Tizen Performance Profiling Tool

Minsoo Ryu

Real-Time Computing and Communications Lab.
Hanyang University

msryu@hanyang.ac.kr

Outline

- Tizen 2.4 SDK Debugging Tools
- Dynamic Analyzer
- Dlog
- T-trace

Tizen 2.4 SDK Debugging Tools

Tizen 2.4 SDK Tools

❒ Tizen 2.4 SDK common Tools

- Analysis and debugging tools
  - Log view
    - ✓ Shows the log, debug, and exception messages
  - Dynamic Analyzer
    - ✓ This tool monitors the performance of your application on a target device or Emulator

Tizen 2.4 SDK Tools

❒ Tizen 2.4 SDK Native Tools

- Analysis and debugging tools
  - Call Stack View
    - ✓ provides useful information for debugging application under crash situation
  - Static Analyzer
    - ✓ finds bugs for source code analysis in Tizen applications
  - Valgrind
    - ✓ detect memory errors or memory leaks in an application
  - T-trace
    - ✓ detect and analyze performance issues

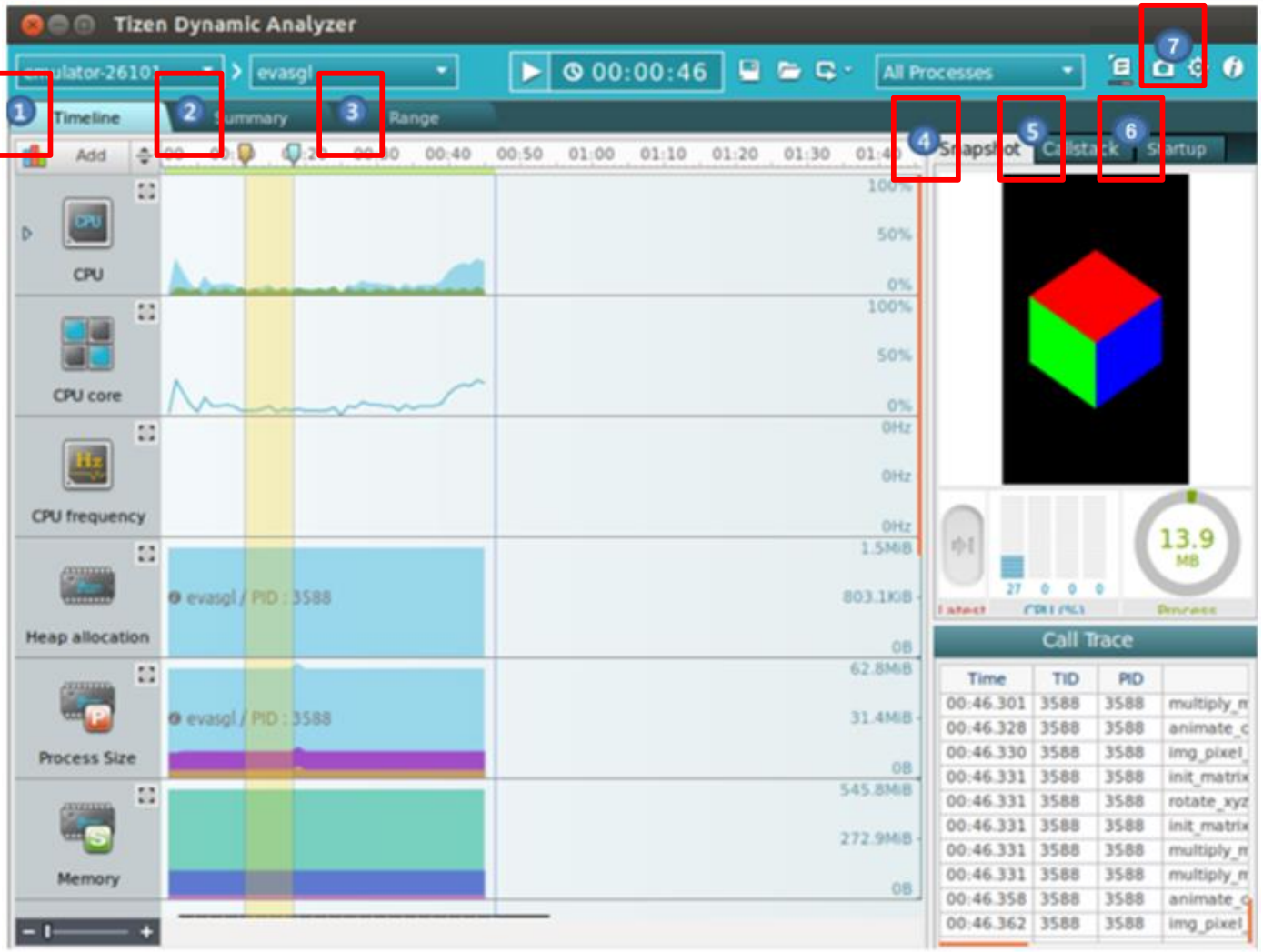
Dynamic Analyzer

Dynamic Analyzer

- Analyze performance
  - To recognize and fix bottlenecks, bugs, and memory and resource leaks
  - To make your applications powerful, faster, and more stable

- Dynamic analyzer provides various functions for profiling applications
  - Monitor the performance and reliability of your application on a target device or the Emulator by running the dynamic analyzer

Dynamic Analyzer



Dynamic Analyzer

- 1. Timeline
  - Show the application data values over time as a graph
- 2. Summary
  - Consist of views showing failed APIs, leaks, profiling information, and warnings
- 3. Range
  - Provide application performance data of a selected range

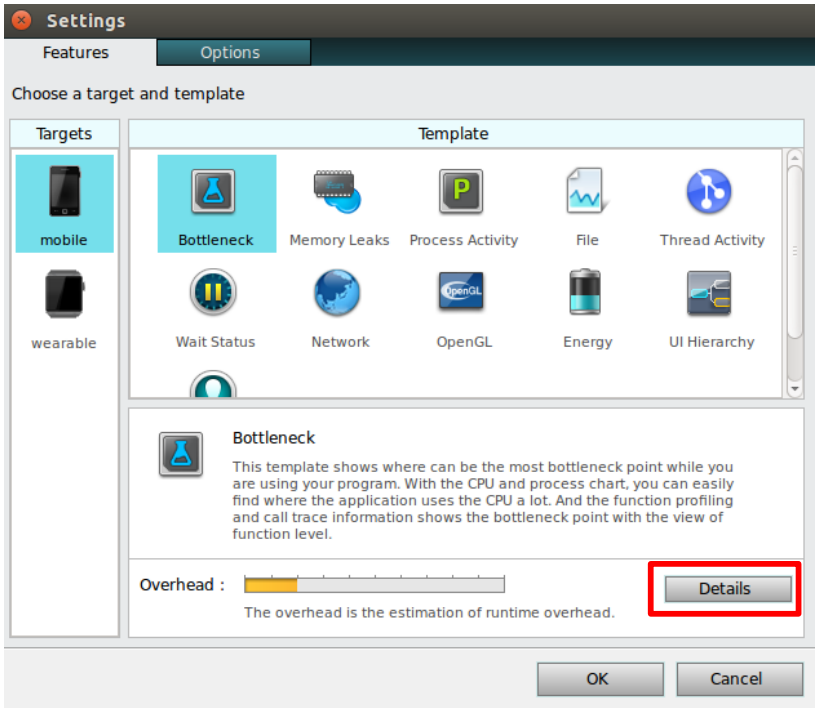
Dynamic Analyzer

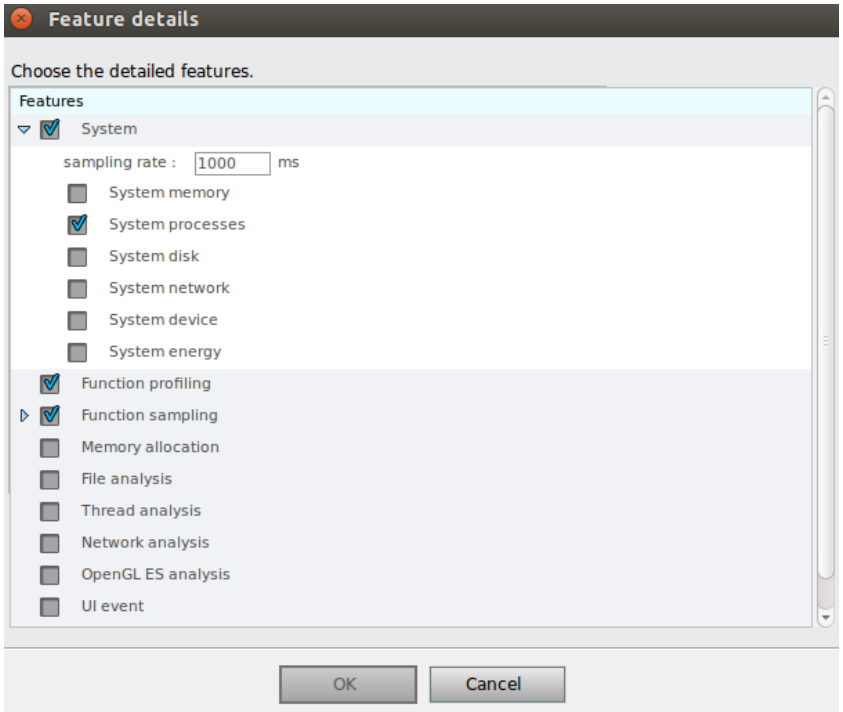
- 4. Snapshot
  - Show the current screen capture and CPU usage, process usage, and available memory
- 5. Callstack
  - Show the callstack of the selected function in the Call Trace
- 6. Startup
  - Show the startup information of the application

Dynamic Analyzer

❒ 7.Settings

- **shows the setting dialog box that you can set the analysis features and other options**
  - Thread, File, Network, OpenGL, CheckPoint, UI Hierarchy, Web





Running Dynamic Analyzer

- **1. Target**
  - **Shows a serial number with the device, or the Emulator name**
- **2. Application**
  - **Contains a list of applications in the selected target**
    - If the Target combo box is empty or disabled, the Application combo box is disabled

Running Dynamic Analyzer

❒ 3. Start/Stop

- **Start or stop the tracing of the selected application**
  - While tracing, the trace result and UI sequence is automatically recorded and temporarily saved

❒ 4. Timer

- **Start when you click the Start button**
  - Updates every second
    - ✓ It shows the current running time of the dynamic analyzer
- **Stops when the analysis processing is complete**

❒ 5. Save Trace

- **Clicking the button saves the trace data permanently**

Running Dynamic Analyzer

- **6. Open Trace**
  - **Loads and displays the saved trace data**
- **7. Replay**
  - **Repeats a previous analysis**
    - If the target or application do not match, the button is disabled
- **8. Process**
  - **Show a process list of the application being traced.**
    - Default: show the analysis results of all processes.
    - Select a process in the list: shows the analysis result of special process only

Running Dynamic Analyzer

❒ 9. View Source

- **Displays the source code as a tooltip**
  - If you click the button and the mouse is on the method name in any table-like view
  - By double-clicking the tooltip you can see the source code in the IDE

- **Apart from the Callstack view, the source code displayed is the caller part of the selected API, not the API definition**
  - If an API is called from a shared library, the source code is not displayed as the source code of the shared library is not available.
  - In the callstack view, the source code corresponding to the address of the selected callstack unit is displayed.

