# Exercise I: Image Gallery

This exercise is inspired by one of the many joys of being programmer at a company: We need to “fix” an existing program, a very small image gallery. Sorry, I wanted to add more drama but it was sold out.

Someone else wrote it and it gets the job done. But a customer reported that it doesn't work. It's “laggy” or just stops working. As luck would have it the original programmer isn't available. He already left the company or (this also sometimes happens) is on holiday. So the task of “fixing” the image gallery is bestowed upon us. Best of luck to you, young Padawan.

## 1. Get ready

First of we need to get the thing running and reproduce the “laggy” behavior the customer described. Fortunately the customer provided the set of images that caused the trouble.

- Download the program i_image_gallery.c as well as the archives i_dev_images.zip and i_user_images.zip from the personal schedule.
- Extract the two ZIP archives into two directories.
- Compile i_image_gallery.c GUI program with the usual command (see cheatsheet).
- Run the program. Enter the path to the extracted i_dev_images directory into the text field and press the “load” button. These are the images the original developer used for testing. Everything should work just fine.
- Run the program again. This time enter the path to the extracted user-images directory. Try to figure out if the program is still responsive or not. If it's not we've reproduced the customers problem.

## 2. Optional: Figure out what blocks the event loop

I already spilled the beans on this one in the lecture. So you can skip this step if you already know and don't want to practice taking time measurements. If you have no idea why the image gallery is unresponsive continue and we'll figure it out.

Remember the time measurement stuff from the memory management exercise (exercise B)? We'll do the same here. We know that the program is unresponsive when we press the “load” button. When that button is pressed the load_gallery_cb() callback is executed and this indicates that the callback takes way to long to execute.

As with all callbacks in a GUI program the execution of one callback has to be fast (couple of milliseconds, max) or the program will be laggy or unresponsive. When the callback takes it's sweet time the program can't return to the event loop to process other events. Making the program unresponsive as long as the callback runs. In this step we'll try to figure out what parts of the load_gallery_cb() callback take how long.

- First we need some tools for time measurement. We'll use the gettimeofday() and sec_between() functions from exercise B.

  To set them up add sys/time.h to the #include lines at the top of the program (so the compiler knows gettimeofday()). Then insert the following function into your program:

```
double sec_between(struct timeval* start, struct timeval* end) {
    return (end->tv_sec - start->tv_sec) +
           (end->tv_usec - start->tv_usec) / 1000000.0;
}
```

- Measure the entire execution time of the load_gallery_cb() callback. Print the measured time at the end of the callback. You can use the following snippet as inspiration on how to do it:

```
struct timeval foo_start, foo_end;
double foo_duration = 0;
…
gettimeofday(&foo_start, NULL);
    // code we want to measure goes here
gettimeofday(&foo_end, NULL);
foo_duration += sec_between(&foo_start, &foo_end);
```

- Run the program with the images in the user-images directory. Think about what the measured execution time means for the responsiveness of the application.
- Measure individual steps of the callback. The parts of the code with the “// Step X: ...” comments. For steps in the for loop accumulate the time into one double variable per step like in the snippet above. Otherwise you'll just track the time of the last loop iteration and not the accumulated time of all iterations.. So in the end we know how long e.g. the resizing step of all images took.
- Run the program again with the images in the user-images directory. Look at the measured time of the different steps in the callback. If a step only took a couple of milliseconds it doesn't hurt the responsiveness. If a step takes longer that that we have to get them out of the callback. Those pieces of code will be our targets for the next part of the exercise.

## Hints

These functions may be helpful to use. Use the manpage to get further information:

- gettimeofday()

