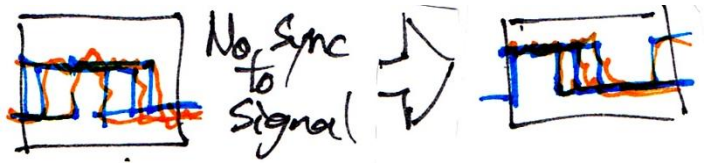
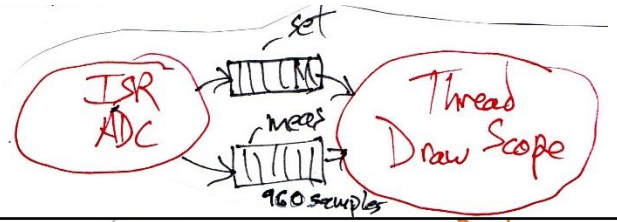
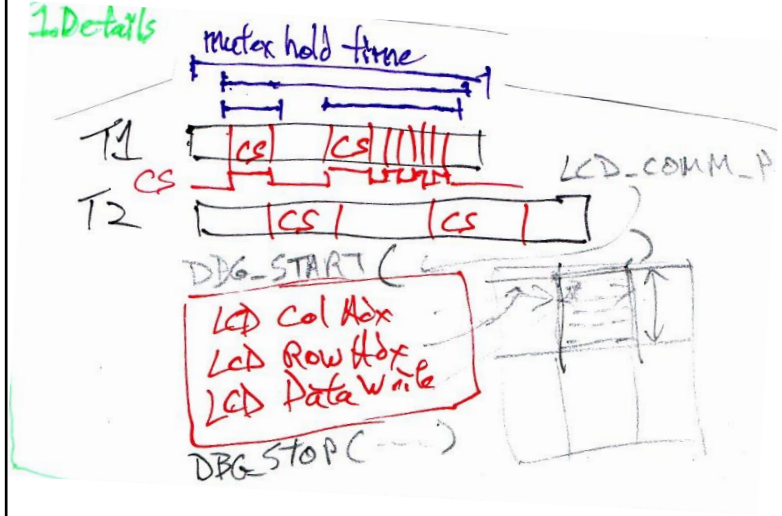
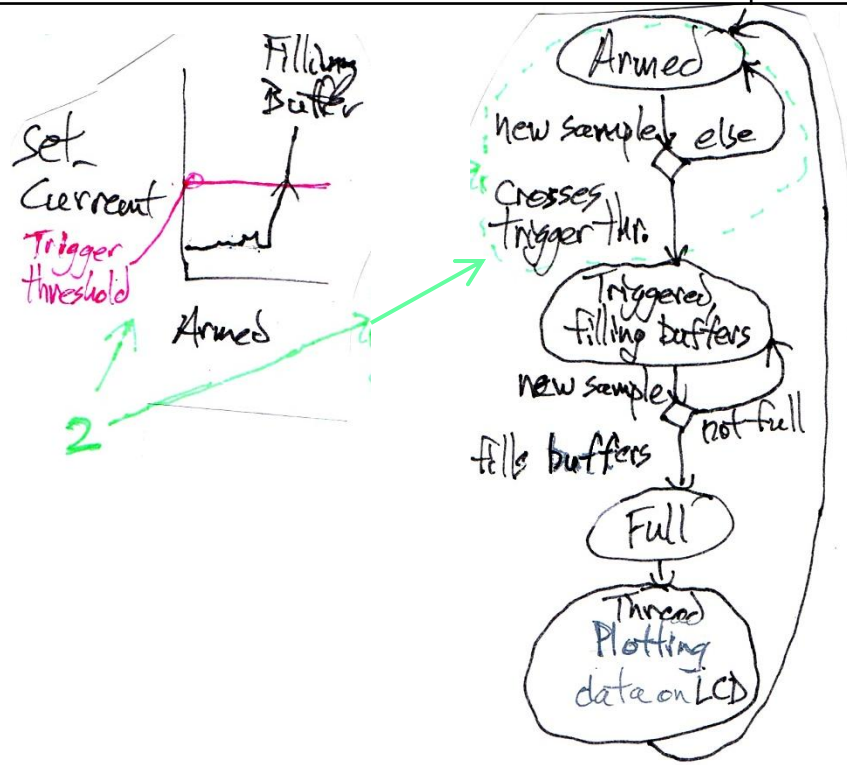
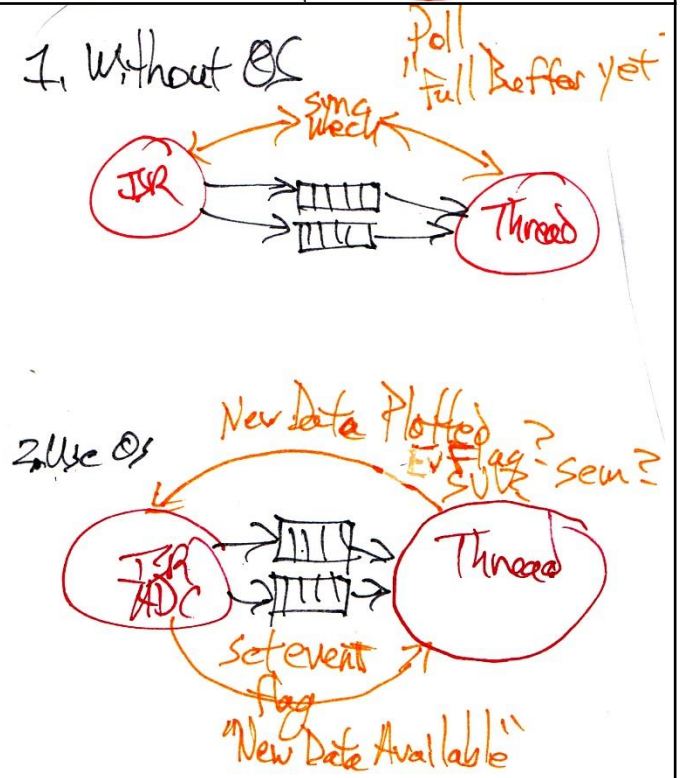
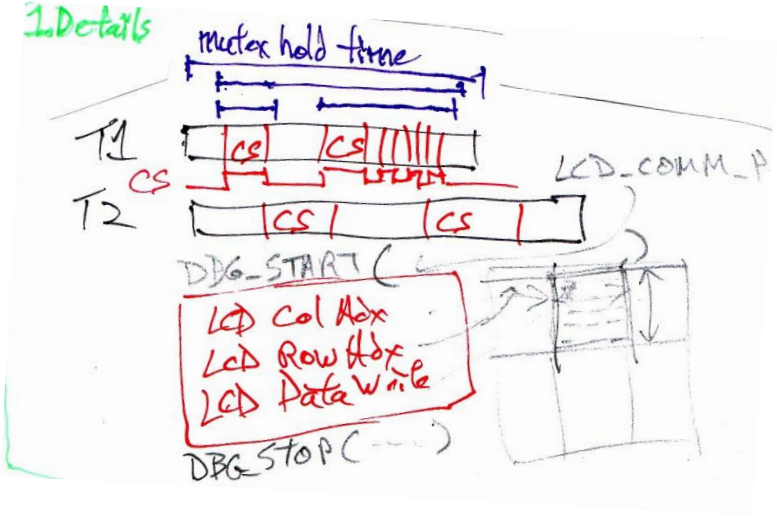
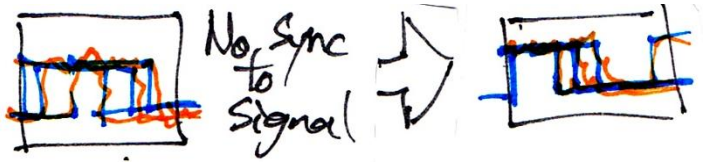
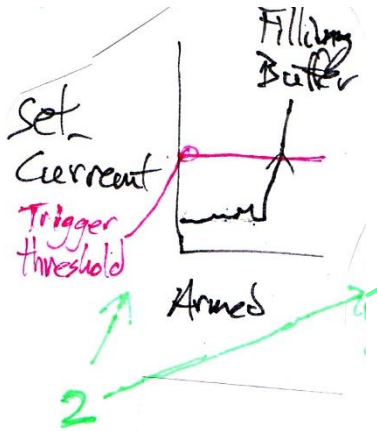
Synchronization in Project