Running Dynamic Analyzer

- 10. Capture screen
  - Capture the screen of the target at the time
  - The screenshot is shown in the Snapshot view
- 11. Settings
  - Support the runtime configuration feature and other settings
- 12. About
  - Displays the dynamic analyzer version, build time, and license

Using the Analysis Result

Using the Analysis Result

- **Use the result selectively to meet your improvement purposes**

- **The following instructions help you to utilize the analysis result:**
  - **Performance Analysis**
    - Describes how to analyze application performance
  - **Detecting Leaks**
    - Describes how to detect memory and resource leaks
  - **Multi-threaded Application and Synchronization Analysis**
    - Describes how to analyze threads and synchronization

Performance Analysis

❒ User Function Profiling

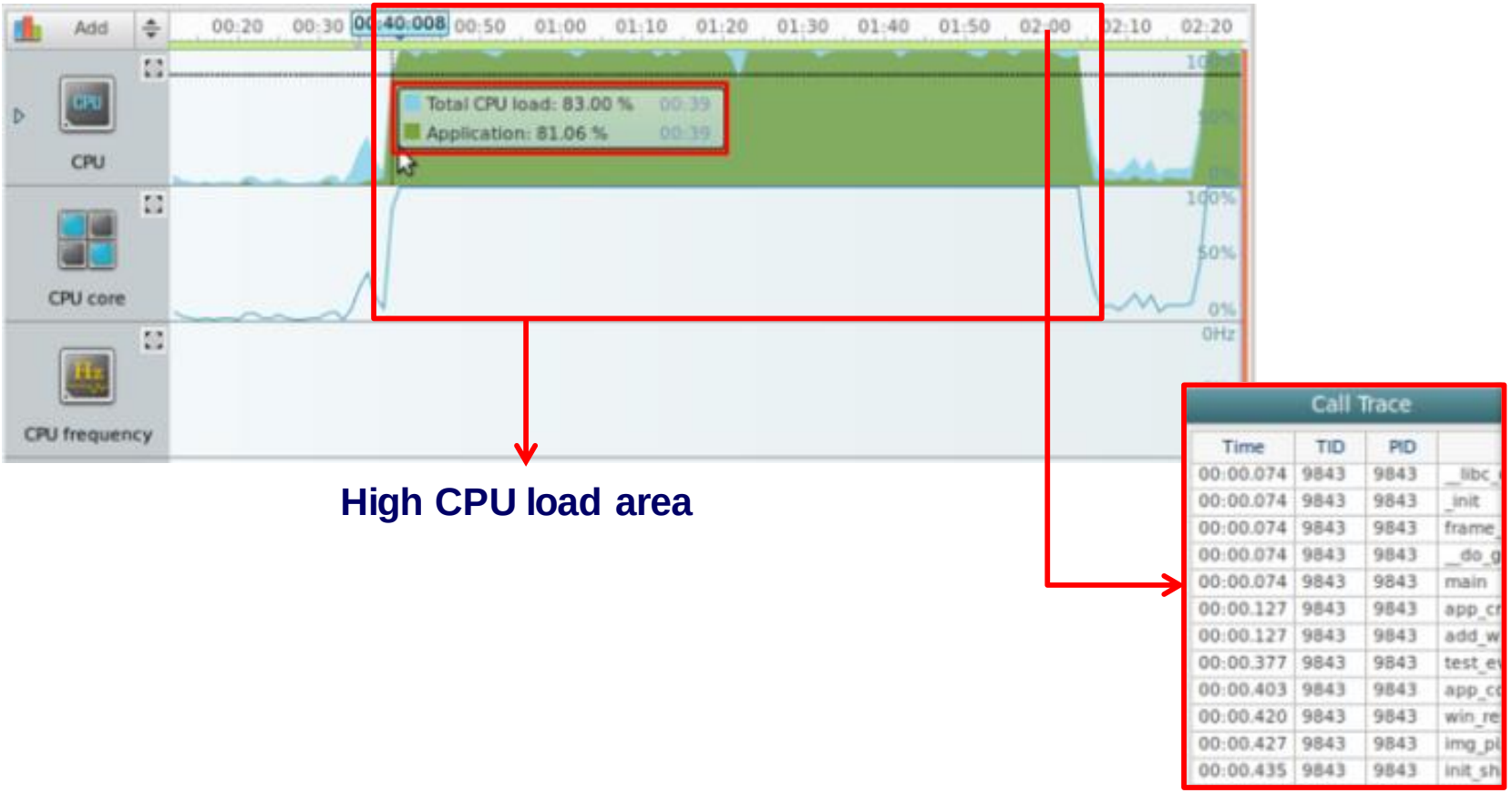
- **Execution time of each method is one of the most significant factors**
- **To improve the performance of your application**
  - By identifying unexpected bottleneck locations
- **To detect and fix the methods consuming the most time:**
  - Select the Summary tab
    - ✓ View the Function Usage Profiling view displaying the methods consuming the most time
    - ✓ Click the title of a column to view the sorted results
  - Use the range information feature of the dynamic analyzer
    - ✓ To view the execution time of the methods called within a specific time period
  - The time consumed by UI-related methods is displayed on the UI Function Profiling view of the UI tab

Performance Analysis

❒ Timeline CPU Chart

- **CPU load is one of the most significant factors**
- **CPU load peak can result in a performance bottleneck**
  - High CPU load leads to increased memory consumption
- **To detect and fix CPU load peaks with the CPU load feature :**
  - Select the Timeline tab and view the CPU chart
  - Hover the mouse on a CPU peak to view the CPU load value in a tooltip
  - Click the CPU peak to highlight the last user method called before the peak in the Call Trace view
  - Click the View Source button and place the mouse on the highlighted method
    - ✓ The source code is displayed as a tooltip

Performance Analysis

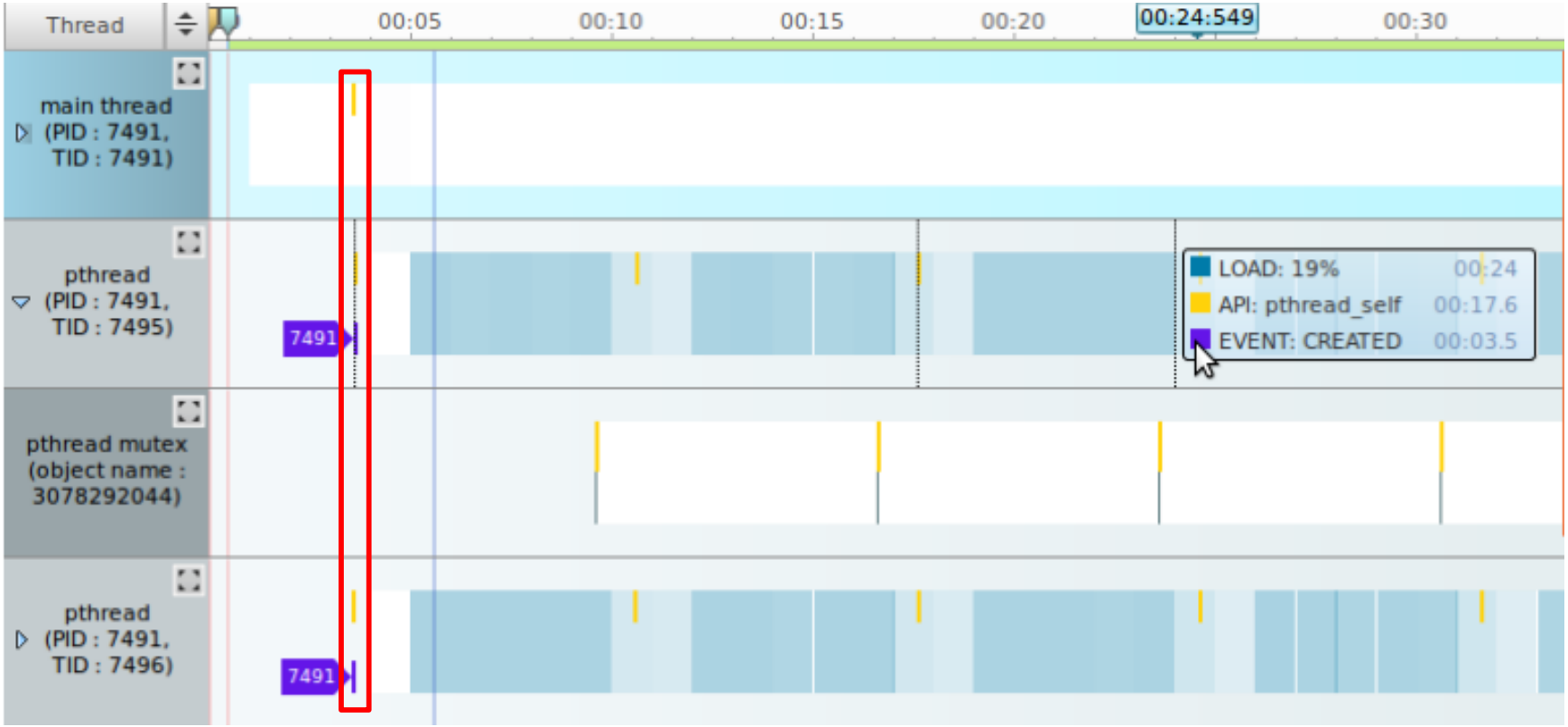


Performance Analysis

❒ Thread Load

- **Need to analyze the load of each thread**
  - Analyzing The thread load feature helps to distribute the thread load
- **The thread load is displayed in the Thread tab**
- **The thread line displayed**
  - Blue: the thread load within a time frame
  - Darker: the higher the load
- **To improve performance, consider the following:**
  - You want to Select the right algorithm and data structures
  - If your code includes sort, search, or compare, use optimal algorithms and data structures
    - ✓ Change the order of complexity
  - Split the jobs into multiple tasks
    - ✓ Running with high and low priority jobs in a single task causes delays

Performance Analysis

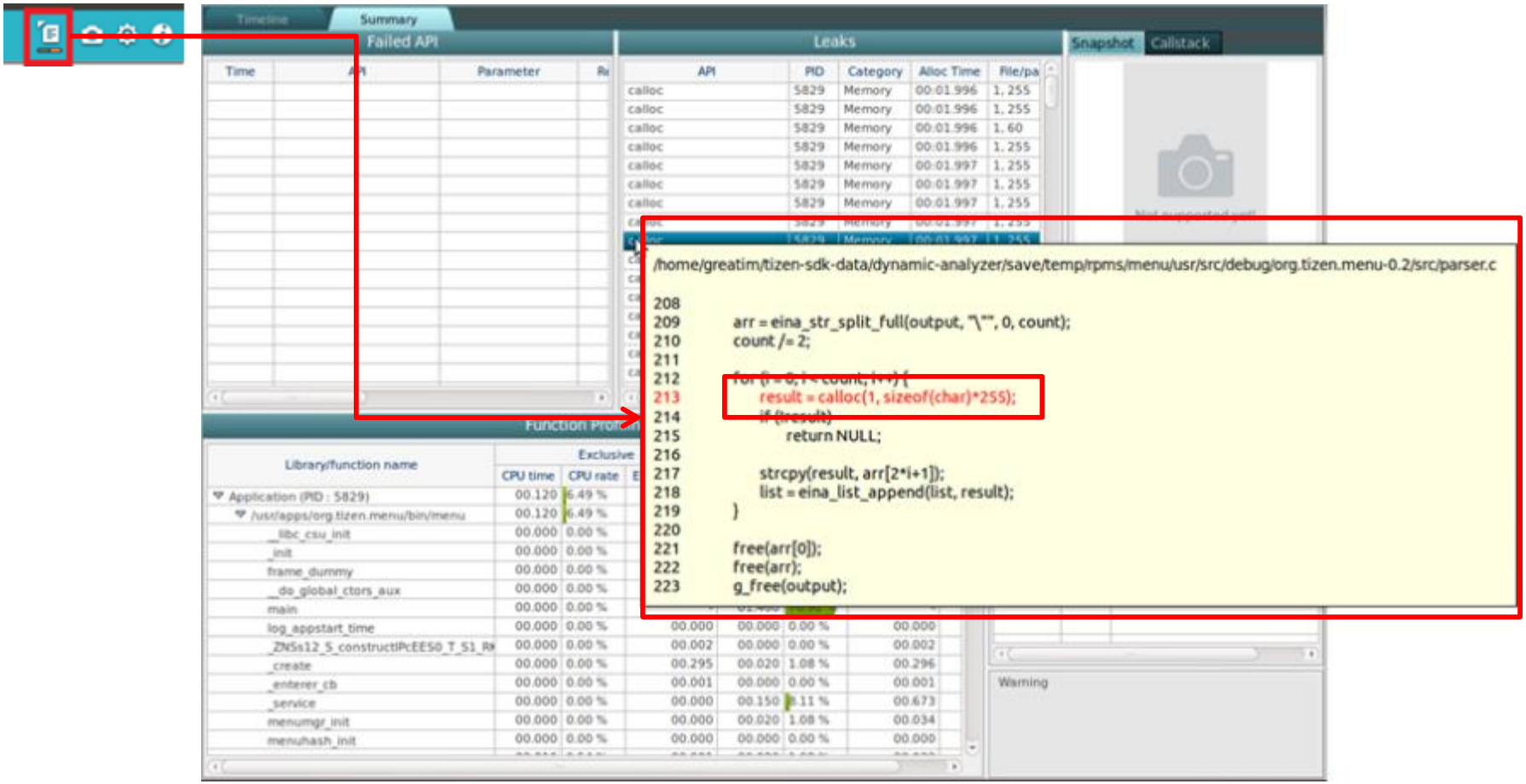


Detecting Leaks

❒ Detect memory leaks

- **Memory leaks occur when memory capacity that is dynamically allocated during application execution is not returned after the execution stops**
  - Severe or accumulating memory leaks can affect the performance of other applications and programs
- **To detect and fix memory leaks:**
  - Select the Summary tab
    - ✓ To view the memory leaks occurring
  - Click the View Source icon in the toolbar
  - Move the mouse pointer to the list item you want to check
    - ✓ The part causing the memory leak is displayed in red