## Includes

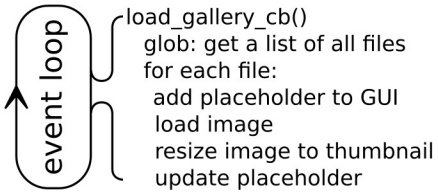
```
#include <sys/time.h>
```

## 3. Interlude: How to kick that stuff out of the event loop

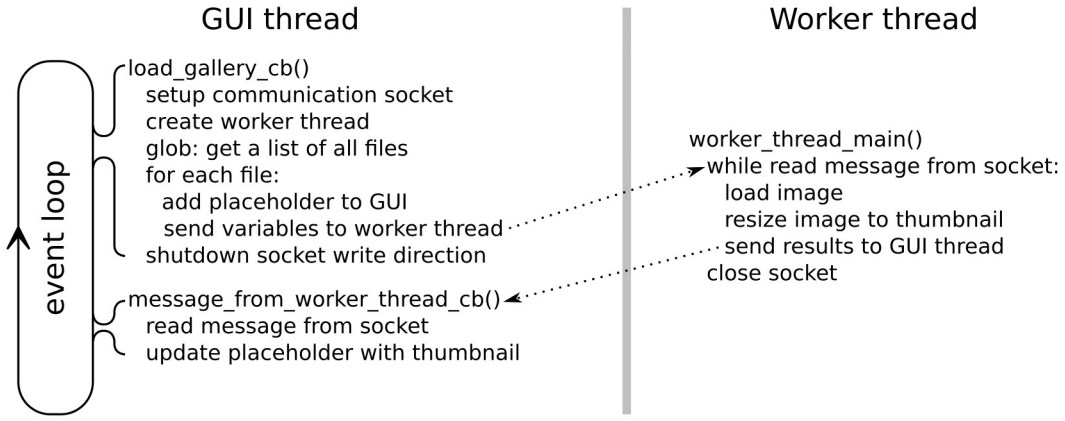
The load_gallery_cb() callback spends pretty much all its time loading and resizing images. We need to get this out of the event loop so the callback finishes quickly and the event loop can continue to respond to other events.

In the previous network chat exercise we had a similar problem: While waiting for a client to connect we blocked our event loop and the program was unresponsive. We fixed that by telling the event loop via IupLoopAddWatch() to wait for the connection instead of blocking the entire program until it happened.

But this time our event loop isn't blocked by waiting for something to happen. Instead loading and resizing of images actually takes a lot of CPU cycles. It's complex work and it simply takes a long time. This is different from before: Our load_gallery_cb() callback isn't waiting for something. It's actually doing real calculations the entire time. Right now our program looks like this:




To get the calculations out of the event loop we need to move them to someone else who can do the work but doesn't block the event loop: To a new thread. Our load_gallery_cb() then tells this worker thread to load and resize the images. Every time a thumbnail is ready the worker thread sends our GUI thread a message and the GUI can display the thumbnail. This is often called doing stuff “in the background”. When we've done that our application should look like that:



We'll need to do a lot of honest work before we see results:

- We need to create a new thread
- Setup the communication between the GUI and worker thread (similar to what our chat program did, but simpler)
- Do the actual sending and receiving of the messages
- And move the “update placeholder with thumbnail” step into the message_from_worker_thread_cb() callback.

Be prepared that this takes a while.

This “worker thread” approach has shown itself to work well in situations like ours. But remember: It's not a silver bullet for all situations! For low-level multi-threaded data structures mutexes and semaphores are more appropriate (e.g. sometimes in games). Other situations simply might require other tools.

But for GUI programs, mobile apps and web applications a worker thread is a very good approach to handle CPU intensive work. The details and APIs vary but the idea is always the same. It's also easy to extend it to use multiple available CPU cores as we'll do later in the exercise.

Let's start with the surgery!

## 4. Create a new thread

First of we need our worker thread. This involves two pieces of code: A new function where we put all the code the thread should do. This is similar to the main() function of our program. Second we have to tell the operating system to execute that function in a new thread. On Linux this is done by the pthread functions.

- To create the function for our worker thread take the following code and add it to your program:

```
void* worker_thread_main(void* arg) {
    printf("worker thread: started\n");
    // Code executed by thread goes here
    printf("worker thread: finished\n");
    return NULL;
}
```

  The name can be different but the argument and return type has to be the same.

- On Linux the pthread functions are used to work with threads. Include “pthread.h” so the compiler known the functions. Just add it to all the other include lines.

- The following code tells the operating system to execute that function in a new thread. Add it to the appropriate location in our load_gallery_cb() callback (see the diagram in part 3 above):

```
pthread_t thread_id;
pthread_create(&thread_id, NULL, worker_thread_main, 0);
pthread_detach(thread_id);
```

  When pthread_create() is called the operating system will execute worker_thread_main() in a new thread. Whatever we'll put into the last argument (here “0”) will later be available in the new thread as the “arg” parameter of the worker_thread_main() function. We'll later replace “0” with with a variable we need to give to the newly born thread.