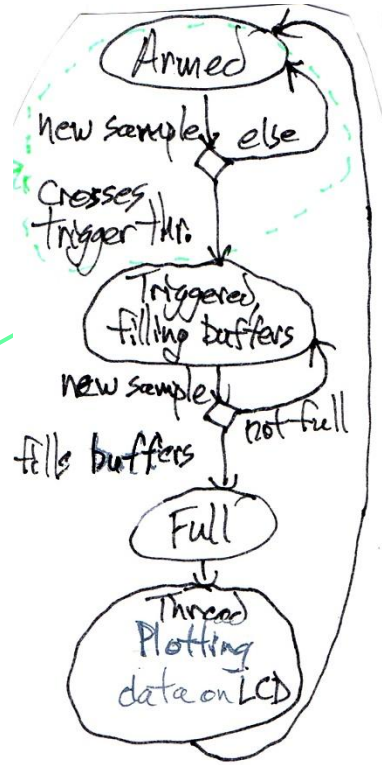
1. Safe Access to LCD	2. Scope Data Capture Triggering	3. Safe Buffer Management
Two threads access LCD, their critical section execution must be mutually exclusive (must not overlap)		
		

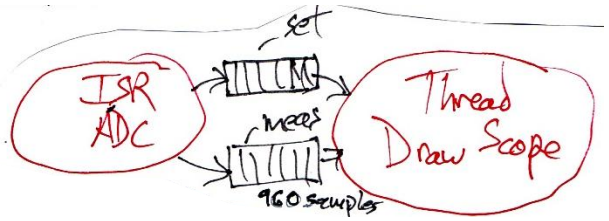
Synchronization in Project

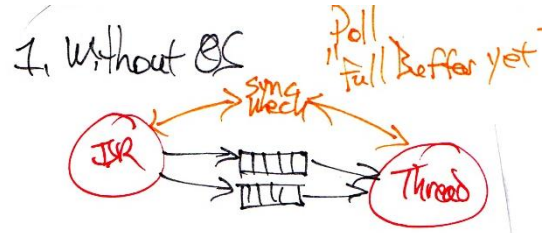


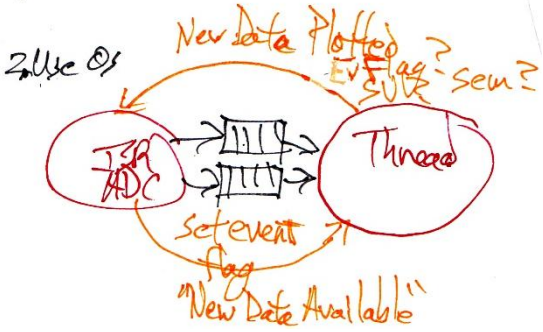








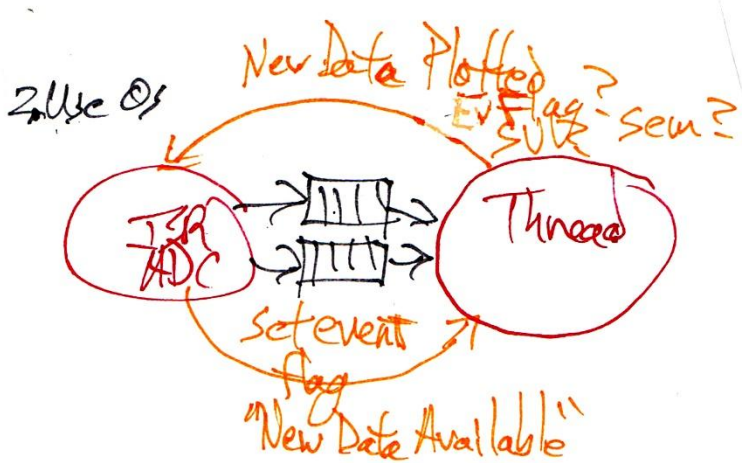




Buffer Synchronization using OS

```
ADC ISR (or a function it calls)
static int recording = 0;
static int n = 0;

read I_measured from ADC
r = osEventFlagsWait(…,ScopeArmed,…,0);
if (!(r & osFlagsError)) {
  // No error, so test for event
  if ((r == ScopeArmed) &&
      (I_setpoint > trigger_threshold)) {
     recording = 1;
     n = 0;
  }
}
if (recording == 1) {
   save I_measured to measured_buffer[n]
   save I_setpoint to setpoint_buffer[n]
   n++;
   if (n==BUFSIZE) {
      recording = 0;
      osEventFlagsSet(…,BufFull);
   }
}
Do normal control loop processing
```



ScopeArmed Event Flag

BufFull Event Flag

```
Thread Draw Waveforms
while (1) {
   osEventFlagsWait(…,BufFull,…,osWaitForever);
   DrawWaveforms();
   osEventFlagsSet(…,ScopeArmed);
}
```

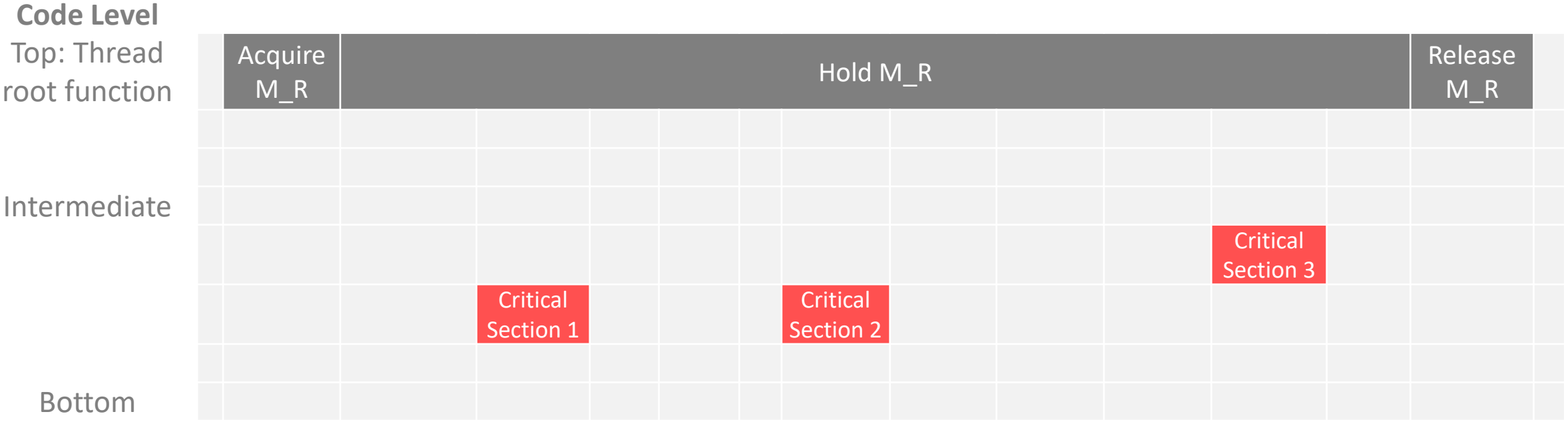
Mutex for LCD Controller Access

Critical Section Length, Blocking and Responsiveness

- Threads A, B access shared resource R, other threads C, D don't
- Access resource safely by protecting critical sections for R with mutex lock M_R
  - Threads A, B must acquire mutex M_R before starting critical section
  - Threads A, B must release mutex M_R after completing critical section