Detecting Leaks

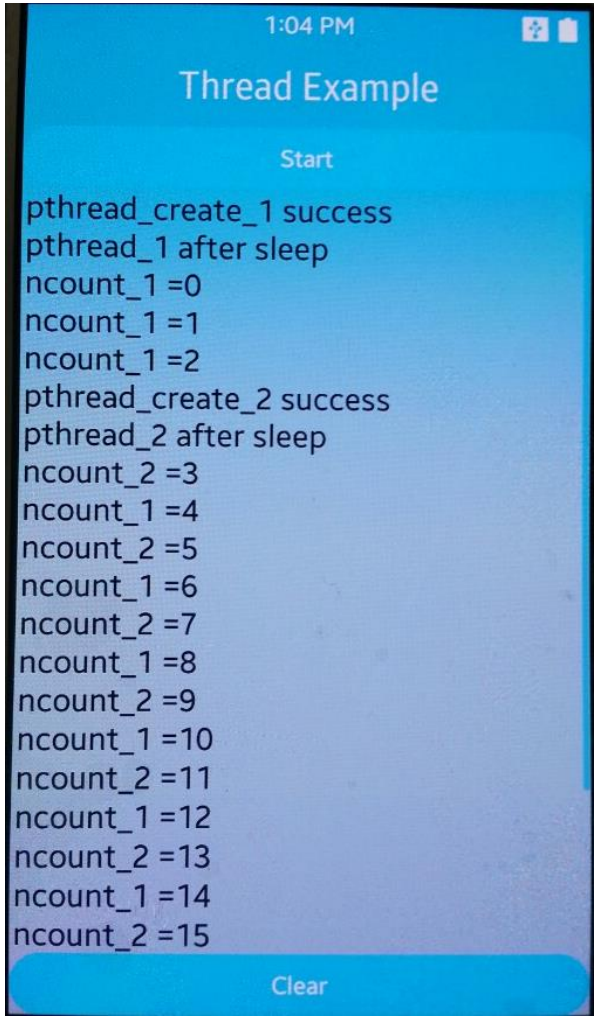


Assignment 1: Dynamic Analyzer

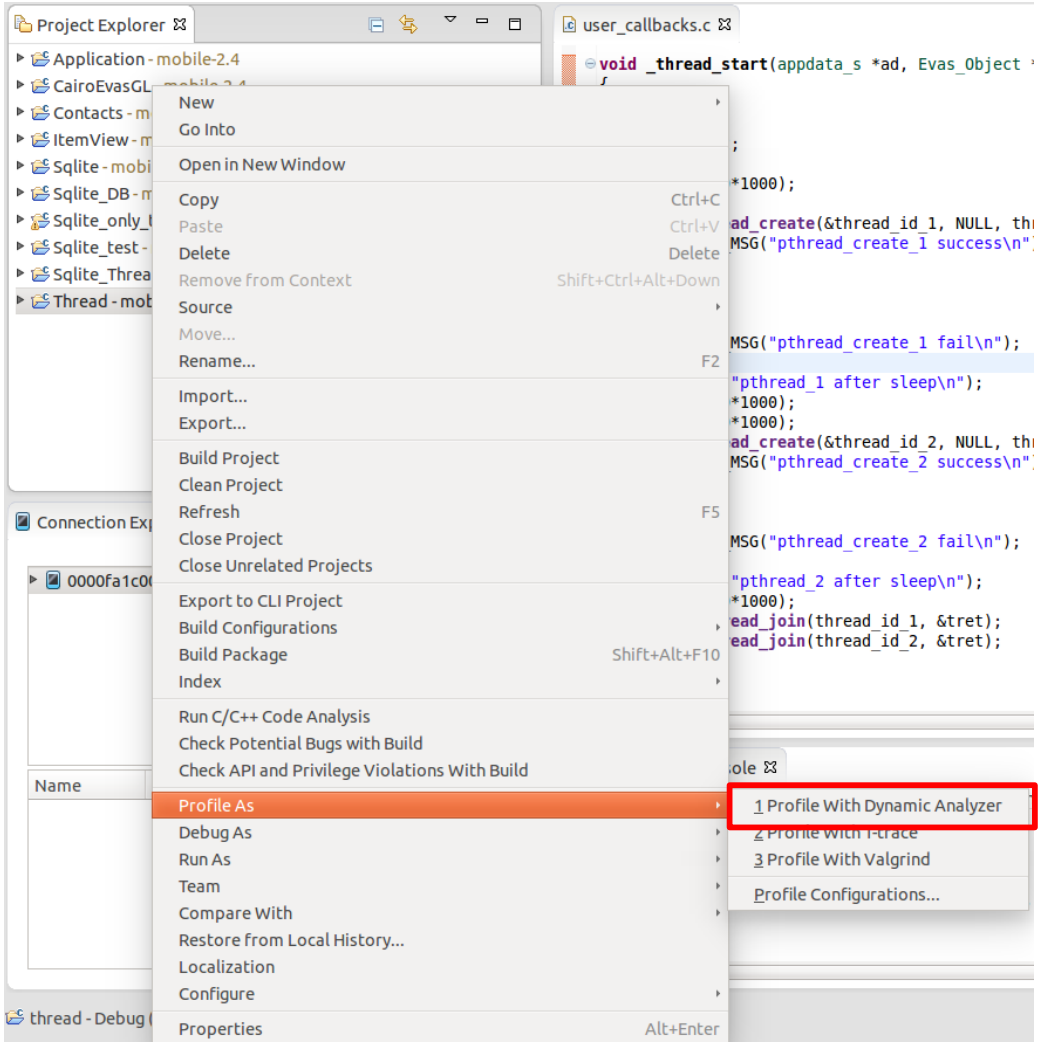
Lab1 : Assignment Dynamic Analyzer

user_callbacks.c

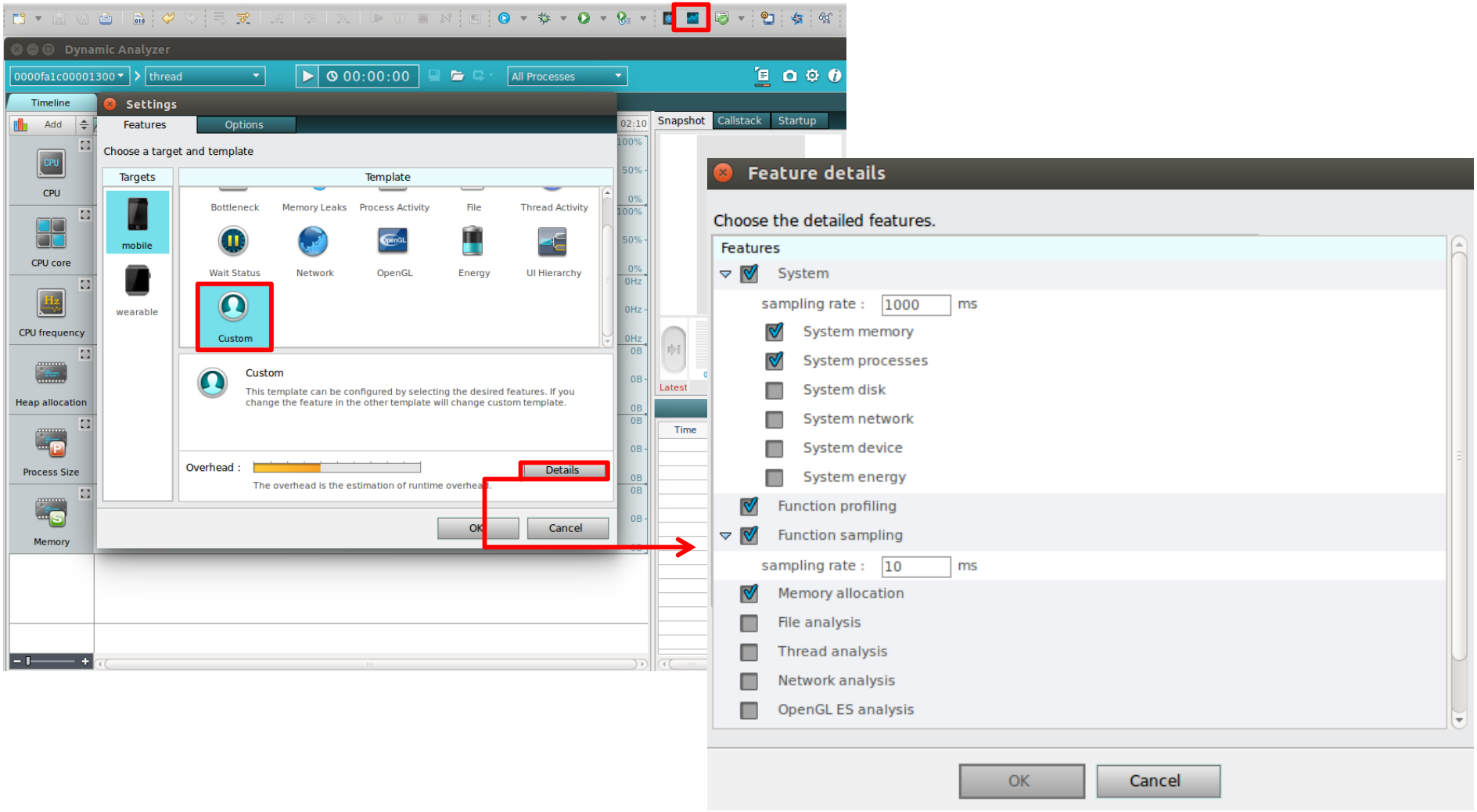
```c
void _thread_start(appdata_s *ad, Evas_Object *obj, void *event_info)
{
    int ret;
    int err;
    void *tret;

    usleep(300*1000);

    if (!pthread_create(&thread_id_1, NULL, thread_run_1, NULL)){
        PRINT_MSG("pthread_create_1 success\n");
    }
    else
    {

        PRINT_MSG("pthread_create_1 fail\n");
    }
    PRINT_MSG("pthread_1 after sleep\n");
    usleep(300*1000);
    usleep(300*1000);
    if (!pthread_create(&thread_id_2, NULL, thread_run_2, NULL)){
        PRINT_MSG("pthread_create_2 success\n");
    }
    else
    {
        PRINT_MSG("pthread_create_2 fail\n");
    }
    PRINT_MSG("pthread_2 after sleep\n");
    usleep(300*1000);
    err = pthread_join(thread_id_1, &tret);
    err = pthread_join(thread_id_2, &tret);

}
```



