Explaining Protothreads and Their Application in Embedded Encryption Systems

Protothreads are a programming abstraction designed to bring the convenience of thread-like, sequential code to memory-constrained embedded systems, without the heavy memory overhead of traditional multi-threading. Below, you'll find a deeper explanation of how protothreads work, how they are configured, and how they are applied to build a concurrent encryption system that interfaces with a Python GUI, processes data, and updates a hardware display.

What Are Protothreads?

- **Definition:** Protothreads are extremely lightweight, stackless threads (here- coroutines) that enable cooperative multitasking in C, especially for embedded systems 9.
- **How They Work:** Each protothread is implemented as a C function using macros like `PT_BEGIN`, `PT_END`, `PT_YIELD`, and `PT_WAIT_UNTIL`. These macros use a technique called Duff's device to save the thread's position, allowing the function to pause and resume without needing a dedicated stack.
- **Advantages:**
  - **Minimal memory use:** Only 1–2 bytes per thread, compared to hundreds for a traditional thread stack 4.
  - **Simple sequential code:** Lets you write code in a linear, readable way instead of complex state machines 7.
  - **Cooperative scheduling:** Threads yield control voluntarily, so there's no preemption or context switching overhead 10.

Limitation: Local (non-static) variables do not retain their values across yields. Use static or global variables for persistent state.

How Are Protothreads Configured?

Dependencies and Setup:

- **Header:** Include the protothread library, e.g., `#include "pt_cornell_1_3_2.h"` which uses <plib.h>.

- **Thread Structure:** Each thread is a function of type `PT_THREAD`, taking a pointer to its state struct:

```
PT_THREAD(thread_name(struct pt *pt)) {
    PT_BEGIN(pt);
    // ... thread logic ...
    PT_END(pt);
}
```

- **Thread Management:** Threads are managed in a list or array, and a scheduler repeatedly calls each thread function in a loop.

```
#define MAX_THREADS 10
struct ptx {
    struct pt pt;
    int rate;
    char (*pf)(struct pt *pt);
};
static struct ptx pt_thread_list[MAX_THREADS];
int pt_task_count = 0;
```

- **Adding Threads:** Threads are registered with a function like `pt_add(protothread_serial, 0);`12.

Scheduler Example:

```
while(1) {
    for(int i=0; i<pt_task_count; i++) {
        (pt_thread_list[i].pf)(&pt_thread_list[i].pt);
    }
}
```

This ensures each protothread gets a chance to run, yielding as needed.

How Do Protothreads Enable Concurrency in the Encryption System?

System Overview

The encryption system uses protothreads to coordinate several concurrent tasks:

- **Serial Communication:** Receives input from a Python GUI over UART.
- **Encryption Processing:** Encrypts the input string using rotor-based logic.
- **TFT Display Update:** Shows the input and encrypted output on a display.
- **Rotary Encoder Handler:** Lets the user adjust rotor positions in real time.

Each of these is implemented as a separate protothread, sharing data via buffers and flags.

Thread Examples

Serial Communication Thread:

```
static PT_THREAD(protothread_serial(struct pt *pt)) {
    PT_BEGIN(pt);
    while(1) {
        PT_YIELD_UNTIL(pt, UARTReceivedDataIsAvailable(UART2));
        int i = 0;
        while(UARTReceivedDataIsAvailable(UART2) && i < 63) {
            receive_string[i++] = UARTGetDataByte(UART2);
            PT_YIELD_TIME_msec(10);
        }
        receive_string[i] = '\0';
        uppercase_convert(receive_string);
        new_data = 1;
    }
    PT_END(pt);
}
```

- **Purpose:** Waits for serial data, reads it into a buffer, converts to uppercase, and signals the encryption thread.

Encryption Processing Thread:

```
static PT_THREAD(protothread_encrypt(struct pt *pt)) {
    PT_BEGIN(pt);
    while(1) {
        PT_WAIT_UNTIL(pt, new_data == 1);
```

```
        PT_SCHEDULE(PT_Encrypt(pt));
        new_data = 0;
        printLine(3, result, ILI9340_GREEN, ILI9340_BLACK);
    }
    PT_END(pt);
}
```

- **Purpose:** Waits for new data, runs the encryption, and updates the display.