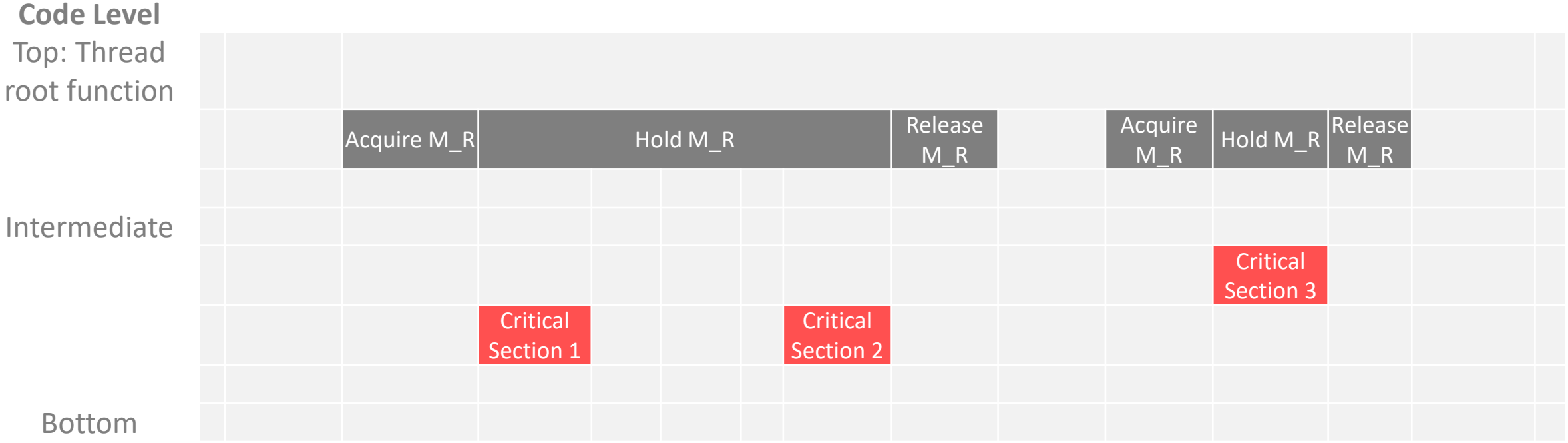
- If mutex not available, then thread blocks
  - Can monitor blocking activity on logic analyzer by adding debug output signal
    - Set before acquiring mutex, clear after acquiring mutex
  - Thread A, B holding mutex longer →
    - Longer possible blocking time for thread B, A = worse responsiveness for B, A
    - If C, D independent of A, B and R, then no impact on responsiveness for C, D

Critical Section Length, Blocking and Responsiveness



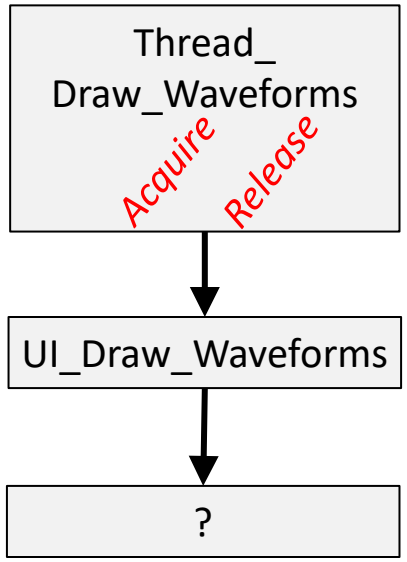
- To get best responsiveness, acquire/release mutex **immediately** before/after each critical section
  - Hard! To do this, we need to know…
    - How many critical sections in each thread?
    - Where does each critical section start and end?
  - Finding critical sections in code requires analysis (thinking, development time)
- Start simple: at a high level
  - Protect **all** critical sections for R in thread by acquiring M_R before first critical section, releasing M_R after last critical section. Better safe than sorry.
  - Mutex held conservatively -- for longer than necessary
    - Mutex was probably acquired earlier than needed
    - If 2+ critical sections, mutex was held when not needed (between critical sections)
    - Mutex was probably released later than necessary

Reducing Blocking Time

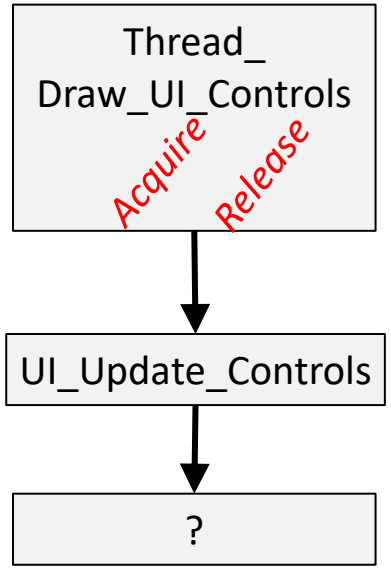


- Analyze critical sections to be able to reduce mutex holding time
- Break mutex holding time into pieces
  - Release mutex after critical section, acquire before next critical section
- Hold for less time
  - Don’t acquire so early, don’t release so late

Example from Project




```
void Thread_Draw_Waveforms(
        void * arg) {
  …
  // Regular operation
  while (1) {
    osDelayUntil(tick);
    osMutexAcquire(LCD_mutex,
      osWaitForever);
    UI_Draw_Waveforms();
    osMutexRelease(LCD_mutex);
    …
  }
}
```



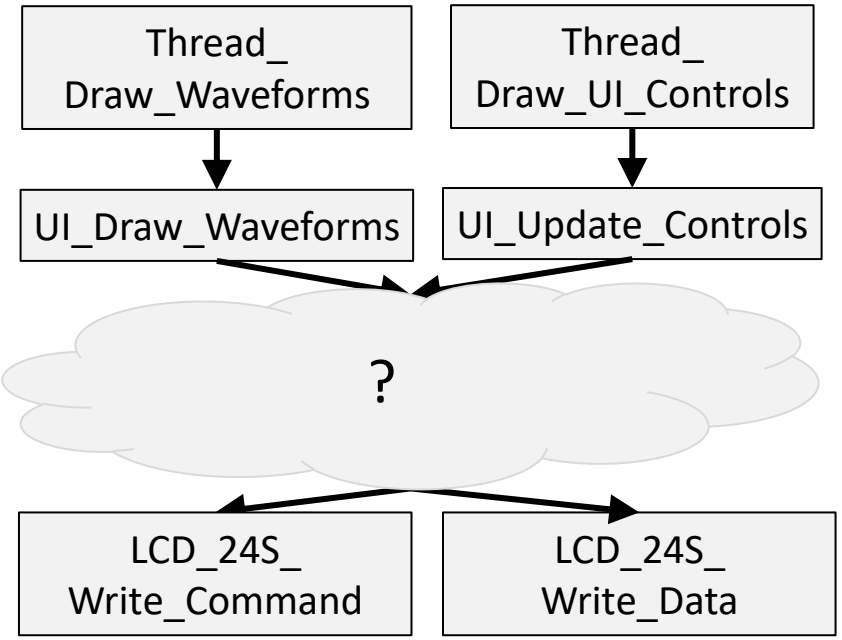

```
void Thread_Draw_UI_Controls(
        void * arg) {
  …
  // Regular operation
  while (1) {
    osDelayUntil(tick);
    osMutexAcquire(LCD_mutex,
      osWaitForever);
    UI_Update_Controls(0);
    osMutexRelease(LCD_mutex);
    …
  }
}
```