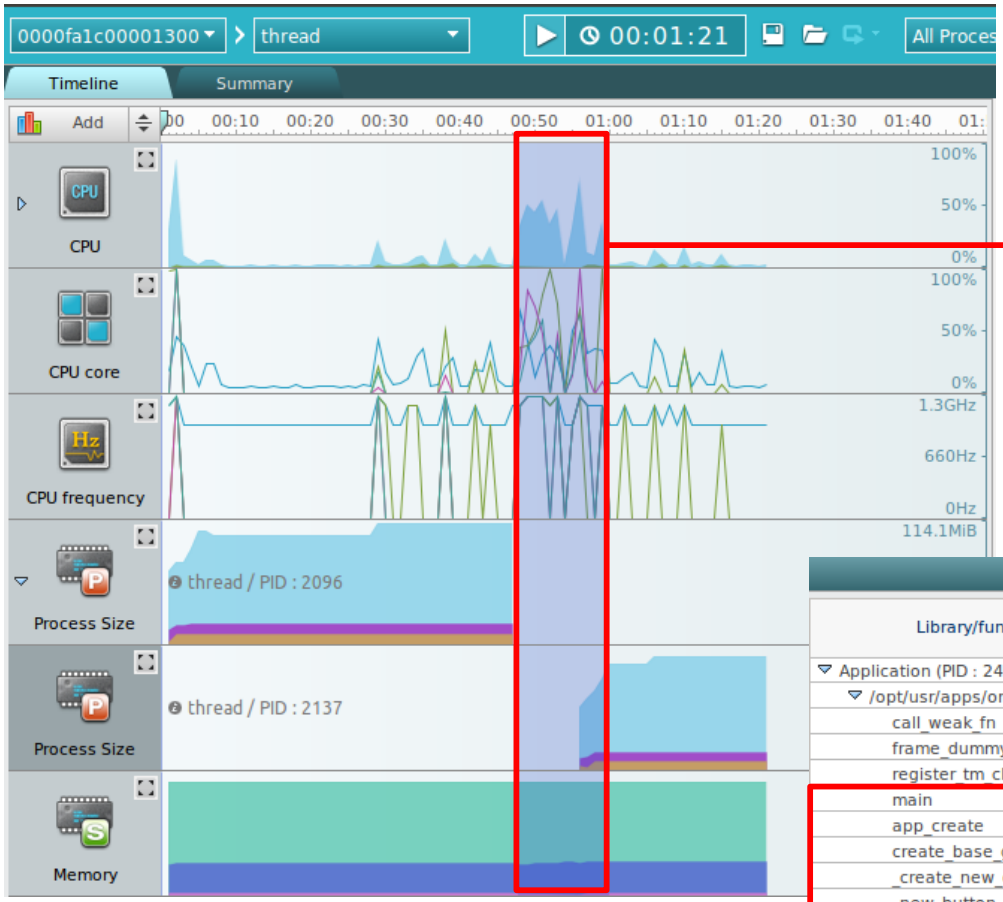
Assignment 1: Dynamic Analyzer



Assignment 1: Dynamic Analyzer



Assignment 1: Dynamic Analyzer



Call Trace

Time	TID	PID	API
00:55.810	2137	2137	app_create
00:55.810	2137	2137	create_base_gui
00:55.950	2137	2137	_create_new_cd_display
00:56.122	2137	2137	_new_button
00:59.022	2137	2137	_thread_start
00:59.323	2137	2137	_add_entry_text
00:59.325	2137	2137	_add_entry_text
00:59.326	2200	2137	thread_run_1
00:59.326	2200	2137	_add_entry_text
00:59.527	2200	2137	_add_entry_text
00:59.728	2200	2137	_add_entry_text
00:59.926	2137	2137	_add_entry_text
00:59.927	2137	2137	_add_entry_text
00:59.929	2200	2137	_add_entry_text
00:59.930	2201	2137	thread_run_2

Function Profiling

Library/function name	Exclusive			Inclusive			Ca
	CPU time	CPU rate	Elapsed time	CPU time	CPU rate	Elapsed time	
Application (PID : 2492)	00.000	0.00 %	-	01.920	47.64 %	-	
/opt/usr/apps/org.example.thread/bin/th	00.000	0.00 %	-	01.920	47.64 %	-	
call_weak_fn	00.000	0.00 %	-	00.000	0.00 %	00.000	
frame_dummy	00.000	0.00 %	00.000	00.000	0.00 %	00.000	
register_tm_clones	00.000	0.00 %	00.000	00.000	0.00 %	00.000	
main	00.000	0.00 %	-	01.820	45.16 %	-	
app_create	00.000	0.00 %	00.000	00.330	8.19 %	00.346	
create_base_gui	00.000	0.00 %	00.163	00.330	8.19 %	00.346	
_create_new_cd_display	00.000	0.00 %	00.173	00.170	4.22 %	00.173	
_new_button	00.000	0.00 %	00.009	00.010	0.25 %	00.009	
_thread_start	00.000	0.00 %	11.650	00.010	0.25 %	11.660	
_add_entry_text	00.000	0.00 %	00.062	00.090	2.23 %	00.062	
thread_run_1	00.000	0.00 %	-	00.050	1.24 %	-	
thread_run_2	00.000	0.00 %	-	00.050	1.24 %	-	

Multi-threaded Application and Synchronization Analysis

- **The dynamic analyzer provides effective thread analysis features**
  - **Understanding the thread execution in multi-threaded applications can be challenging**
  - **Using synchronization object is better than GNU Debugger**
    - The GDB supports the process of debugging multi-threaded applications
    - The debugging can be quite difficult
- **Using Dynamic analyzer:**
  - **Analyze thread life-cycle**
  - **Analyze thread concurrency**
  - **Analyze synchronization**

Analyzing Thread Life-cycle

- Testing threads is difficult as they are non-deterministic
- Visualizing the thread life-cycle is an effective method for analyzing the thread life-cycle

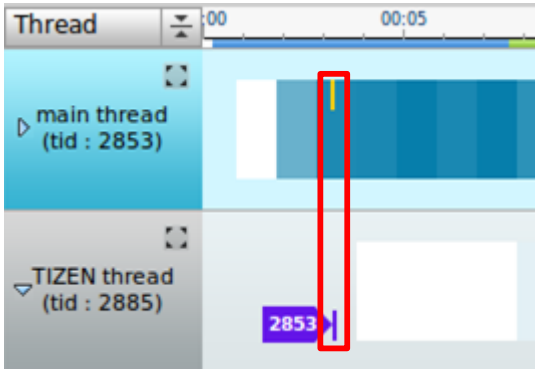
- The dynamic analyzer has 3 types of user threads:
  - **Main thread**
    - created from the system for running applications
  - **Tizen thread**
    - including worker threads and event-driven threads
  - **pthread**

Analyzing Thread Life-cycle

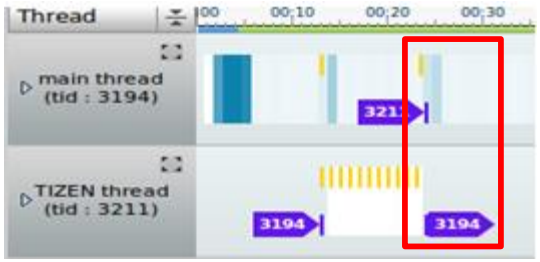
❒ Thread chart

- **The thread life-cycle is displayed as follows:**

- **Thread creation:**
  - When an API is called to create the thread, a yellow bar is displayed and a new chart item is created with a TID arrow



- **Thread exiting:**
  - When a thread exits, a purple bar is displayed with a joined TID arrow
  - If another thread calls the API of the exited thread, a yellow bar is displayed



Analyzing Thread Life-cycle

- Check the state of the thread on the Thread tab
- A new thread is created in a joinable state
  - Otherwise, memory and resource leaks can occur until the process ends
- To make the process faster
  - set the thread to the detach state, or call the detach API

Analyzing Thread Concurrency

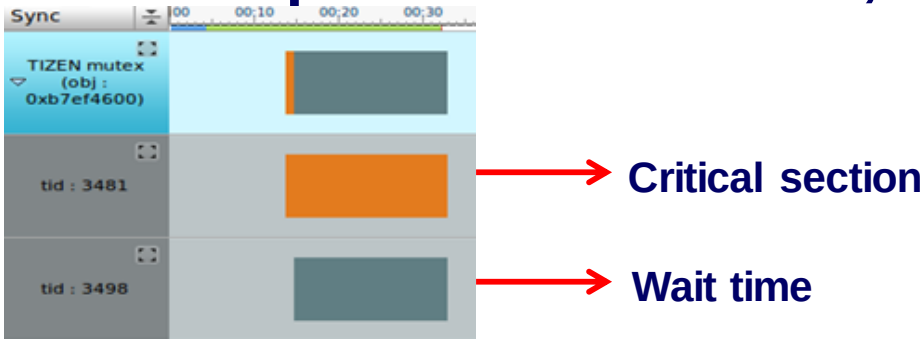
- The number of live threads and their resource usage can be used to determine an efficient thread concurrency
- The thread chart displays the relationship and progress between threads and allows you to check the number of live threads
- The CPU load of a thread is also displayed

Analyzing Synchronization

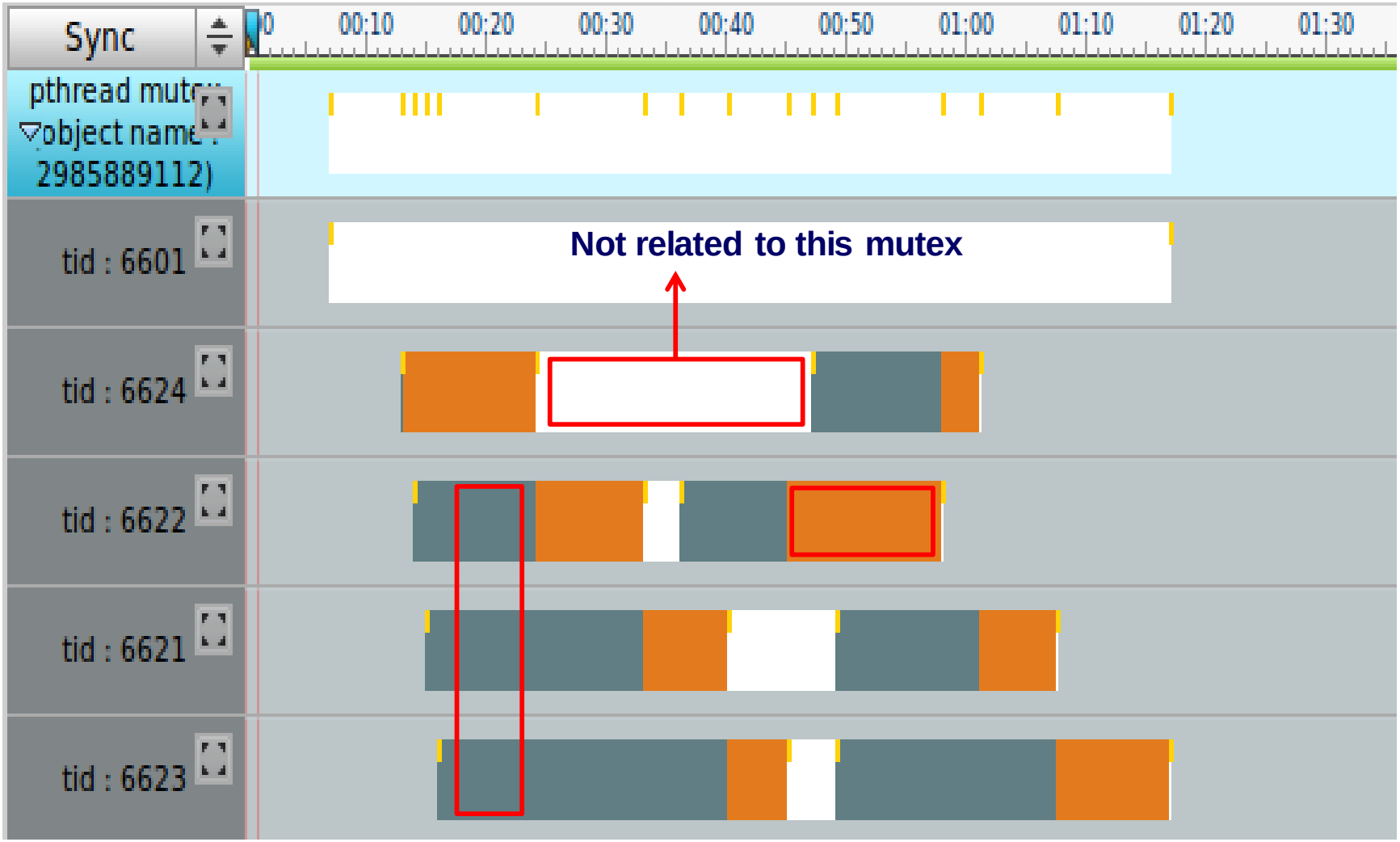
- Threads must be synchronized
  - When multiple threads access the same resources, data race occurs
- The synchronization chart in the dynamic analyzer has the following synchronization objects:
  - Tizen mutex / Tizen monitor / Tizen semaphore
  - pthread mutex / pthread condition variable
- View the synchronization chart based on the thread, or the synchronization status:
  - Select Thread in the synchronization chart combo box
    - To view the child of each thread chart item
  - Select Sync
    - To view the parent item with the thread for each usage showing the child items