  The pthread_detach() call tells the operating system to throw away the return value of the thread. Otherwise it would keep finished threads alive in a zombie state until we looked up their return value (no kidding, they're really called zombies).

- Compile the program. You'll have to add the “-pthread” compiler option since the compiler has to do some extra stuff for threaded programs. The entire compile command looks like this (it's still all in one line):

```
gcc -Wall -std=c99 -pthread i_image_gallery.c -o i_image_gallery
gui.a $(ar -p gui.a LDLIBS)
```

T Run the program. When you click the load button you should now see the worker thread started and finished messages. Meaning our thread gets started and is almost instantly done since it doesn't do anything yet.

### Hints

These functions may be helpful to use. Use the manpage to get further information:

- pthread_create()
- pthread_detatch()

### Includes

```
#include <pthread.h>
```

## 5. Setup the communication

To communicate with the worker thread we'll use messages, small blocks of data. This way we can send and receive messages between the GUI and worker thread. Linux offers something that is very similar to what we already did: A UNIX domain socket. This is a connection similar to the network connection we used in our chat program. But it's optimized for communication between threads and processes. It's also way simpler to setup. :)

- To create a UNIX domain socket add the following code to your load_gallery_cb() callback (see the diagram in part 3 above for where to put it):

```
int fds[2];
socketpair(AF_UNIX, SOCK_SEQPACKET, 0, fds);
int to_worker_thread_fd = fds[0];
int to_main_thread_fd = fds[1];
```

  Now the to_worker_thread and to_main_thread variables contain the two ends of the socket connection. Just like in the chat they're file descriptors we can read() and write() from or watch for pending data. Unlike our chat both ends are already connected so there's no need for all that listen() and connect() stuff. We can send and receive messages right away. We just have to give one end to the thread we want to communicate with and we can exchange messages with it.

- We'll use the “arg” parameter of the worker thread to give it the “to_main_thread” end of the connection. To do that replace the “0” parameter so the pthread_create() call looks like this:

```
pthread_create(&thread_id, NULL, worker_thread_main, (void*)(long)to_main_thread_fd);
```

  That puts the file descriptor of the connection into the worker thread parameter. When the worker thread starts in its worker_thread_main() function that value is inside the “arg” parameter. Pull it out in the following way:

```
void* worker_thread_main(void* arg) {
    int to_main_thread_fd = (int)(long)arg;
    printf("worker thread: started\n");
    // Code executed by thread goes here
    printf("worker thread: finished\n");
    return NULL;
}
```

T The GUI thread can send messages by write()ing on the to_worker_thread_fd file descriptor. The worker thread can wait for and receive messages by read()ing on the to_main_thread_fd. It can send messages to the GUI thread by write()ing on the to_main_thread_fd.

But our GUI thread is managed by an event loop. So we can't use read() to wait for and receive messages. We would block the event loop! Instead we just tell the event loop to look for data on that file descriptor and call a callback when it's available. This is the same we did in our network chat.

First we need a new function the event loop will call when our GUI thread receives a message from the worker thread. Insert this callback into your code:

```
int message_from_worker_thread_cb(int to_worker_thread_fd, short revents) {
     // TODO: read message from worker thread and react to it
     return IUP_CONTINUE_WATCH;
}
```

T And lastly we have to tell the event loop to call message_from_worker_thread_cb() when data is available on the to_worker_thread_fd file descriptor. Just like in our chat program the IupLoopAddWatch() function does that:

```
IupLoopAddWatch(to_worker_thread_fd, POLLIN, message_from_worker_thread_cb);
```

Insert this call after creating the UNIX domain socket but before creating the worker thread. This way everything to receive messages in the GUI thread is in place before we start the worker thread.

T Compile the program and see if everything still works. You might have to add an extra include line.

## Hints

These functions may be helpful to use. Use the manpage to get further information:

- socketpair()

## Includes

```
#include <sys/socket.h>
```

# Operating Systems

## 6. Sending and receiving messages

The communication infrastructure is in place. Now it's time to actually put code into the worker thread, send it work and react to the results.