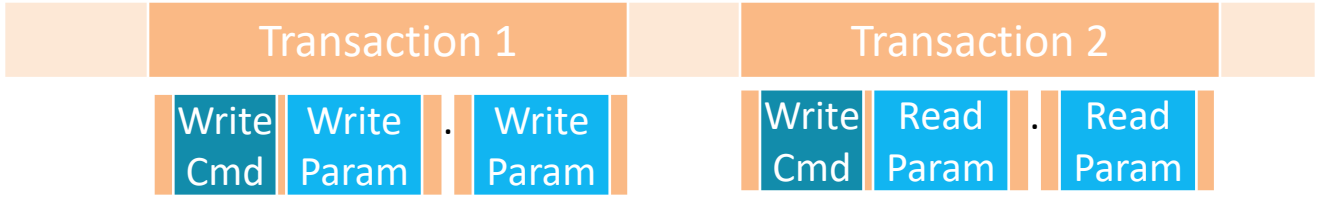
- LCD Controller interface
  - Used by two threads: Thread_Draw_Waveforms, Thread_Draw_UI_Controls
  - Want to protect with LCD_mutex

- **Where** should threads acquire and release LCD_mutex?
  - Depends on number, position and duration of critical sections
  - Easy first solution: in or near thread root function

Example from Project

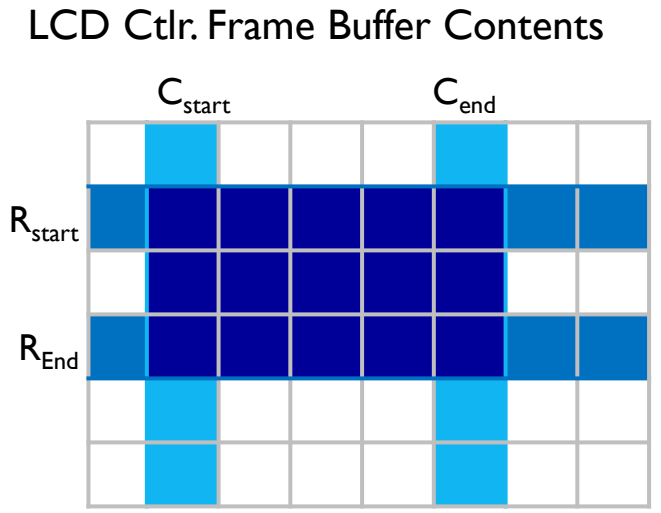
- If mutex is held too long, responsiveness suffers, must improve
  - Look for code which may be a critical section or contain one (e.g. through subroutine calls).
- What does a critical section look like?
  - “Any code that talks to the LCD Controller”
- What does that look like? How does code talk to the LCD controller?
  - Functions to access GPIO-emulated LCD interface bus
    - LCD_24S_Write_[Command|Data]
    - (Read not implemented yet)
- LCD interface transactions
  - Two types
    - Write command, write optional data parameters
    - Write command, read optional data parameters





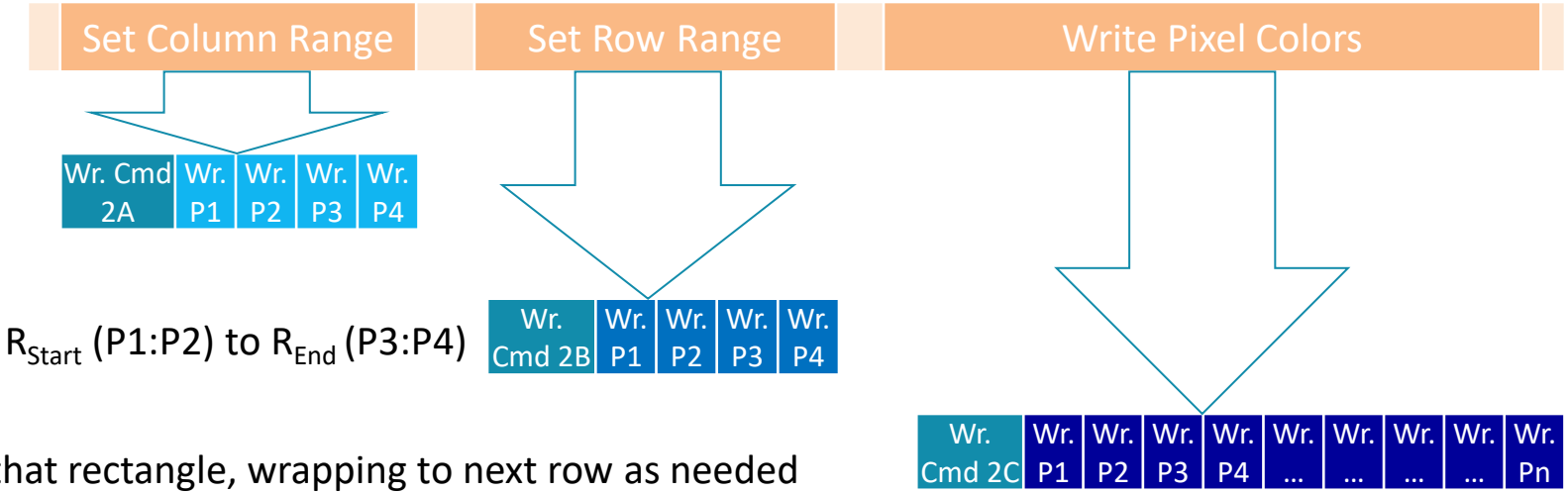
Draw a Filled Rectangle

- Writing pixels to rectangle is fundamental operation
  - Many shapes are rectangles. Pixel: 1 x 1, rectangle: n x m, H line: n x 1, V line: 1 x n
  - Character? Rectangle with font bitmap selecting fore/background color per pixel



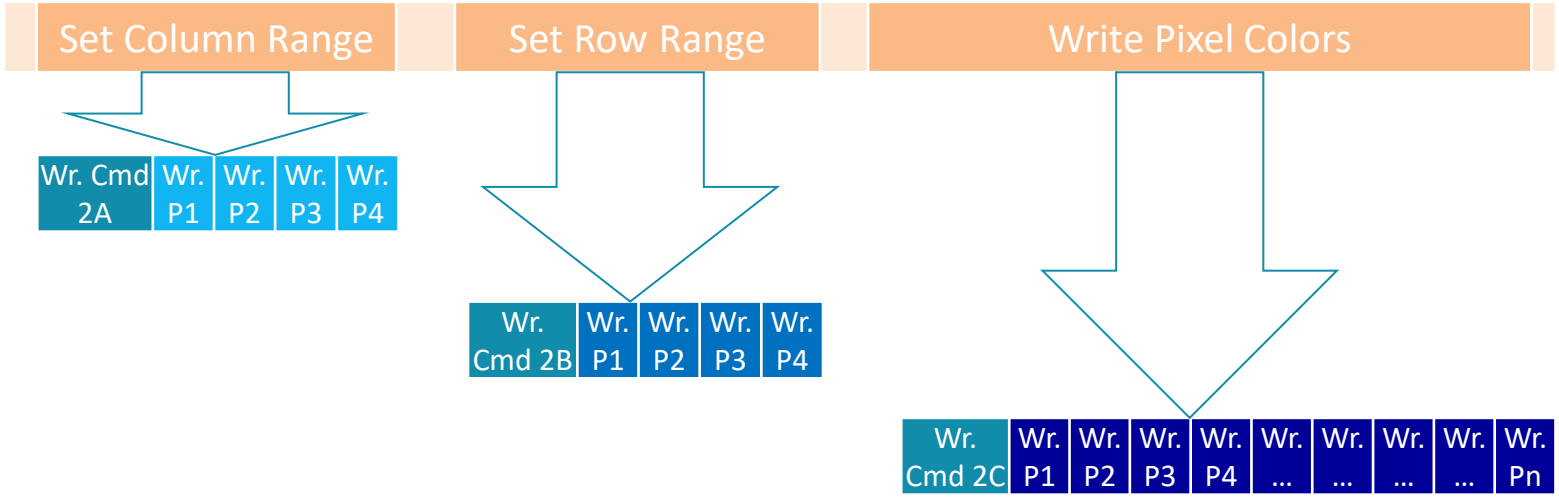
- LCD controller (ST7789) has H/W support for drawing rectangles quickly
  - "Fill this rectangle with these pixels from left to right, wrapping rows as needed."