Analyzing Synchronization

- **Analyze the critical section duration and waiting time**
  - **A synchronization object can be checked using the synchronization chart**
  - **When a synchronization object acquires a lock, the thread enters the critical section**
    - If the critical section duration increases, the thread stops working concurrently and affects the performance
    - If a thread acquires a lock, the critical section waiting time of the other threads increases
- **To avoid potential dead lock, reduce the waiting time**

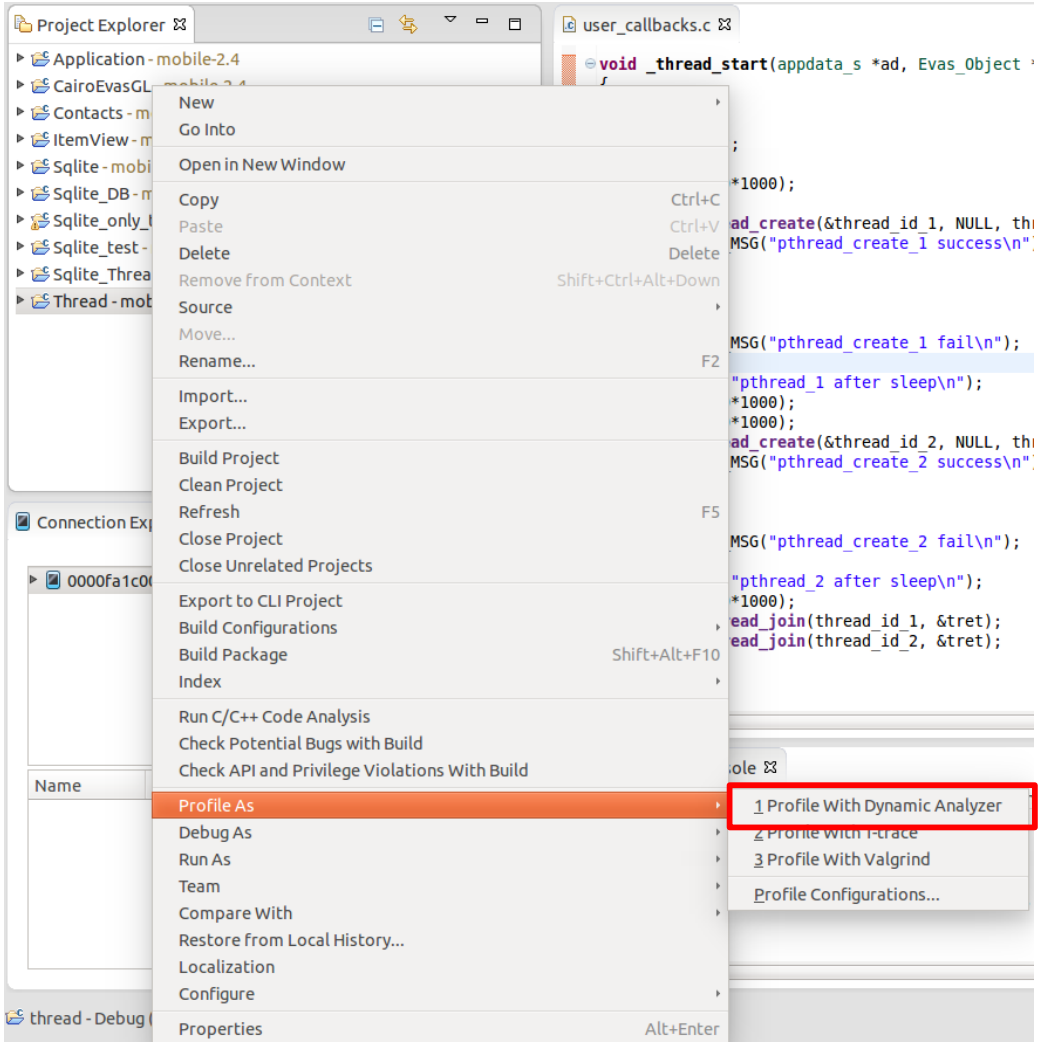


Analyzing Synchronization

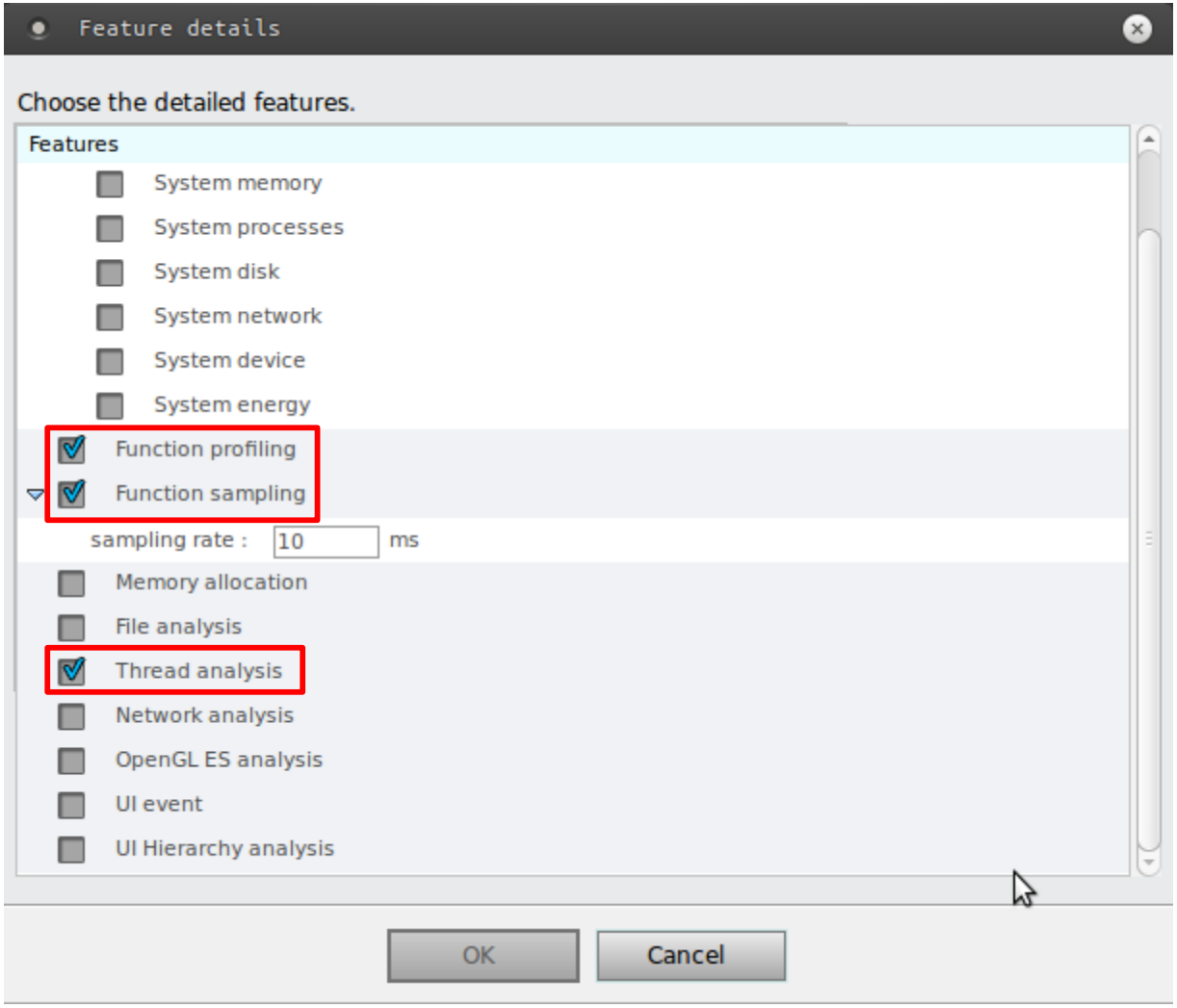


Assignment 2: Dynamic Analyzer

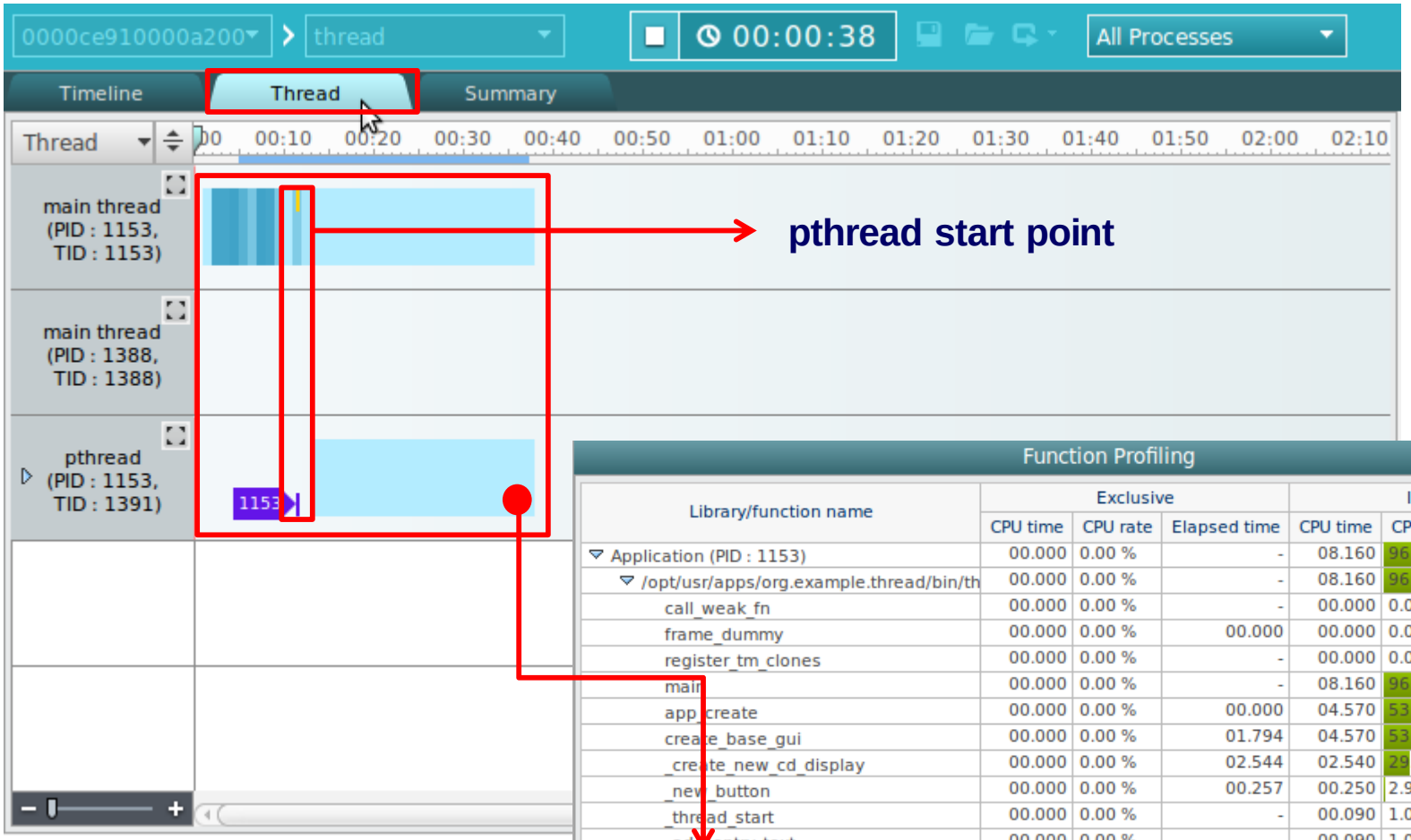
Assignment 2: Dynamic Analyzer



Assignment 2: Dynamic Analyzer



Assignment 2: Dynamic Analyzer



Function Profiling

Library/function name	Exclusive			Inclusive		
	CPU time	CPU rate	Elapsed time	CPU time	CPU rate	Elapsed time
Application (PID : 1153)	00.000	0.00 %	-	08.160	96.00 %	-
/opt/usr/apps/org.example.thread/bin/th	00.000	0.00 %	-	08.160	96.00 %	-
call_weak_fn	00.000	0.00 %	-	00.000	0.00 %	00.000
frame_dummy	00.000	0.00 %	00.000	00.000	0.00 %	00.000
register_tm_clones	00.000	0.00 %	-	00.000	0.00 %	-
main	00.000	0.00 %	-	08.160	96.00 %	-
app_create	00.000	0.00 %	00.000	04.570	53.76 %	04.596
create_base_gui	00.000	0.00 %	01.794	04.570	53.76 %	04.596
_create_new_cd_display	00.000	0.00 %	02.544	02.540	29.88 %	02.544
_new_button	00.000	0.00 %	00.257	00.250	2.94 %	00.257
_thread_start	00.000	0.00 %	-	00.090	1.06 %	-
_add_entry_text	00.000	0.00 %	-	00.090	1.06 %	-
thread_run_1	00.000	0.00 %	-	00.000	0.00 %	-
Dependent library (PID : 1153)	08.470	99.65 %	-	08.470	99.65 %	-

