expired.\n");

       /*
        * Break out from the for() loop and start from the
        * beginning of the while(1) loop.
        */
       break;
     }
   }

   /*
    * Check if the pressed key was correct.
    */
   if(key != code[keys]) {
     printf("Incorrect key '%c' found\n", key);
     /*
      * Break out of the for() loop since the key was incorrect.
      */
     break;
   } else {
     printf("Correct key '%c' found\n", key);
   }
 }

 /*
  * Check if we have gotten all keys.
  */
 if(keys == sizeof(code)) {
   printf("Correct code entered, waiting for 500 ms before unlocking.\n");

   /*
    * Ok, we got the correct code. But to make sure that the code
    * was not just a fluke of luck by an intruder, but the correct
    * code entered by a person that knows the correct code, we'll
    * wait for half a second before opening the lock. If another
    * key is pressed during this time, we'll assume that it was a
    * fluke of luck that the correct code was entered the first
    * time.
    */
```

```
      timer_set(&codelock_timer, 500);
      PT_WAIT_UNTIL(pt, key_pressed() || timer_expired(&codelock_timer));

      /*
       * If we continued from the PT_WAIT_UNTIL() statement without
       * the timer expired, we don't open the lock.
       */
      if(!timer_expired(&codelock_timer)) {
        printf("Key pressed during final wait, code lock locked again.\n");
      } else {

        /*
         * If the timer expired, we'll open the lock and exit from the
         * protothread.
         */
        printf("Code lock unlocked.\n");
        PT_EXIT(pt);
      }
    }
  }

  /*
   * Finally, we'll mark the end of the protothread.
   */
  PT_END(pt);
}
/*---------------------------------------------------------------------------*/
/*
 * This is the second protothread in this example. It implements a
 * simulated user pressing the keys. This illustrates how a linear
 * sequence of timed instructions can be implemented with
 * protothreads.
 */
static
PT_THREAD(input_thread(struct pt *pt))
{
  PT_BEGIN(pt);

  printf("Waiting 1 second before entering first key.\n");

  timer_set(&input_timer, 1000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));
```

```
  press_key('2');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 1500);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 300);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 400);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 500);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  PT_END(pt);
}
/*---------------------------------------------------------------------------*/
/*
 * This is the main function. It initializes the two protothread
 * control structures and schedules the two protothreads. The main
 * function returns when the protothread the runs the code lock exits.
 */
int
main(void)
{
```

```
  /*
   * Initialize the two protothread control structures.
   */
  PT_INIT(&input_pt);
  PT_INIT(&codelock_pt);

  /*
   * Schedule the two protothreads until the codelock_thread() exits.
   */
  while(PT_SCHEDULE(codelock_thread(&codelock_pt))) {
    PT_SCHEDULE(input_thread(&input_pt));

    /*
     * When running this example on a multitasking system, we must
     * give other processes a chance to run too and therefore we call
     * usleep() resp. Sleep() here. On a dedicated embedded system,
     * we usually do not need to do this.
     */
#ifdef _WIN32
    Sleep(0);
#else
    usleep(10);
#endif
  }

  return 0;
}
/*---------------------------------------------------------------------------*/
/*
 * Finally, the implementation of the simple timer library follows.
 */
#ifdef _WIN32

static int clock_time(void)
{ return (int)GetTickCount(); }

#else /* _WIN32 */

static int clock_time(void)
{
  struct timeval tv;
  struct timezone tz;
  gettimeofday(&tv, &tz);
  return tv.tv_sec * 1000 + tv.tv_usec / 1000;
}

#endif /* _WIN32 */

static int timer_expired(struct timer *t)
{ return (int)(clock_time() - t->start) >= (int)t->interval; }

static void timer_set(struct timer *t, int interval)
{ t->interval = interval; t->start = clock_time(); }
/*---------------------------------------------------------------------------*/
```

6.2.3 The bounded buffer with protothread semaphores

The following example shows how to implement the bounded buffer problem using the protothreads semaphore library. The example uses three protothreads: one producer() protothread that produces items, one consumer() protothread that consumes items, and one driver_thread() that schedules the producer and consumer protothreads.

Note that there is no need for a mutex to guard the add_to_buffer() and get_from_buffer() functions because of the implicit locking semantics of protothreads - a protothread will never be preempted and will never

block except in an explicit PT_WAIT statement.