- LCD commands used
  - Column Address Set: CASET (0x2A)
    - Set column range in frame buffer to C_{start} (params P1:P2) to C_{End} (P3:P4)
  - Row Address Set: RASET (0x2B)
    - Set row (*page*) range in frame buffer to R_{Start} (P1:P2) to R_{End} (P3:P4)
  - Memory Write: RAMWR (0x2C)
    - Write the following pixel data items to that rectangle, wrapping to next row as needed



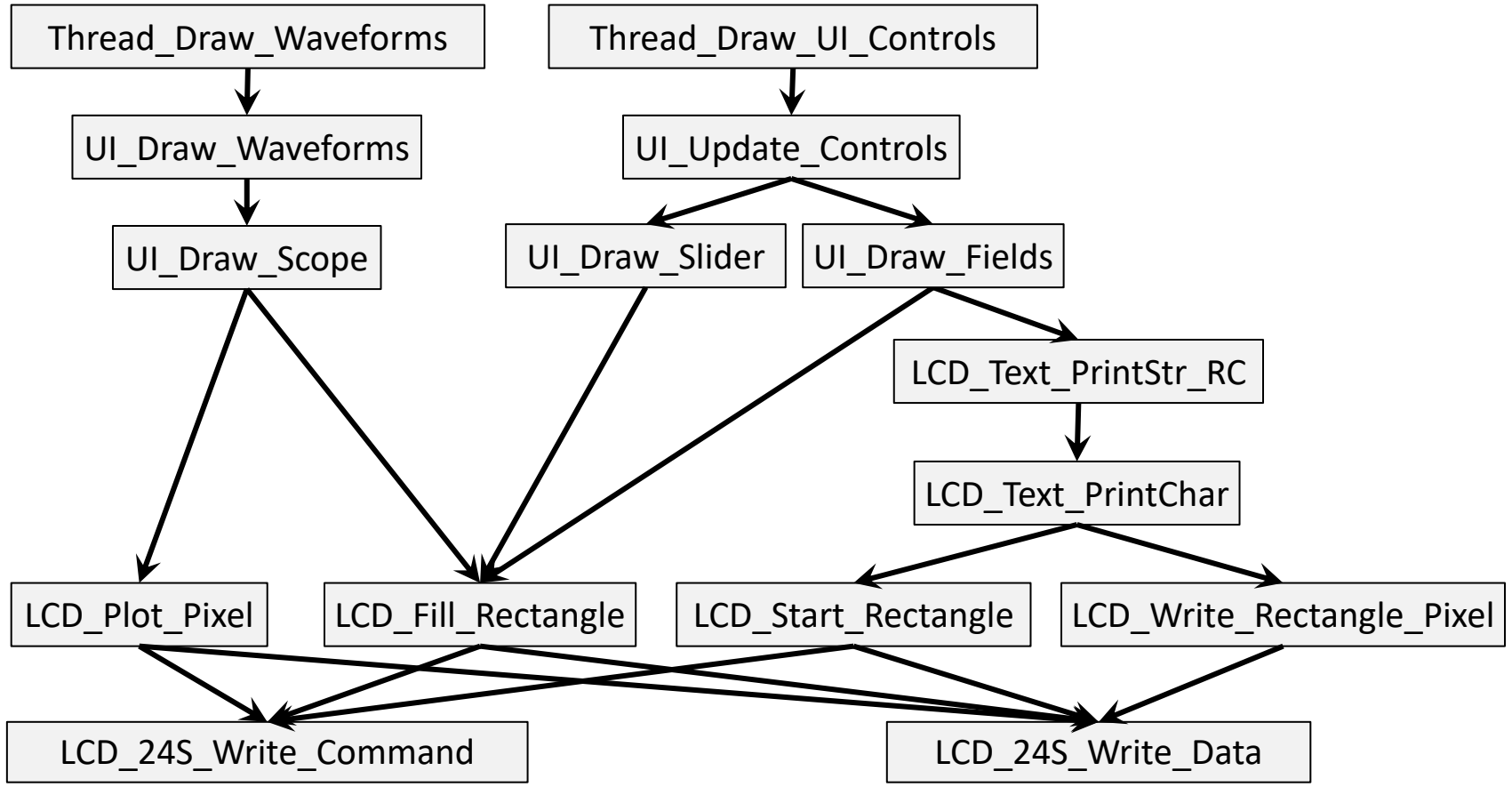
Code and Critical Section

- Example code for LCD transactions
  - Column Address Set: CASET (0x2A)
  - Row Address Set: RASET (0x2B)
  - Memory Write: RAMWR (0x2C)
- Critical section
  - Start?
  - End?



Threads and Functions Accessing LCD Controller

Gathered from Linker-generated Callgraph



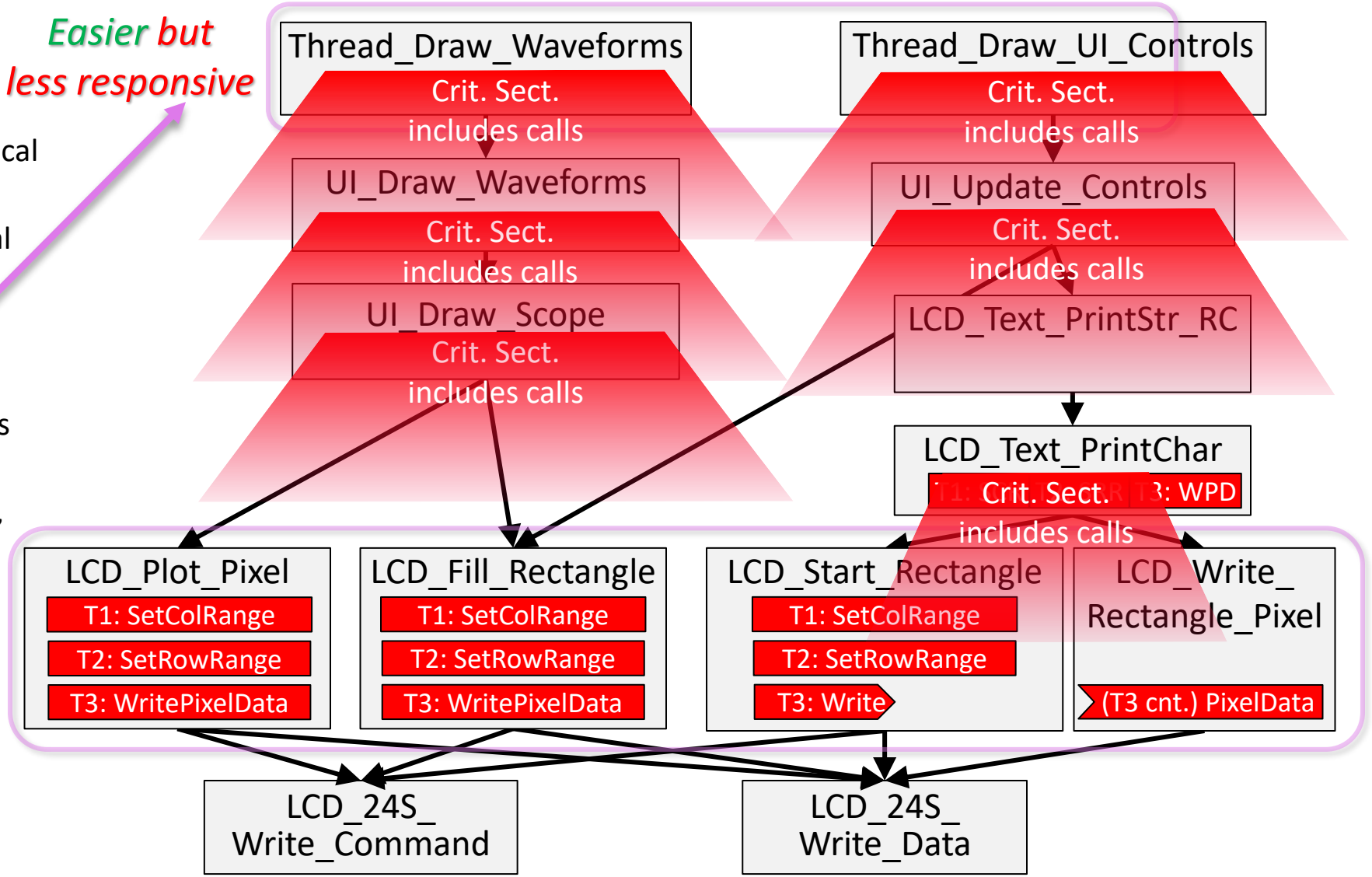
Threads and Functions Accessing LCD Controller