- First of move all the code we want to have in the worker thread into the worker_thread_main() function. That's the code that loads and resizes images and blocked our event loop. Move the entire code blocks labeled with the “// Step 3: ...” and “// Step 4: ...” comments.

- The code in our worker thread won't work just now. It needs some information to do it's job. Step 2 (loading the image) needs the variable “img_path”. That's the filename of the image to load. In a similar fashion step 3 (resizing image to thumbnail) stores its result in the variables “thumb_w”, “thumb_h” and “thumb_ptr”. We'll also need the “thumb_img_label” variable so we know which GUI element we should update with the finished thumbnail. We need these variables back in the GUI thread to display the thumbnail in the GUI.

  So what we need is a way to send packets of variables between our GUI and worker thread. One way to do this is to put all these variables into a struct and send those structs as messages. While our chat program send text we're now going to send blocks of data, small packages of variables.

  First of define this struct in your program:

```
struct message {
    char* img_path;            // File to load
    Ihandle* thumb_img_label;  // GUI element for the thumbnail
    int thumb_w, thumb_h;      // Dimensions of the thumbnail
    unsigned char* thumb_ptr;  // Pointer to the thumbnail data
};
```

- Now to send a message we create a variable of that struct, set its members to the values we want to send and write() it to the socket file descriptor.

  In the load_gallery_cb() callback we iterate over all the files in the directory. For each file we have to send the “img_path” variable to the worker thread so it knows which image it should load. Use the following snippet to do it. Looking at the diagram in part 3 above might help.

```
struct message msg = { 0 };
msg.img_path = strdup(img_path);
msg.thumb_img_label = thumb_img_label;
write(to_worker_thread_fd, &msg, sizeof(msg));
```

T After we send all the messages in load_gallery_cb() we better tell the worker thread that we're done. We've already send one message per file and we won't send any more. This way the worker thread knows when it has done all the work and can terminate itself.

As with most stuff there's an operating system function for exactly that. And as usual it has a funny name:

```
shutdown(to_worker_thread_fd, SHUT_WR);
```

This doesn't shutdown your computer! Instead it shuts down the write direction of the connection. Insert that call in your load_gallery_cb() callback after you've send all the messages.

After that function we can no longer write() on the connection. And after the worker thread read() all the messages read() will return 0 (end of file) since there can't be any more data to read.

T The GUI thread now sends messages to the worker thread. One message for every image the worker thread has to load and resize. Time to receive those messages in the worker thread. This works by read()ing on the connection until read() returns 0, meaning that we've read all messages. Implement that in your worker_thread_main() function. It should look similar to that:

```
void* worker_thread_main(void* arg) {
    int to_main_thread_fd = (int)(long)arg;
    printf("worker thread: started\n");

    struct message msg;
    while ( read(to_main_thread_fd, &msg, sizeof(msg)) > 0 ) {
         // msg contains the variables we've send in the GUI
         char* img_path = msg.img_path;

         // Code for image loading and resizing
    }

    close(to_main_thread_fd);
    printf("worker thread: finished\n");
    return NULL;
}
```

The final close(to_main_thread_fd) makes sure that the operating system knows we're not going to use that end of the connection any longer. That our worker thread won't send any more messages.

T We send messages from the GUI, receive it in the worker thread and do the work there. Now it's time to send the finished work back to the GUI thread. We need to do this after the worker thread has loaded and resized a particular image. The code is similar to how we already send the messages from the GUI thread to the worker thread. But this time we take the received message, store the results in it and send it back to the GUI thread:

```
msg.thumb_w = thumb_w;
msg.thumb_h = thumb_h;
msg.thumb_ptr = thumb_ptr;
write(to_main_thread_fd, &msg, sizeof(msg));
```

T After that the message should travel back to the GUI thread. Thanks to our IupLoopAddWatch() call quite a while ago the event loop there will see that data is pending on the GUI threads end of the connection. Seeing that it will call our message_from_worker_thread_cb() callback.

In that callback we can read the message and finally have the results back in the GUI thread so we can display them. To read the message we just have to call read() on the GUI end of the connection like this:

```
struct message msg;
if ( read(to_worker_thread_fd, &msg, sizeof(msg)) == 0 ) {
    return IUP_REMOVE_WATCH;
}
// Code that reacts to the message
```

Here the twist is that when read() returns 0 the worker thread is done and won't send any more messages. In that case the watch is also useless (because we can't receive anything anymore) and we can remove it. Also we just read one message since the event loop will call our callback as long as there are messages to read.