Display Thread:

```
static PT_THREAD(protothread_display(struct pt *pt)) {
    PT_BEGIN(pt);
    while(1) {
        printLine(0, "Input:", ILI9340_WHITE, ILI9340_BLACK);
        printLine(1, receive_string, ILI9340_WHITE, ILI9340_BLACK);
        printLine(2, "Encrypted:", ILI9340_GREEN, ILI9340_BLACK);
        printLine(3, result, ILI9340_GREEN, ILI9340_BLACK);
        PT_YIELD_TIME_msec(500);
    }
    PT_END(pt);
}
```

- **Purpose:** Periodically updates the TFT display with the latest input and encrypted output.

Rotary Encoder Handler:

```
static PT_THREAD(protothread_encoder(struct pt *pt)) {
    static int last_state;
    PT_BEGIN(pt);
    while(1) {
        PT_YIELD_TIME_msec(10);
        int current = ENCODER_A;
        if(current != last_state) {
            // Update rotor offset logic
        }
        last_state = current;
    }
    PT_END(pt);
}
```

- **Purpose:** Reads hardware input to adjust rotor positions in real time.

How Do Threads Communicate?

- **Shared Buffers:** Data moves between threads using global or volatile buffers (e.g., `receive_string`, `result`).
- **Flags:** Volatile flags like `new_data` signal when new input is ready for processing.
- **Synchronization:** Use `PT_WAIT_UNTIL` to block a thread until a condition is met, ensuring safe communication without race conditions.

Debugging and Simulation

- **Intermediate Outputs:** Print statements can be added inside threads to show the state at each stage (e.g., `[SERIAL] Received: HELLO`, `[ENCRYPT] Processing: HELLO -> MJQQT`).
- **Simulation:** You can run the scheduler in a PC environment or on the target hardware to observe thread behavior and outputs.

Summary Table: Thread Roles

Thread	Function	Key Resource	Trigger/Sync
Serial Communication	Reads input from Python GUI	`receive_string`	UART data available
Encryption	Encrypts input string	`result`	`new_data` flag
Display	Updates TFT with input/output	`receive_string`, `result`	Timer or always

Encoder	Handles rotor position changes	`rotor_offsets`	Hardware change

Types of Scheduling in Embedded Systems and Their Application with Protothreads

pEmbedded systems require efficient scheduling to manage multiple tasks, especially when using lightweight concurrency models like protothreads. Here's a structured explanation of different scheduling types, their characteristics, and how they fit with protothreads in the context of an embedded encryption system.

---

1. Common Scheduling Algorithms in Embedded Systems

A. Round-Robin Scheduling

- How it works: Each task (or thread) is given a fixed time slice in a circular order. After its time slice, the next task runs, and so on.
- Pros: Simple, fair, and easy to implement. All tasks get CPU time without priority.
- Cons: Not suitable for tasks with strict deadlines or real-time requirements.

B. Rate-Controlled (Rate-Monotonic) Scheduling

- How it works: Tasks are assigned different execution rates. Some tasks run every loop, others every 2nd, 4th, 8th, or 16th loop, etc. This allows higher-frequency tasks to run more often than lower-frequency ones.
- Pros: Efficient for systems where some tasks need to run more frequently than others. Still cooperative and predictable.
- Cons: More complex to configure; not as flexible as dynamic priority systems.

C. Priority-Driven Scheduling

- How it works: Each task is assigned a priority. Higher-priority tasks can preempt lower-priority ones (preemptive), or simply run first in each cycle (cooperative).
- Pros: Ensures critical tasks get CPU time when needed. Good for real-time systems.
- Cons: Increased complexity; risk of starvation for low-priority tasks.

D. Clock-Driven (Time-Triggered) Scheduling

- How it works: Scheduling decisions are made at predetermined time instants (frames). Tasks are executed according to a static schedule.
- Pros: Highly deterministic, suitable for hard real-time systems.
- Cons: Inflexible; not ideal for systems with variable task execution times.