Gathered from Linker-generated Callgraph

- Where to acquire mutex and release mutex?
  - Must acquire mutex **before** critical section starts
  - Must release mutex **after** critical section ends
- Trade-Offs
  - Higher up in call graph: easier, but longer blocking times
  - Lower in call graph: harder (requires more analysis), shorter blocking times

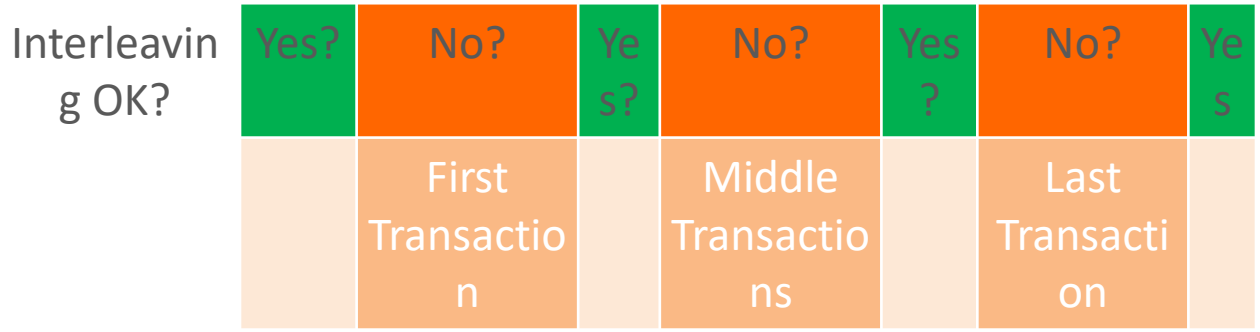
Easier but less responsive

Harder but more responsive



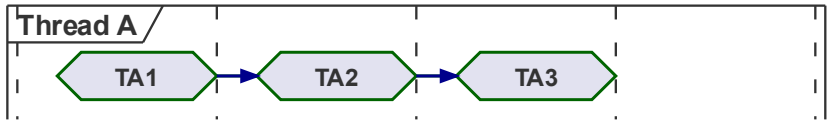
Is the Whole Transaction a Critical Section?

- OK to interleave critical sections …
  - **During** a transaction?
    - Probably usually **not OK**, but…
    - … maybe at very beginning or end, or special cases? Need to analyze code more.
  - **Between** transactions?
    - Probably usually **OK**, but …
    - … not when transactions from different threads affect shared state… meaning what? Examine LCD Controller example

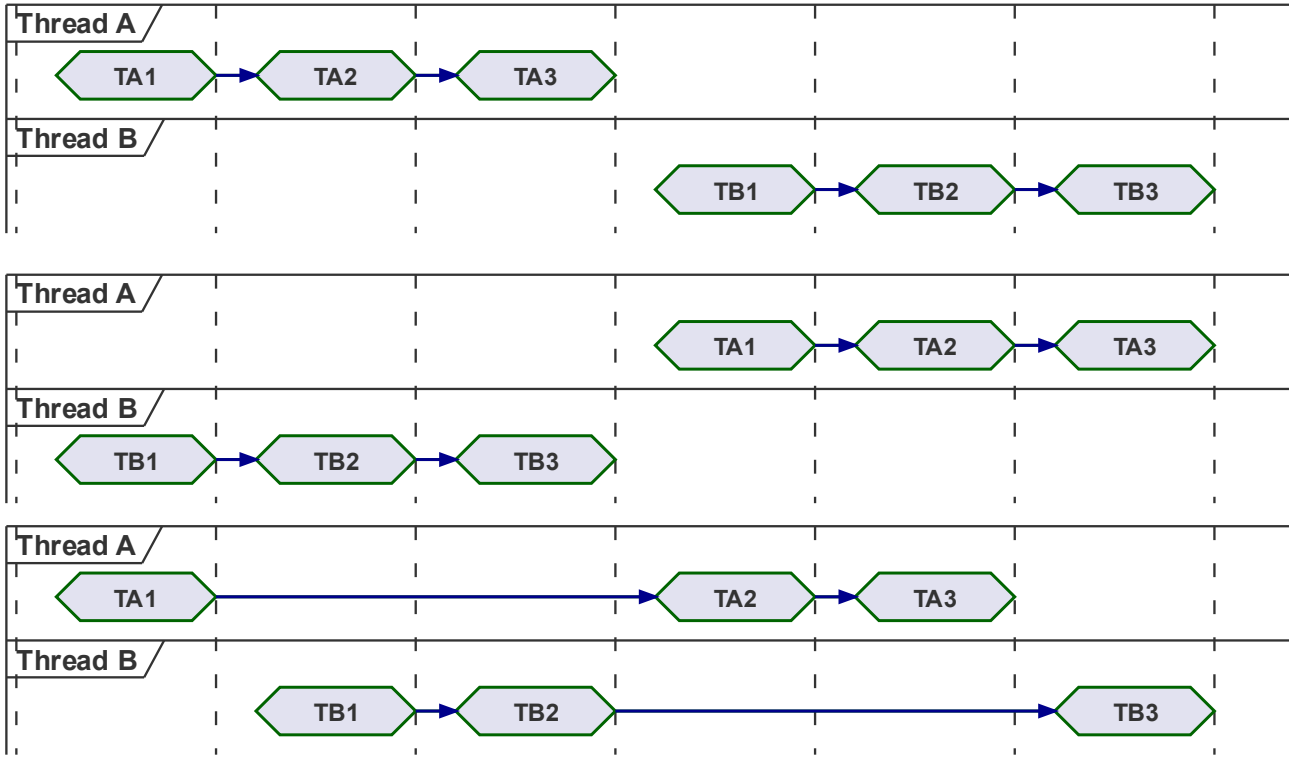


Transaction Ordering: General Case

- Within Thread *n*, assume transactions T*n*1, T*n*2, T*n*3 must be performed in order



- Does transaction ordering **between different threads** matter?
  - If not, we can interleave transactions arbitrarily, switching between threads without restrictions