Dlog

Dlog

- Tizen log system
- The dlog logging service consists of the dlogutil and dlog library
- The dlog service sends a log message to the circular buffer with APIs, including Priority and Tag information
  - With this information, it is easy to filter and check the messages with dlogutil

Dlog

Priority

The priority level indicates the urgency of the log message.

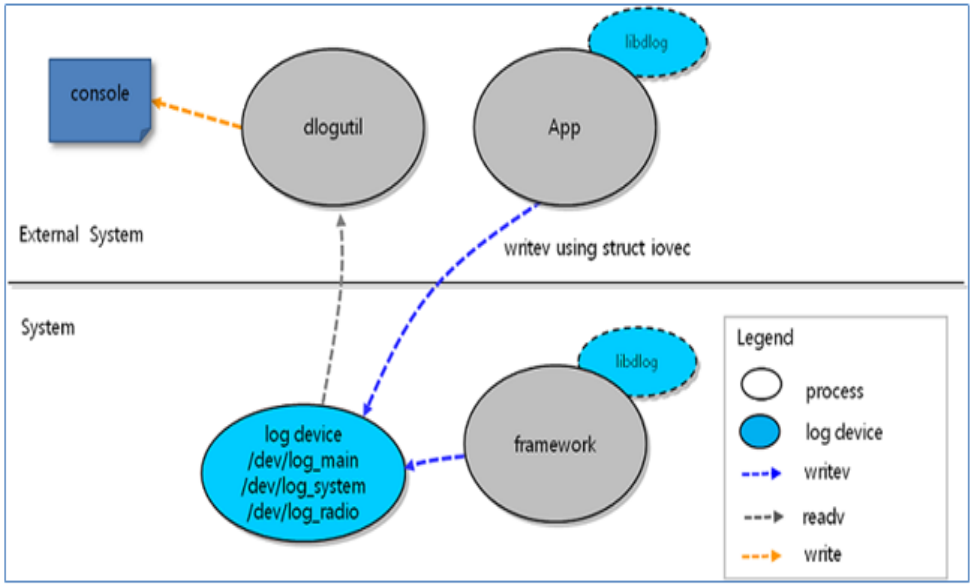
Table: Priority levels

Priority	Description
DLOG_DEBUG	Log message which the developer wants to check
DLOG_INFO	Normal operational message
DLOG_WARN	Not an error, but a warning that an error will occur if action is not taken
DLOG_ERROR	An error

Architecture

The following figure illustrates the general architecture of the dlog logging service.

Figure: Architecture



Dlog

❒ Log Tag

- A tag is used to identify the source of the log message
- There are no naming limitations, but do not forget that the tag is an identification of a module
- Each tag must be unique

❒ Dlogutil

- You can collect and print out logs with logutil. Logutil supports filtering, and managing the log device

```
dlogutil [<option>] ... [<filter-spec>] ...
```

- The filter expression follows the *tag:priority* or *tag* format

Dlog

❒ Initialize dlog

```
#include <dlog.h>
```

❒ Sending a Log Message

```
#define TAG "MY_APP"

int
main(void)
{
   int integer = 21;
   char string[] = "test dlog";

   dlog_print(DLOG_DEBUG, TAG, "debug message");
   dlog_print(DLOG_INFO, TAG, "info message");
   dlog_print(DLOG_WARN, TAG, "warning message");
   dlog_print(DLOG_ERROR, TAG, "error message");
   dlog_print(DLOG_INFO, TAG, "%s, %d", string, integer);

   return 0;
}
```

T-trace

T-trace

- The T-trace is a profiling tool that allows you to optimize application performance by measuring and visualizing instrumented function calls in the Tizen platform
  - It helps you to understand what the system is doing while your application is running
- Tracing points is already added in Tizen 2.4 platform
  - Application Manager, Video, Camera, Graphics and so on

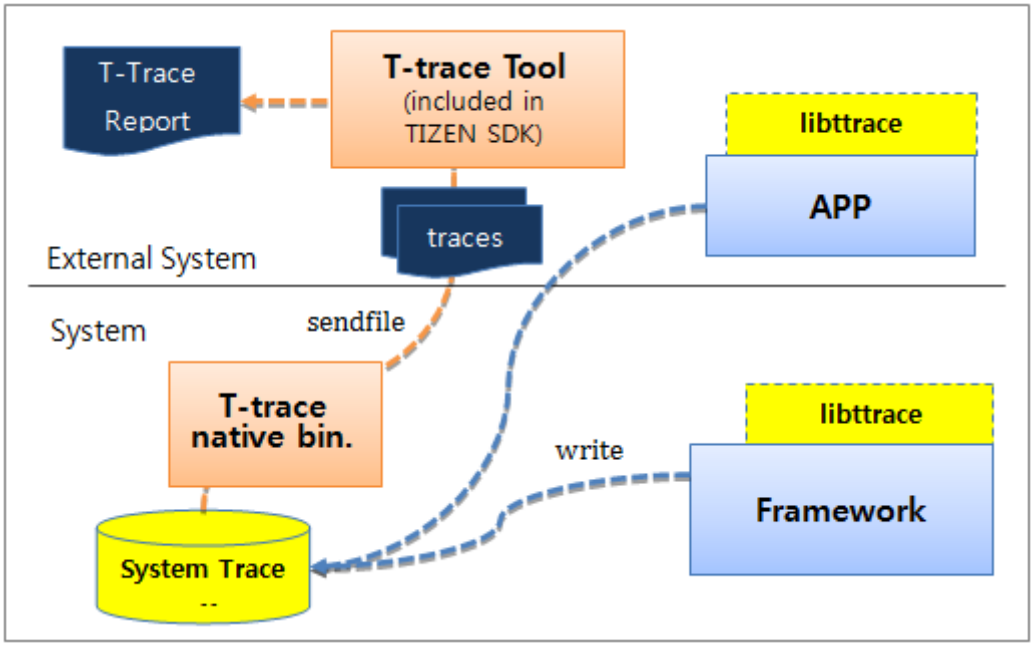
T-trace

❒ Android Systrace based

- **Tizen git : framework/system/ttrace**
  - Android trace code is founded in t-trace

❒ Host Requirements

- **Python 2.7 ↑**
- **Google Chrome**



T-trace : Insert tracepoints

❒ Initializing Tracing

- To use the functions and data types of the T-trace API, include the <trace.h> header file in your application

```
#include <trace.h>
```

T-trace : Insert tracepoints

❒ Inserting Tracepoints

- **Use synchronous tracing**
  - If the trace event starts and ends in a same context within the same process, thread, and function, use the trace_begin() and trace_end() functions to track the event.
  - Every trace_begin() function matches up with a trace_end() function that occurs after it.

```
int
main(void)
{
    int integer = 12;
    trace_begin("event name: %d", integer);

    trace_end();

    return 0;
}
```

T-trace : Insert tracepoints

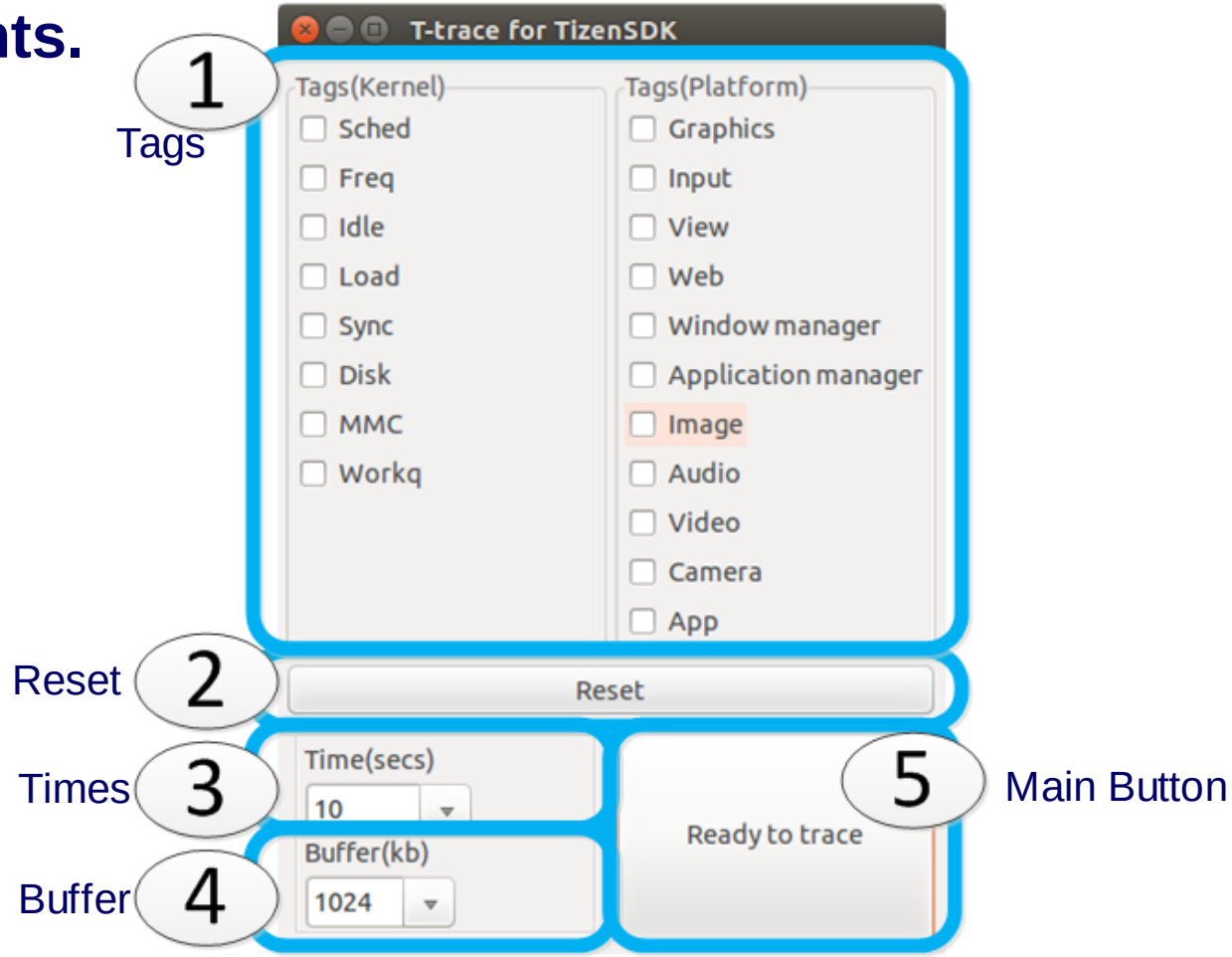
❒ Inserting Tracepoints

- **Use asynchronous tracing**
  - If the trace event starts and ends in a different context, use the trace_async_begin() and trace_async_end() functions to track the event.
  - Every trace_async_begin() function matches with a trace_async_end() function that has the same name and cookie.

```
void
function1()
{
    int cookies_f1 = 123;

    trace_async_begin(cookies, "event name");
}

void
function2()
{
    int cookies_f2 = 123;

    trace_async_end(cookies_f2, "event name");

}
```

T-trace Dialog

- The following figure illustrates the T-trace dialog elements.