T Finally after we've received the message we can update the GUI with the finished thumbnail. For that move the entire code block labeled “// Step 5: ...” to the message_from_worker_thread_cb() callback. Right after you've received the message. Instead of the local variables for “thumb_w”, “thumb_h”, etc. make the code use the fields of the “msg” structure directly (e.g. “msg.thumb_w” instead of “thumb_w”).

T Finally compile the whole thing and test it with the user_images directory. The program should stay responsive while the thumbnails appear, slowly, one after the other.

### Hints

These functions may be helpful to use. Use the manpage to get further information:

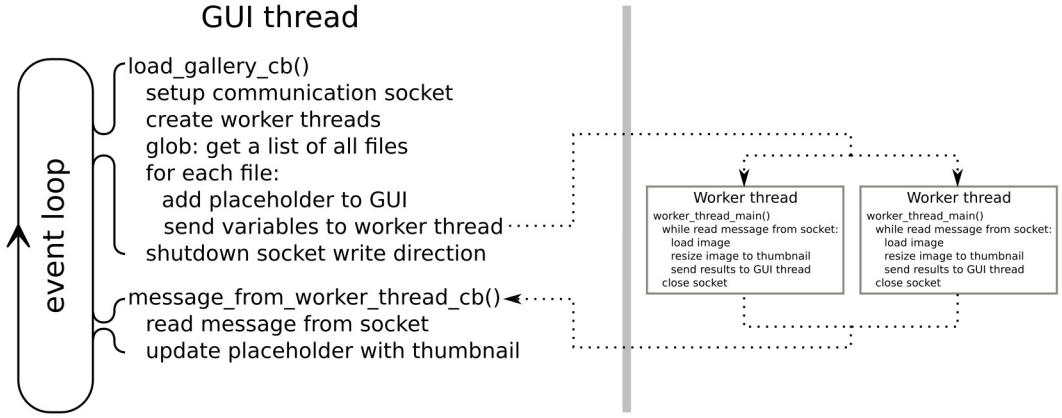
- read()
- write()
- shutdown()
- close()

### Includes

```
#include <unistd.h>
#include <sys/socket.h>
```

## 7. Using multiple CPU cores

We've restructured our image gallery to do all the CPU intensive heavy lifting in a worker thread. One of the nice advantages of the way we did that is that we can easily use more than one worker thread:




T Use the following snippet to create two instead of one worker thread:

```
pthread_t worker_thread_id;
pthread_create(&worker_thread_id, NULL, worker_thread_main, (void*)(long)dup(to_main_thread_fd));
pthread_detach(worker_thread_id);
pthread_create(&worker_thread_id, NULL, worker_thread_main, (void*)(long)dup(to_main_thread_fd));
pthread_detach(worker_thread_id);
close(to_main_thread_fd);
```

All the dup()ing stuff makes sure every thread gets it's own duplicate of the to_worker_thread_fd end of the communication connection. After every thread closed their end of the connection the operating system knows that everyone is done and read() on the GUI thread returns 0. Otherwise the GUI thread wouldn't know all worker threads are done.

T Compile and run the program. Open the “System Monitor” program (press the Windows key and enter the name and press enter). Load the user_images directory in the image gallery. The system monitor should show you how the work is distributed over the available CPU cores.

T Use a for loop to create as many threads as there are CPU cores. You can use the following call to get the number of CPU cores of the system:

```
sysconf(_SC_NPROCESSORS_ONLN);
```

T Again, compile and run the program and load the user_images directory. This time all CPU cores should be busy.

T Now whenever we saturate all CPU cores with work it's a good idea to lower the priority of the background threads. Otherwise the background threads compete with our GUI thread and other programs for CPU time. But when the GUI thread receives an event it's more important for us than resizing just another image in the background.

To make a thread nicer to the rest of the system (or less important) we can use the nice() function:

```
nice(10);
```

Add this call to the beginning of your worker_thread_main() function. After this call all worker threads should have a low priority for the operating system.

## Hints

These functions may be helpful to use. Use the manpage to get further information:

- sysconf()
- nice()

## Includes

```
#include <unistd.h>
```