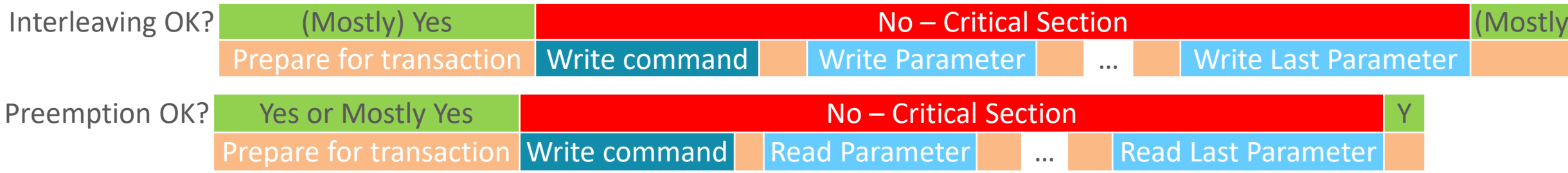


Transaction Ordering: LCD Controller

- Does transaction ordering **between different threads** matter?
  - Depends on nature of interleaved transactions – do they access shared state within the LCD Controller?
- Consider transactions to draw a rectangle (or pixel, character, horizontal/vertical line)
  - Transaction 1: Set Column Range in Frame Buffer
  - Transaction 2: Set Row Range in Frame Buffer
  - Transaction 3: Write Pixel Color Data to Frame Buffer

Critical Sections and LCD Code

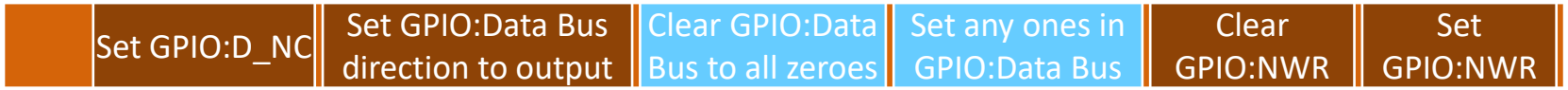
- Each transaction has a critical section.
  - Does critical section span entire transaction? Where does it start, end?

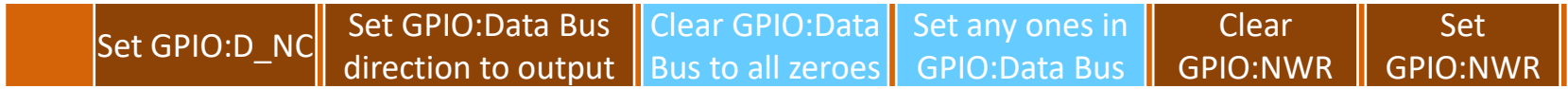


  - Critical section:
    - Starts with writing command
    - Ends when last parameter has been read or written
- Which is the **last parameter** for a transaction?
  - Release mutex after that last parameter

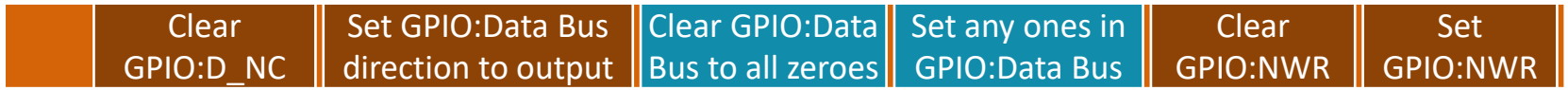
Finer-Grain Concurrency for LCD Controller Interface?

- Write to / read from the LCD controller by emulating a bus with GPIO
- Write command byte, write parameter byte(s)



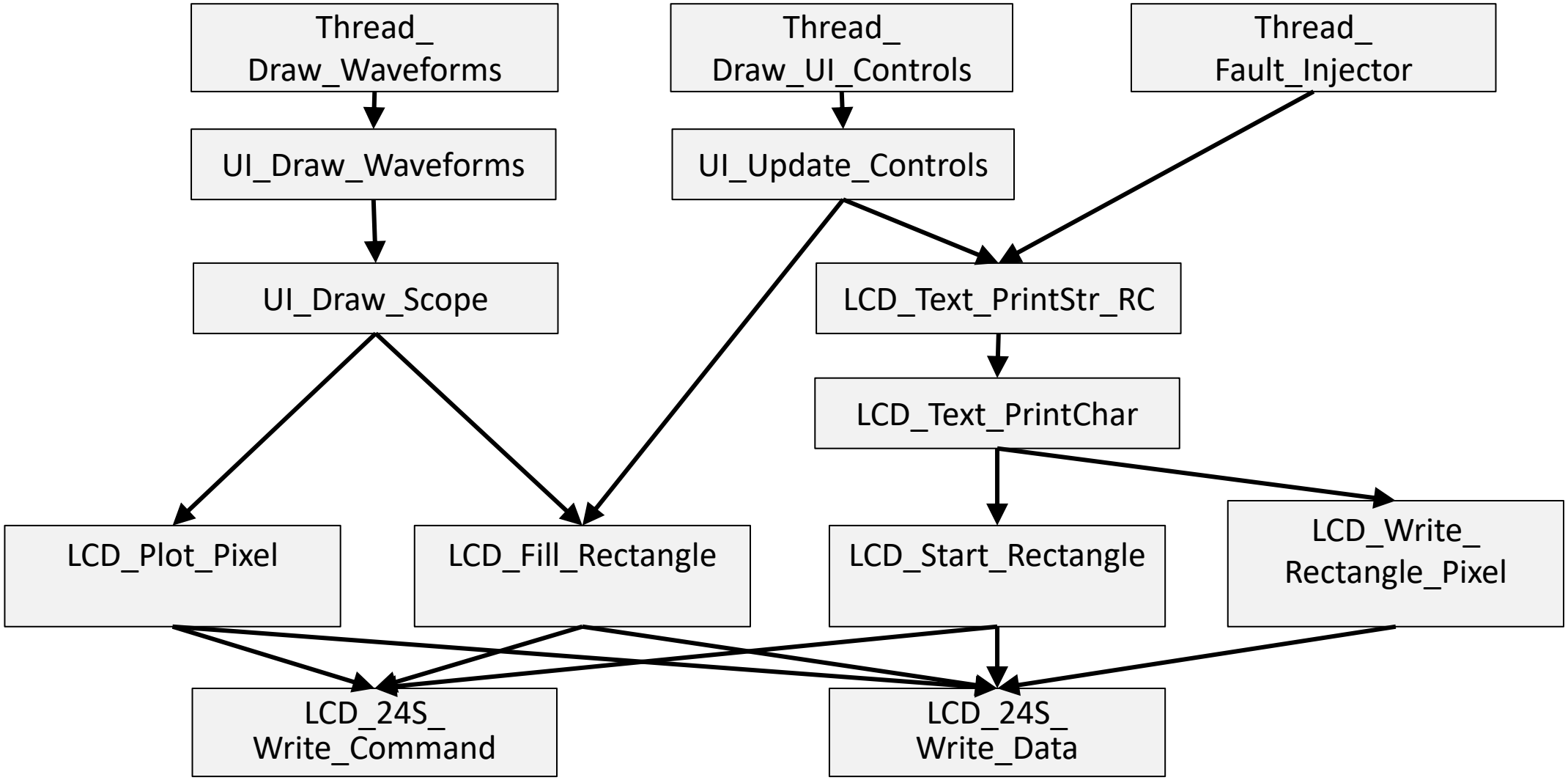


- Write command byte, read parameter byte(s)