```
/*
 * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the protothreads library.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 * $Id: example-buffer.c,v 1.5 2005/10/07 05:21:33 adam Exp $
 */

#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
#include <stdio.h>

#include "pt-sem.h"

#define NUM_ITEMS 32
#define BUFSIZE 8

static int buffer[BUFSIZE];
static int bufptr;

static void
add_to_buffer(int item)
{
  printf("Item %d added to buffer at place %d\n", item, bufptr);
  buffer[bufptr] = item;
  bufptr = (bufptr + 1) % BUFSIZE;
}
static int
get_from_buffer(void)
{
  int item;
  item = buffer[bufptr];
  printf("Item %d retrieved from buffer at place %d\n",
         item, bufptr);
  bufptr = (bufptr + 1) % BUFSIZE;
```

```
  return item;
}

static int
produce_item(void)
{
  static int item = 0;
  printf("Item %d produced\n", item);
  return item++;
}

static void
consume_item(int item)
{
  printf("Item %d consumed\n", item);
}

static struct pt_sem full, empty;

static
PT_THREAD(producer(struct pt *pt))
{
  static int produced;

  PT_BEGIN(pt);

  for(produced = 0; produced < NUM_ITEMS; ++produced) {

    PT_SEM_WAIT(pt, &full);

    add_to_buffer(produce_item());

    PT_SEM_SIGNAL(pt, &empty);
  }

  PT_END(pt);
}

static
PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;

  PT_BEGIN(pt);

  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {

    PT_SEM_WAIT(pt, &empty);

    consume_item(get_from_buffer());

    PT_SEM_SIGNAL(pt, &full);
  }

  PT_END(pt);
}

static
PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;

  PT_BEGIN(pt);

  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
```

```
  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);

  PT_WAIT_THREAD(pt, producer(&pt_producer) &
                     consumer(&pt_consumer));

  PT_END(pt);
}


int
main(void)
{
  struct pt driver_pt;

  PT_INIT(&driver_pt);

  while(PT_SCHEDULE(driver_thread(&driver_pt))) {

    /*
     * When running this example on a multitasking system, we must
     * give other processes a chance to run too and therefore we call
     * usleep() resp. Sleep() here. On a dedicated embedded system,
     * we usually do not need to do this.
     */
#ifdef _WIN32
    Sleep(0);
#else
    usleep(10);
#endif
  }
  return 0;
}
```

6.3 Protothread semaphores

6.3.1 Detailed Description

This module implements counting semaphores on top of protothreads.

Semaphores are a synchronization primitive that provide two operations: "wait" and "signal". The "wait" operation checks the semaphore counter and blocks the thread if the counter is zero. The "signal" operation increases the semaphore counter but does not block. If another thread has blocked waiting for the semaphore that is signalled, the blocked thread will become runnable again.

Semaphores can be used to implement other, more structured, synchronization primitives such as monitors and message queues/bounded buffers (see below).

The following example shows how the producer-consumer problem, also known as the bounded buffer problem, can be solved using protothreads and semaphores. Notes on the program follow after the example.

```
#include "pt-sem.h"

#define NUM_ITEMS 32
#define BUFSIZE 8

static struct pt_sem mutex, full, empty;

PT_THREAD(producer(struct pt *pt))
{
  static int produced;

  PT_BEGIN(pt);
```

```
  for(produced = 0; produced < NUM_ITEMS; ++produced) {

    PT_SEM_WAIT(pt, &full);

    PT_SEM_WAIT(pt, &mutex);
    add_to_buffer(produce_item());
    PT_SEM_SIGNAL(pt, &mutex);

    PT_SEM_SIGNAL(pt, &empty);
  }

  PT_END(pt);
}

PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;

  PT_BEGIN(pt);

  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {

    PT_SEM_WAIT(pt, &empty);

    PT_SEM_WAIT(pt, &mutex);
    consume_item(get_from_buffer());
    PT_SEM_SIGNAL(pt, &mutex);

    PT_SEM_SIGNAL(pt, &full);
  }

  PT_END(pt);
}

PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;

