Protothreads

The Protothreads Library 1.0 Reference Manual

February 2005

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:
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 in that the code cannot use switch() statements itself.

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 1.0 Module Index

2.1 The Protothreads Library 1.0 Modules

Here is a list of all modules:

3 The Protothreads Library 1.0 File Index

3.1 The Protothreads Library 1.0 File List

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

4 The Protothreads Library 1.0 Module Documentation

4.1 Protothreads

4.1.1 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

- groupProtothread semaphores
- groupLocal continuations

Defines

- #define PT_THREAD(name_args)

  Declaration of a protothread.

- #define PT_INIT(pt)

  Initialize a protothread.

- #define PT_BEGIN(pt)

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

- #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.

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.

- #define PT_SPAWN(pt, thread)

  Spawn a child protothread and wait until it exits.

- #define PT_RESTART(pt)

  Restart the protothread.

- #define PT_EXIT(pt)

  Exit the protothread.

- #define PT_END(pt)

  Declare the end of a protothread.

- #define PT_SCHEDULE(f)

  Schedule a protothread.

4.1.2 Define Documentation

4.1.2.1 #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.

Example:

```
PT_THREAD(producer(struct pt *p, int event)) {
  PT_BEGIN(p);
  while(1) {
    PT_WAIT_UNTIL(event == CONSUMED || event == DROPPED);
    produce();
    PT_WAIT_UNTIL(event == PRODUCED);
  }

  PT_END(p);
}
```

4.1.2.2 #define PT_END(pt)

Declare the end of a protothread.

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

Parameters:

pt A pointer to the protothread control structure.

4.1.2.3 #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.

4.1.2.4 #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.

Example:

```
void main(void) {
  struct pt p;
  int event;

  PT_INIT(&p);
  while(PT_SCHEDULE(consumer(&p, event))) {
    event = get_event();
  }
}
```

4.1.2.5 #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.

4.1.2.6 #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.

Example

```
void main(void) {
  struct pt p;
  int event;

  PT_INIT(&p);
  while(PT_SCHEDULE(consumer(&p, event))) {
    event = get_event();
  }
}
```

Parameters:

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

4.1.2.7 #define PT_SPAWN(pt, 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.

thread The child protothread with arguments

4.1.2.8 #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.

Example:

```
PT_THREAD(consumer(struct pt *p, int event)) {
  PT_BEGIN(p);
  while(1) {
    PT_WAIT_UNTIL(event == AVAILABLE);
    consume();
    PT_WAIT_UNTIL(event == CONSUMED);
    acknowledge_consumed();
  }
  PT_END(p);
}
```

Parameters:

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

4.1.2.9 #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

Example:

```
PT_THREAD(child(struct pt *p, int event)) {
  PT_BEGIN(p);

  PT_WAIT_UNTIL(event == EVENT1);

  PT_END(p);
}

PT_THREAD(parent(struct pt *p, struct pt *child_pt, int event)) {
  PT_BEGIN(p);

  PT_INIT(child_pt);

  PT_WAIT_THREAD(p, child(child_pt, event));

  PT_END(p);
}
```

4.1.2.10 #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.

Example:

```
PT_THREAD(seconds(struct pt *p)) {
  PT_BEGIN(p);

  PT_WAIT_UNTIL(p, time >= 2 * SECOND);
  printf("Two seconds have passed\n");

  PT_END(p);
}
```

4.1.2.11 #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.

4.2 Protothread semaphores

4.2.1 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.

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.

4.2.2 Define Documentation

4.2.2.1 #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.

4.2.2.2 #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

4.2.2.3 #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

4.3 Local continuations

4.3.1 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.

Typedefs

- typedef unsigned short lc_t

  The local continuation type.

4.3.2 Define Documentation

4.3.2.1 #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.

4.3.2.2 #define LC_INIT(lc)

Initialize a local continuation.

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

4.3.2.3 #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.

4.3.2.4 #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.

5 The Protothreads Library 1.0 File Documentation

5.1 lc-addrlabels.h File Reference

5.1.1 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.html

5.2 lc-switch.h File Reference

5.2.1 Detailed Description

Implementation of local continuations based on switch() statment.

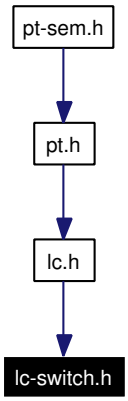
Author:
Adam Dunkels <adam@sics.se>

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

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef unsigned short lc_t
  The local continuation type.

5.3 lc.h File Reference

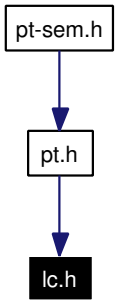
5.3.1 Detailed Description

Local continuations.

Author:
Adam Dunkels <adam@sics.se>

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

This graph shows which files directly or indirectly include this file:



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.

5.4 pt-sem.h File Reference

5.4.1 Detailed Description

Couting semaphores implemented on protothreads.

Author:
Adam Dunkels <adam@sics.se>

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

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.

5.5 pt.h File Reference

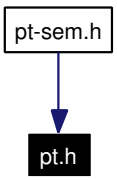
5.5.1 Detailed Description

Protothreads implementation.

Author:
Adam Dunkels <adam@sics.se>

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

This graph shows which files directly or indirectly include this file:



Defines

- #define PT_THREAD(name_args)
  Declaration of a protothread.
- #define PT_INIT(pt)
  Initialize a protothread.
- #define PT_BEGIN(pt)
  Declare the start of a protothread inside the C function implementing the protothread.
- #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.

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.

- #define PT_SPAWN(pt, thread)

  Spawn a child protothread and wait until it exits.

- #define PT_RESTART(pt)

  Restart the protothread.

- #define PT_EXIT(pt)

  Exit the protothread.

- #define PT_END(pt)

  Declare the end of a protothread.

- #define PT_SCHEDULE(f)

  Schedule a protothread.

Index