T-trace Dialog

❒ 1. Tags

- **You can define which categories to trace**
  - several Tizen platform-specific categories
  - a few low level system information categories
  - The Linux kernel and Tizen platform modules support the categories
    - ✓ such as EFL, xorg, and mmfw
- **To enable the categories you want, select the applicable check boxes**

❒ 2. Reset

- **Select the check box to return all tags and options in the T-trace dialog to their default values**

T-trace Dialog

❒ 3.Time(secs)

- **Set the time period to be used for tracing**
  - You can select a predefined value of 10, 30, 60, or 120 seconds, enter a value of your own, or select manual (which means that no specific tracing time is set and you stop tracing when you want)

❒ 4. Buffer(kb)

- **Set the target buffer size**
  - You can select a predefined value of 1024, 2048, 4096, or 10240 kb. If the set buffer size is insufficient, the oldest trace data is overwritten to accommodate new data

T-trace Dialog

- **5. Main button - allows you to control the tracing process based on the current operation state:**
  - **Ready to trace**
    - When the Ready to trace button is displayed, click it to start tracing based on the selected tags and settings
  - **No Connection**
    - When the No Connection button is displayed, you cannot perform any trace operations
    - Connect a target device to the computer to see the Ready to trace button
  - **Waiting**
    - When the Waiting button is displayed, the T-trace is working on the target and you must wait for it to finish
  - **Stop**
    - When the Stop button is displayed, click it to stop the tracing process, This button is displayed when the trace operation is started with the time period set to manual
  - **Show result**
    - When the Show result button is displayed, the tracing process is finished. Click the button to run the viewer

Running the T-trace

- When you run the T-trace, the tracing process gathers traces during a specified time period
- After tracing is finished, the T-trace processes the traces and creates a trace report in the HTML format
  - To operate the T-trace, you must first install Python(2.7.x) and Google Chrome on your computer

Running the T-trace

❒ To run the T-trace in the Tizen IDE:

- Set up the target device for the debugging mode and connect it to your computer with a USB cable
- Install your application on the target device
- In the Project Explorer view, right-click the project
- Select Profile as > Profile With T-trace
- The T-trace for TizenSDK dialog opens
- In the dialog, set the tracing options and click Ready to trace
  - You can also run the T-trace from the command line
  - The T-trace script is located in the <TIZEN_SDK>/tools/ttrace directory, and it requires that Python(2.7.x) or later to be installed on your computer

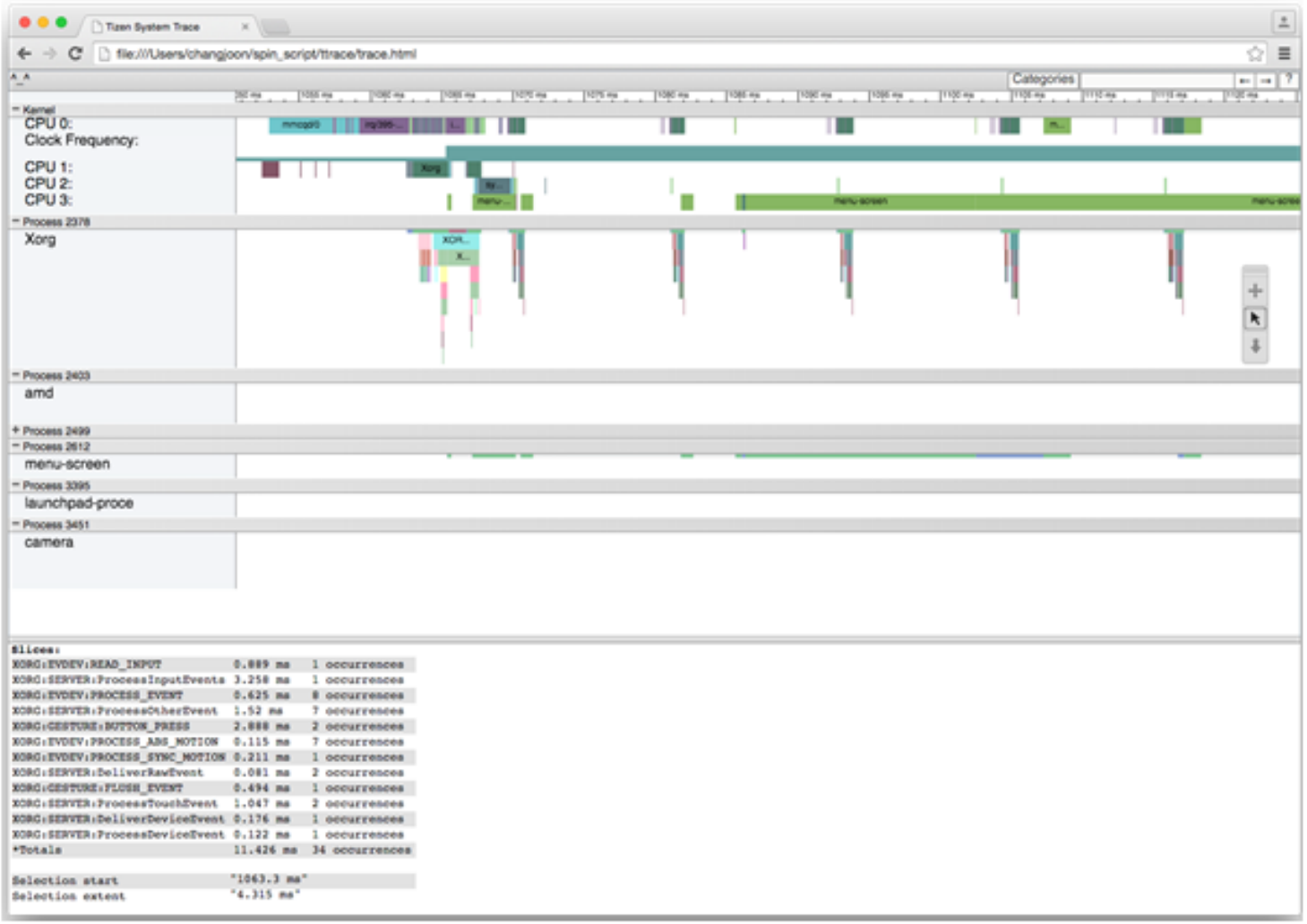
Running the T-trace

- To run the T-trace from the command line:
  - Open command prompt and move to the T-trace script directory:
    - $ cd TIZEN_SDK_HOME/tools/ttrace
  - Run the T-trace script with applicable options:
    - $ python ttrace.py –-time=10 –-buf-size=10240 –o output_filename.html
  - For more information on the command options, access the help:
    - $ python ttrace.py --help

Viewing the Tracing Result

- The results are stored in the <TIZEN_SDK>/tools/ttrace/trace directory
- Both a .text binary-format trace file and a.html result report file are generated
  - The files are named with a timestamp (YYYYMMDDHHMMSS)
- You can view the results using the Google Chrome browser as a viewer
  - The viewer is launched with the result report when you click Show result in the T-trace dialog
  - If you run the T-trace from the command line, open the result report manually in the viewer

T-trace Result



Assignment 3/4: T-trace

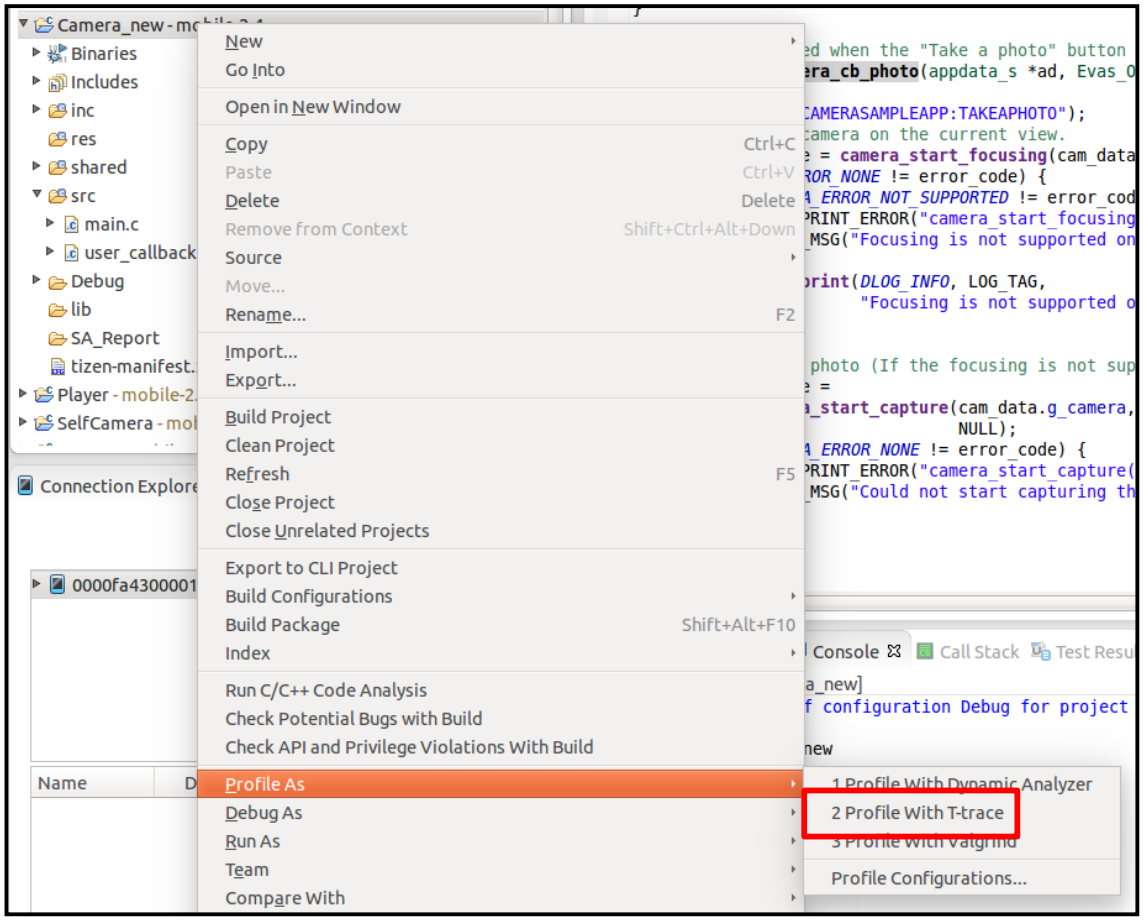
Assignment 3: T-trace

- Inserting synchronous tracepoint at Camera Sample Application
  - Add trace at take a photo and preview callback function

```
// Callback invoked when the "Take a photo" button is clicked.
static void __camera_cb_photo(appdata_s *ad, Evas_Object *obj, void *event_info)
{
    trace_begin("CAMERASAMPLEAPP:TAKEAPHOTO");
    // Focus the camera on the current view.
    int error_code = camera_start_focusing(cam_data.g_camera, false);
    if (CAMERA_ERROR_NONE != error_code) {
        if (CAMERA_ERROR_NOT_SUPPORTED != error_code) {
            DLOG_PRINT_ERROR("camera_start_focusing()", error_code);
            PRINT_MSG("Focusing is not supported on this device. The picture will be taken without focusing.");
        } else {
            dlog_print(DLOG_INFO, LOG_TAG,
                       "Focusing is not supported on this device. The picture will be taken without focusing.");
        }

        // Take a photo (If the focusing is not supported, then just take a photo, without focusing).
        error_code =
            camera_start_capture(cam_data.g_camera, _camera_capturing_cb, _camera_completed_cb,
                                 NULL);
        if (CAMERA_ERROR_NONE != error_code) {
            DLOG_PRINT_ERROR("camera_start_capture()", error_code);
            PRINT_MSG("Could not start capturing the photo.");
        }
    }
    trace_end();
```