  PT_BEGIN(pt);

  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
  PT_SEM_INIT(&mutex, 1);

  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);

  PT_WAIT_THREAD(pt, producer(&pt_producer) &
                     consumer(&pt_consumer));

  PT_END(pt);
}
```

The program uses three protothreads: one protothread that implements the consumer, one thread that implements the producer, and one protothread that drives the two other protothreads. The program uses three semaphores: "full", "empty" and "mutex". The "mutex" semaphore is used to provide mutual exclusion for the buffer, the "empty" semaphore is used to block the consumer is the buffer is empty, and the "full" semaphore is used to block the producer is the buffer is full.

The "driver_thread" holds two protothread state variables, "pt_producer" and "pt_consumer". It is important to note that both these variables are declared as *static*. If the static keyword is not used, both variables are stored on the stack. Since protothreads do not store the stack, these variables may be overwritten during a protothread wait operation. Similarly, both the "consumer" and "producer" protothreads declare their local variables as static, to avoid them being stored on the stack.

Files

- file pt-sem.h

  Couting semaphores implemented on protothreads.

Data Structures

- struct pt_sem

Defines

- #define PT_SEM_INIT(s, c)

  Initialize a semaphore.

- #define PT_SEM_WAIT(pt, s)

  Wait for a semaphore.

- #define PT_SEM_SIGNAL(pt, s)

  Signal a semaphore.

6.3.2 Define Documentation

6.3.2.1 #define PT_SEM_INIT(s, c)

Initialize a semaphore.

This macro initializes a semaphore with a value for the counter. Internally, the semaphores use an "unsigned int" to represent the counter, and therefore the "count" argument should be within range of an unsigned int.

Parameters:

s (struct pt_sem *) A pointer to the pt_sem struct representing the semaphore

c (unsigned int) The initial count of the semaphore.

Definition at line 183 of file pt-sem.h.

6.3.2.2 #define PT_SEM_SIGNAL(pt, s)

Signal a semaphore.

This macro carries out the "signal" operation on the semaphore. The signal operation increments the counter inside the semaphore, which eventually will cause waiting protothreads to continue executing.

Parameters:

pt (struct pt *) A pointer to the protothread (struct pt) in which the operation is executed.

s (struct pt_sem *) A pointer to the pt_sem struct representing the semaphore

Definition at line 222 of file pt-sem.h.

6.3.2.3 #define PT_SEM_WAIT(pt, s)

Wait for a semaphore.

This macro carries out the "wait" operation on the semaphore. The wait operation causes the protothread to block while the counter is zero. When the counter reaches a value larger than zero, the protothread will continue.

Parameters:

pt (struct pt ∗) A pointer to the protothread (struct pt) in which the operation is executed.

s (struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore

Definition at line 201 of file pt-sem.h.

6.4 Local continuations

6.4.1 Detailed Description

Local continuations form the basis for implementing protothreads.

A local continuation can be *set* in a specific function to capture the state of the function. After a local continuation has been set can be *resumed* in order to restore the state of the function at the point where the local continuation was set.

Files

- file lc.h

  Local continuations.

- file lc-switch.h

  Implementation of local continuations based on switch() statment.

- file lc-addrlabels.h

  Implementation of local continuations based on the "Labels as values" feature of gcc.

Defines

- #define LC_INIT(lc)

  Initialize a local continuation.

- #define LC_SET(lc)

  Set a local continuation.

- #define LC_RESUME(lc)

  Resume a local continuation.

- #define LC_END(lc)

  Mark the end of local continuation usage.

- #define LC_INIT(s) s = 0;

- #define LC_RESUME(s) switch(s) { case 0:
- #define LC_SET(s) s = __LINE__; case __LINE__:
- #define LC_END(s) }
- #define LC_INIT(s) s = NULL
- #define LC_RESUME(s)
- #define LC_CONCAT2(s1, s2) s1##s2
- #define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)
- #define LC_SET(s)
- #define LC_END(s)

Typedefs

- typedef unsigned short lc_t

  The local continuation type.

- typedef void * lc_t

6.4.2 Define Documentation

6.4.2.1 #define LC_END(lc)

Mark the end of local continuation usage.

The end operation signifies that local continuations should not be used any more in the function. This operation is not needed for most implementations of local continuation, but is required by a few implementations.

Definition at line 108 of file lc.h.

6.4.2.2 #define LC_INIT(lc)

Initialize a local continuation.

This operation initializes the local continuation, thereby unsetting any previously set continuation state.

Definition at line 71 of file lc.h.

6.4.2.3 #define LC_RESUME(lc)

Resume a local continuation.

The resume operation resumes a previously set local continuation, thus restoring the state in which the function was when the local continuation was set. If the local continuation has not been previously set, the resume operation does nothing.

Definition at line 96 of file lc.h.

6.4.2.4 #define LC_SET(lc)

Set a local continuation.

The set operation saves the state of the function at the point where the operation is executed. As far as the set operation is concerned, the state of the function does **not** include the call-stack or local (automatic) variables, but only the program counter and such CPU registers that needs to be saved.

Definition at line 84 of file lc.h.

7 The Protothreads Library 1.4 Data Structure Documentation

7.1 pt Struct Reference

7.1.1 Detailed Description

Definition at line 54 of file pt.h.

Data Fields

- lc_t lc

7.2 pt_sem Struct Reference

7.2.1 Detailed Description

Definition at line 165 of file pt-sem.h.

Data Fields

- unsigned int count

8 The Protothreads Library 1.4 File Documentation

8.1 lc-addrlabels.h File Reference

8.1.1 Detailed Description

Implementation of local continuations based on the "Labels as values" feature of gcc.

Author:
Adam Dunkels <adam@sics.se>

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label.

For more information, see the GCC documentation: http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values

Definition in file lc-addrlabels.h.

Defines

- #define LC_INIT(s) s = NULL
- #define LC_RESUME(s)
- #define LC_CONCAT2(s1, s2) s1##s2
- #define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)
- #define LC_SET(s)
- #define LC_END(s)

Typedefs

- typedef void * lc_t

8.2 lc-switch.h File Reference

8.2.1 Detailed Description

Implementation of local continuations based on switch() statment.

Author:
Adam Dunkels <adam@sics.se>

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statmenets.

This implementation borrows heavily from Simon Tatham's coroutines implementation in C: http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

Definition in file lc-switch.h.

Defines

- #define LC_INIT(s) s = 0;
- #define LC_RESUME(s) switch(s) { case 0:
- #define LC_SET(s) s = __LINE__; case __LINE__:
- #define LC_END(s) }

Typedefs

- typedef unsigned short lc_t

  The local continuation type.

8.3 lc.h File Reference

8.3.1 Detailed Description

Local continuations.

Author:
Adam Dunkels <adam@sics.se>

Definition in file lc.h.