---

2. Cooperative vs. Preemptive Scheduling

Scheduler Type	Description	Pros	Cons
Cooperative	Tasks yield control voluntarily (e.g., via PT_YIELD). No task is forcibly interrupted.	Simple, low overhead, predictable	One misbehaving task can block others
Preemptive	Tasks can be interrupted by higher-priority tasks at any time, often via hardware timer interrupts.	Responsive, supports strict priorities	Higher overhead, more complex

Protothreads are inherently **cooperative**: tasks yield explicitly, making them predictable and efficient for resource-constrained systems.

---

3. Scheduling in Protothread-Based Systems

A. Simple Round-Robin Scheduler

- Implementation: Each protothread is called in sequence within the main loop.

```
while(1) {
    for(int i=0; i<pt_task_count; i++) {
        (pt_thread_list[i].pf)(&pt_thread_list[i].pt);
    }
```

```
}
```

- Use Case: Suitable for systems where all tasks are equally important and have similar timing requirements.

B. Rate-Controlled Scheduling

- Implementation: Each thread is assigned a rate. For example, rate 0 runs every loop, rate 1 every other loop, rate 2 every fourth loop, etc.

```
// Example from Cornell ECE4760

if (rate == 0) run every loop;
if (rate == 1) run every 2nd loop;
if (rate == 2) run every 4th loop;


// etc.
```

- Use Case: Allows mixing fast and slow tasks efficiently (e.g., display updates less often than serial input processing).

C. Priority-Based Scheduling

- Implementation: Not native to protothreads, but can be emulated by calling higher-priority threads first or more frequently, or by using a custom scheduler.
- Use Case: Useful if some tasks (like real-time sensor reading) must always run before others.

D. Event-Driven Scheduling

- Implementation: Threads are scheduled in response to events (e.g., data received, timer expired). Protothreads support this via `PT_WAIT_UNTIL` and `PT_SCHEDULE` macros.
- Use Case: Efficient for systems where tasks are mostly idle, waiting for events.

---

4. Practical Example: Rate-Controlled Round-Robin in Protothreads

Suppose you have four threads:

- Serial input (needs to run often)

- Encryption (runs when new data arrives)
- Display update (can run less frequently)
- Rotary encoder (needs regular polling)

Scheduler Example:

```
#define MAX_THREADS 10
struct ptx {
    struct pt pt;
    int rate;
    char (*pf)(struct pt *pt);
};
static struct ptx pt_thread_list[MAX_THREADS];
int pt_task_count = 0;
unsigned int loop_count = 0;

while(1) {
    loop_count++;
    for(int i=0; i<pt_task_count; i++) {
        if ((loop_count % (1 << pt_thread_list[i].rate)) == 0) {
            (pt_thread_list[i].pf)(&pt_thread_list[i].pt);
        }
    }
}
```

- **rate = 0: runs every loop**
- **rate = 1: runs every 2nd loop**
- **rate = 2: runs every 4th loop, etc.**

---

5. Summary Table: Scheduling Types

Scheduling Type	Preemptive/Cooperative	Typical Use Case	Example in Protothreads

Round-Robin	**Cooperative**	**General multitasking**	**Main loop calls each thread**
Rate-Controlled	**Cooperative**	**Mixed fast/slow tasks**	**Thread rate parameter**
Priority-Driven	**Both**	**Real-time, critical tasks**	**Call order/frequency**
Clock-Driven	**Cooperative**	**Deterministic, hard real-time**	**Static schedule, timer triggers**
Event-Driven	**Cooperative**	**Event-based systems**	**PT_WAIT_UNTIL, PT_SCHEDULE**

---

6. Key Takeaways for Embedded Encryption Systems

- Protothreads use cooperative scheduling by default, making them ideal for predictable, low-overhead multitasking on microcontrollers.
- Rate-controlled round-robin is a practical way to balance tasks with different timing needs in protothread-based systems.
- Priority and event-driven scheduling can be layered on top for more complex requirements, though with increased complexity.
- Choosing the right scheduling strategy depends on your system’s real-time needs, resource constraints, and task criticality.

Why Use Protothreads in Embedded Systems?

- **Memory Efficiency:** Enables complex, concurrent applications on microcontrollers with only a few kilobytes of RAM.
- **Code Simplicity:** Avoids the spaghetti code of state machines, making programs easier to write and maintain.
- **Real-World Use:** Ideal for applications like sensor networks, device control, and—in this case—real-time encryption with user interaction.