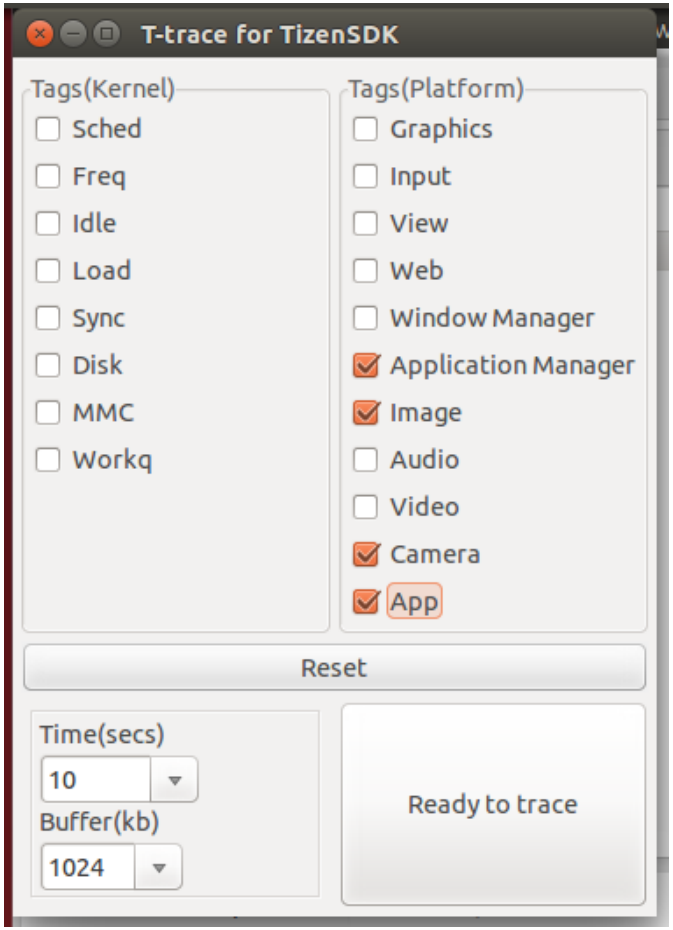
Assignment 3: T-trace

❒ Running T-trace with Tizen SDK



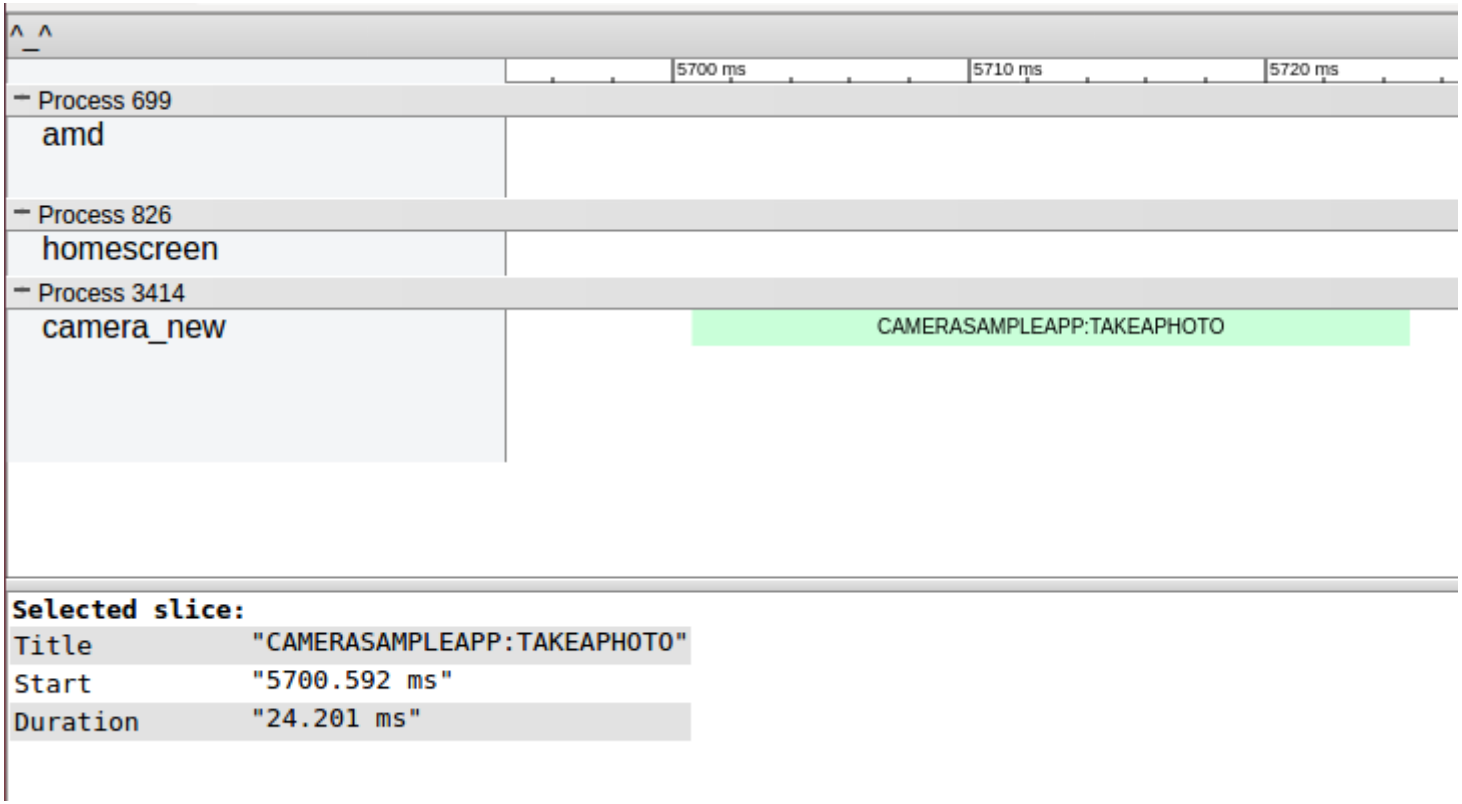
Assignment 3: T-trace

- Select Tags, Time, and Buffer and Start Trace, after that show tracing result



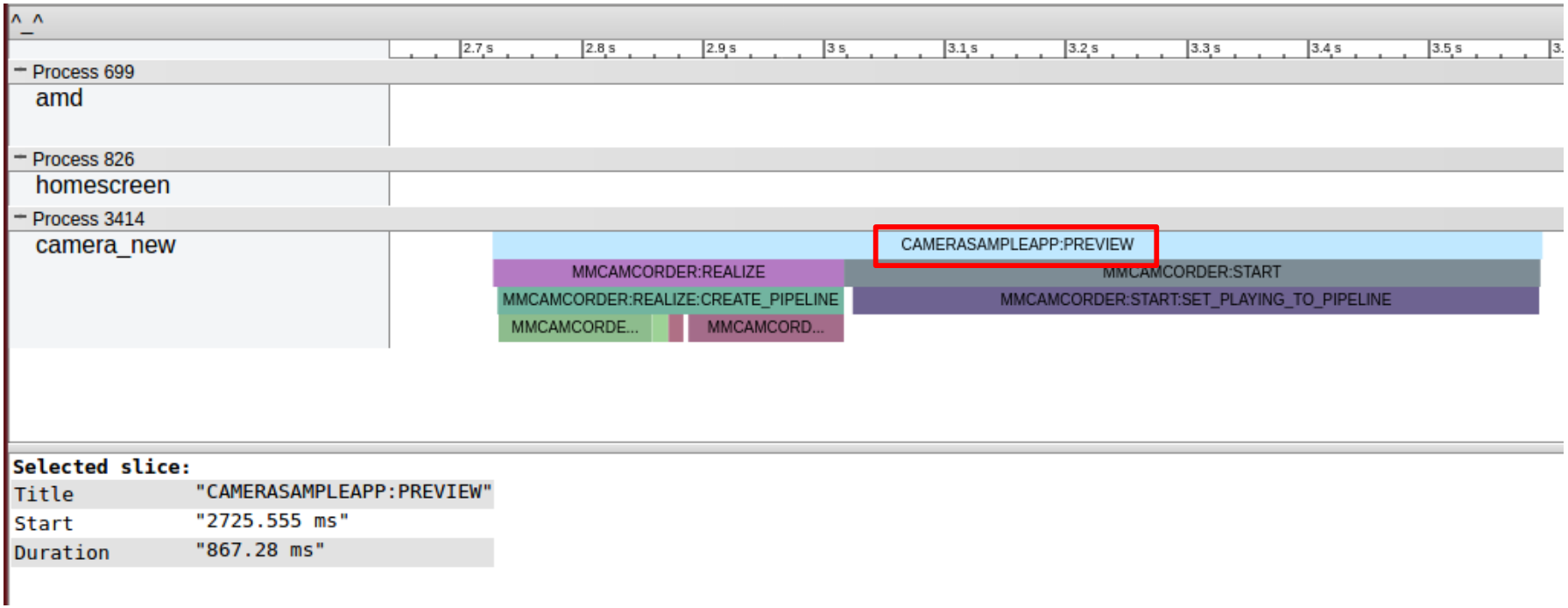
Assignment 3: T-trace

- ‘Take a photo’ callback function tracing result



Assignment 3: T-trace

- ‘Preview’ callback function tracing result
  - mmcamcorder tracing points also showed.
    - It means there are already tracing points at Tizen camera framework
  - Framework/multimedia/libmm-camcorder, mm_camcorder.c



Assignment 4: T-trace

❒ Insert asychronous tracepoint thread application

- Add trace point at _thread_start callback function

```
void _thread_start(appdata_s *ad, Evas_Object *obj, void *event_info)
{
    int ret;
    int err;
    void *tret;

    int thread1_count = 0;
    int thread2_count = 0;
    usleep(300*1000);

    trace_async_begin(thread1_count, "THREADSAMPLEAPP:THREAD1");
    if (!pthread_create(&thread_id_1, NULL, thread_run_1, NULL)){
        PRINT_MSG("pthread_create_1 success\n");
    }
    else
    {

        PRINT_MSG("pthread_create_1 fail\n");
    }
    PRINT_MSG("pthread_1 after sleep\n");
    usleep(300*1000);
    usleep(300*1000);
    trace_async_begin(thread2_count, "THREADSAMPLEAPP:THREAD2");
    if (!pthread_create(&thread_id_2, NULL, thread_run_2, NULL)){
        PRINT_MSG("pthread_create_1 success\n");
    }
    else
    {
```

- Add trace_async_begin at each thread_create
- Use different event name

Assignment 4: T-trace

- At each thread, insert trace_update_count function for tracking counter
- Add trace_async_end at each thread end point

```c
static void *thread_run_1(void *arg)
{
    int i;

    for (i = 0; i < 10; i++)
    {
        pthread_mutex_lock(&mutex);
        PRINT_MSG("ncount =%d\n",ncount);
        ncount ++;
        trace_update_counter(ncount, "THREADSAMPLEAPP:THREAD1");
        pthread_mutex_unlock(&mutex);
        usleep(200*1000);
    }
    trace_async_end(ncount, "THREADSAMPLEAPP:THREAD1");
    pthread_exit(NULL);

    return NULL;
}
```

<Thread1>

```c
static void *thread_run_2(void *arg)
{

    int i;

    for (i = 0; i < 10; i++)
    {
        pthread_mutex_lock(&mutex);
        PRINT_MSG("ncount =%d\n",ncount);
        ncount ++;
        trace_update_counter(ncount, "THREADSAMPLEAPP:THREAD2");
        pthread_mutex_unlock(&mutex);
        usleep(200*1000);
    }
    trace_async_end(ncount, "THREADSAMPLEAPP:THREAD2");
    pthread_exit(NULL);

    return NULL;

}
```

<Thread2>

Assignment 4: T-trace

❒ Run t-trace, select dialog setting, and show result

Selected counter:

Title	"THREADSAMPLEAPP:THREAD1"
Timestamp	"3145.142 ms"
1	7

counter value