```
#include "lc-switch.h"
```

Defines

- #define LC_INIT(lc)
  Initialize a local continuation.
- #define LC_SET(lc)
  Set a local continuation.
- #define LC_RESUME(lc)
  Resume a local continuation.
- #define LC_END(lc)
  Mark the end of local continuation usage.

8.4 pt-sem.h File Reference

8.4.1 Detailed Description

Couting semaphores implemented on protothreads.

Author:
Adam Dunkels <adam@sics.se>

Definition in file pt-sem.h.

```
#include "pt.h"
```

Data Structures

- struct pt_sem

Defines

- #define PT_SEM_INIT(s, c)
  Initialize a semaphore.
- #define PT_SEM_WAIT(pt, s)
  Wait for a semaphore.
- #define PT_SEM_SIGNAL(pt, s)
  Signal a semaphore.

8.5 pt.h File Reference

8.5.1 Detailed Description

Protothreads implementation.

Author:
Adam Dunkels <adam@sics.se>

Definition in file pt.h.

```
#include "lc.h"
```

Data Structures

- struct pt

Initialization

- #define PT_INIT(pt)

  Initialize a protothread.

Declaration and definition

- #define PT_THREAD(name_args)

  Declaration of a protothread.

- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the C function implementing the protothread.

- #define PT_END(pt)

  Declare the end of a protothread.

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.

Calling a protothread

- #define PT_SCHEDULE(f)

  Schedule a 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.

Defines

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

Index