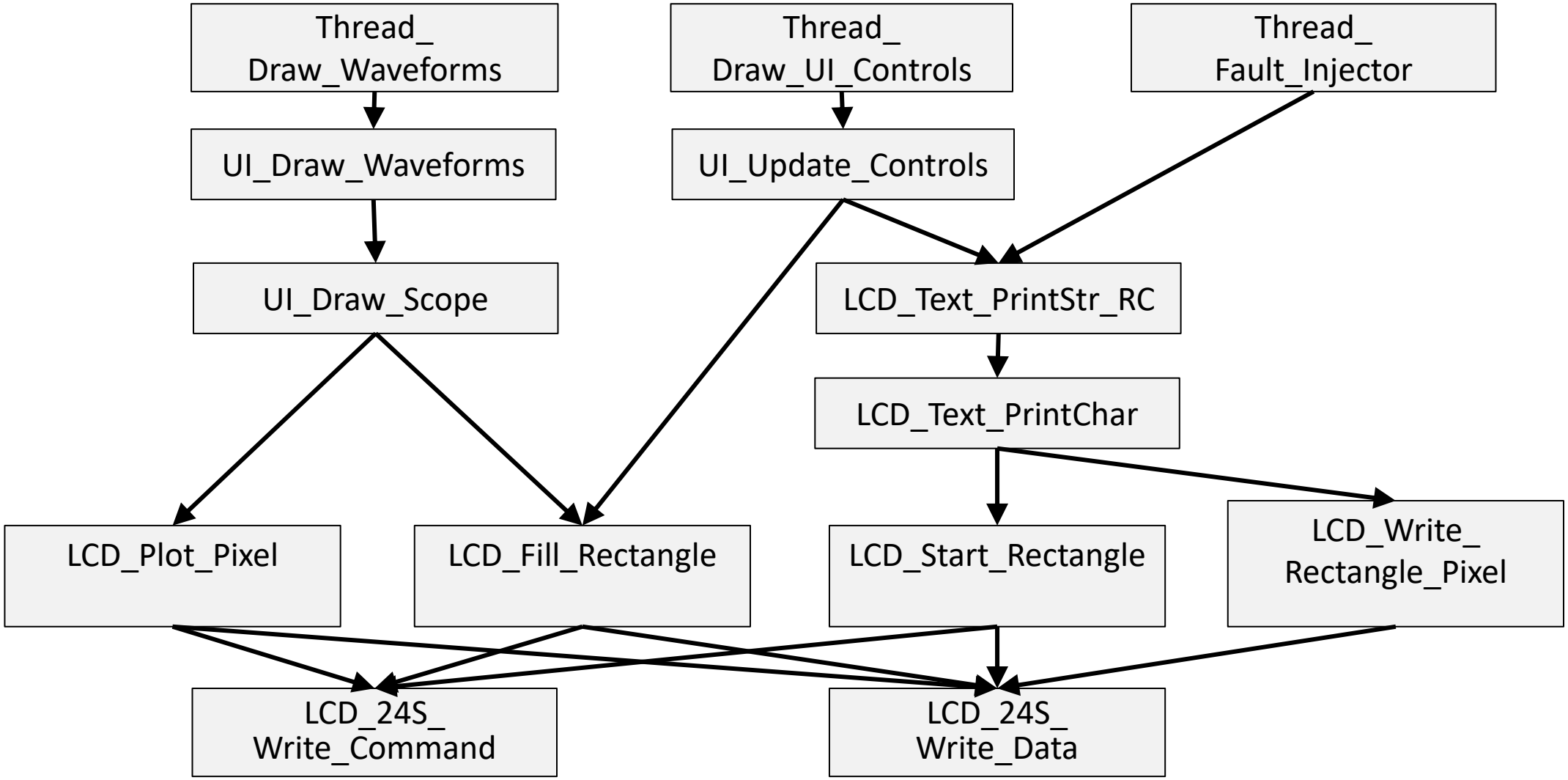
Threads and Functions Accessing LCD Controller

Gathered from Linker-generated Callgraph



Threads and Functions Accessing LCD Controller

Gathered from Linker-generated Callgraph



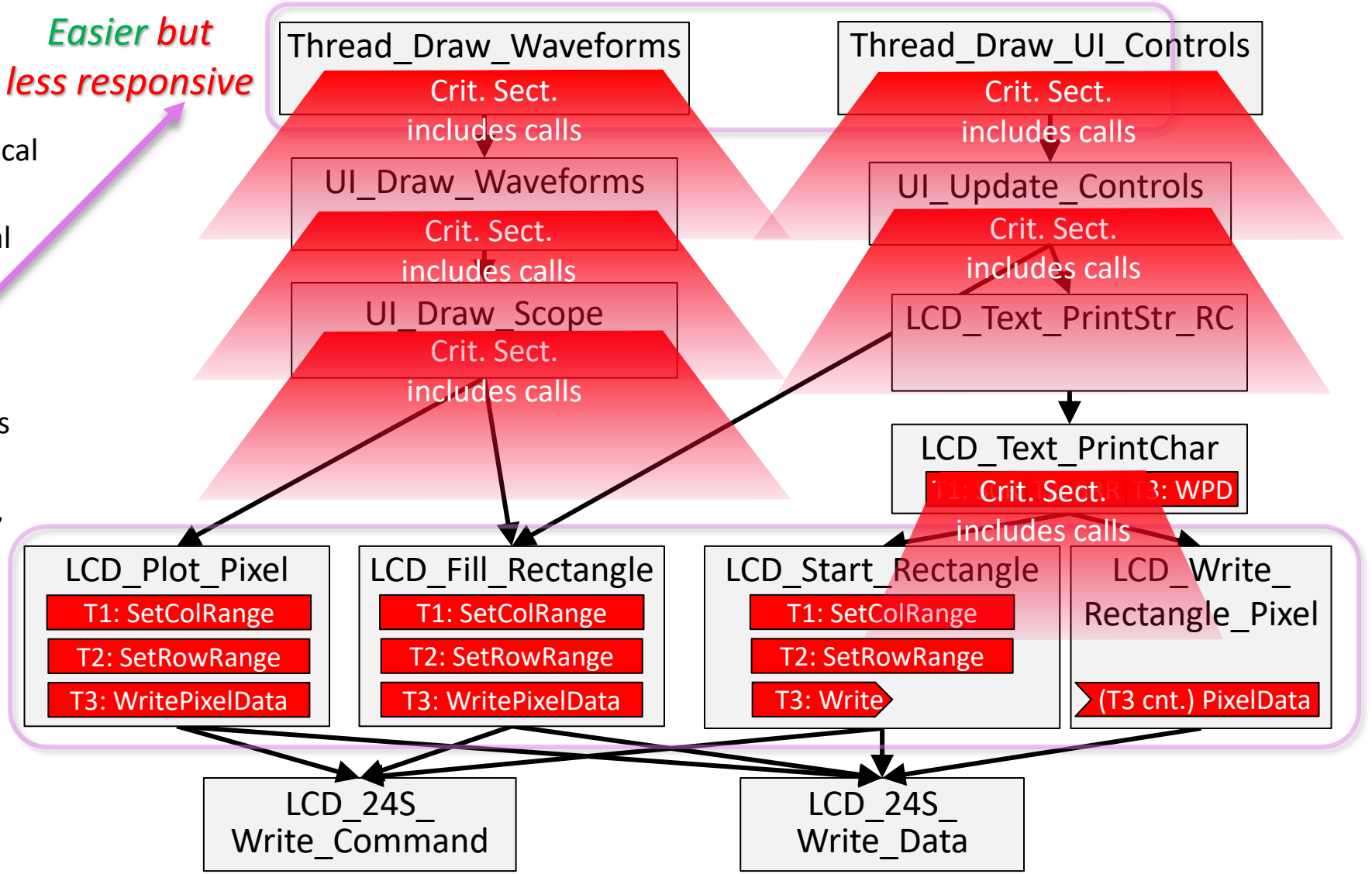
Threads and Functions Accessing LCD Controller

Gathered from Linker-generated Callgraph

- Where to acquire mutex and release mutex?
  - Must acquire mutex **before** critical section starts
  - Must release mutex **after** critical section ends
- Trade-Offs
  - Higher up in call graph: easier, but longer blocking times
  - Lower in call graph: harder (requires more analysis), shorter blocking times

Easier but less responsive

Harder but more responsive



LCD Update Code