# Objectives - Thu 4/11/2019

- Code coverage
  - line coverage
  - branch coverage
  - path coverage
  - example
- Parallel programming - arrays of threads
  - `pthread_t`
  - `pthread_create(…)`
  - `pthread_join(…)`
  - `man pthread.h`
  - `gcc -pthread` [redacted]

## Types of test code coverage

- "Line coverage" means every line of the code being tested was executed at least once.
- "Branch coverage" means for every conditional jump (If/While/For/Switch), we took the jump (condition true) and did not take the jump (condition false) at least once.
- "Path coverage" means we tested every possible path through the code (unique combination of branches). *This can be hard.*

**line coverage $\subseteq$ branch coverage $\subseteq$ path coverage**

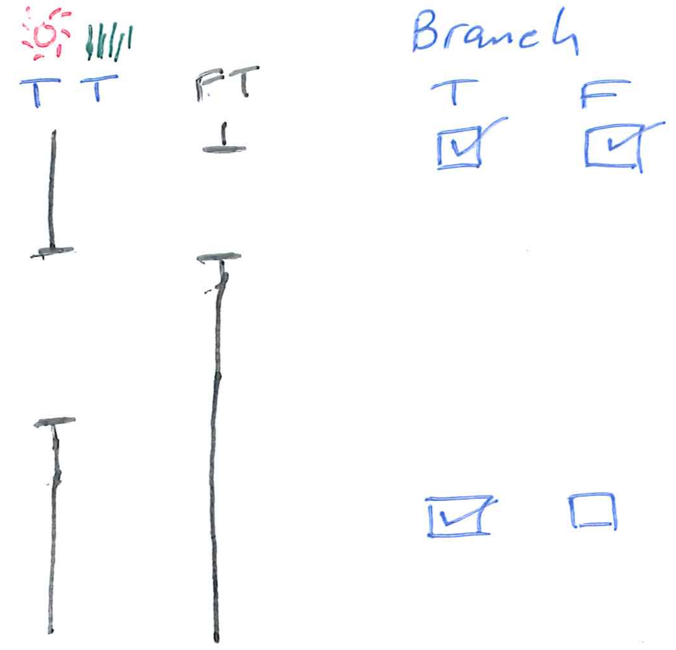
not checked in ECE 264

```c
////// IMPLEMENTATION CODE //////
void report_weather(bool is_sunny, bool is_raining) {
    if(is_sunny) {
        printf("The sun is shining.\n");
    }
    else {
        printf("The sun is not shining.\n");
    }

    if(is_raining) {
        printf("It is raining.\n");
    }
}
```

Branch 1

Branch 2



```c
/////////// TEST CODE ///////////
void test_report_weather_1() { // LINE coverage
    report_weather(true,  true);  // The sun is shining.  It is raining.
    report_weather(false, true);  // The sun is not shining.  It is raining.
}

void test_report_weather_2() { // BRANCH coverage
    report_weather(true,  true);  // The sun is shining.  It is raining.
    report_weather(false, false); // The sun is not shining.
}
```

Branch 1 Branch 2

```c
void test_report_weather_3() { // PATH coverage
    report_weather(true,  true);  // The sun is shining.  It is raining.
    report_weather(true,  false); // The sun is shining.  It is raining.
    report_weather(false, true);  // The sun is not shining.  It is raining.
    report_weather(false, false); // The sun is not shining.
}
```

## Parallel programming

## Parallel programming

| processes | threads |
|---|---|
| Often an application that you started explicitly<br><br>Ex: ./test_bmp<br>Ex: vim test_bmp.c<br>Ex: bash | Created by a process to perform part of the work that process is responsible for<br><br>Ex: Background spell-check<br>Ex: Process part of an image |
| Memory is separate from all other processes | Shares memory with other threads of the same process |

# Application of threads

- Suppose you need to run an image filter on a 4000x3000-pixel image.
- Each pixel takes 1 microsecond ($10^{-6}$ secs).
- If we process sequentially (one thread):
  - ≥12 seconds
- If we process in parallel with 12 threads:
  - ≥1 second per 1000x1000-pixel segment
  - ≥1 second to process entire image
- Note: We assume each pixel is independent.

`pthread_create(---)`

K

shift-k to get to man page in Vim.

<u>Comment</u>

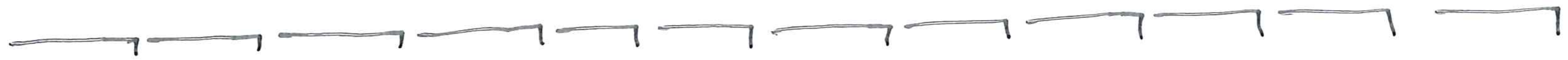
Visual select, then
gc

## Standard conversion of grayscale to B&W

Intensity $> 127 \rightarrow$ White $\underset{255}{}$

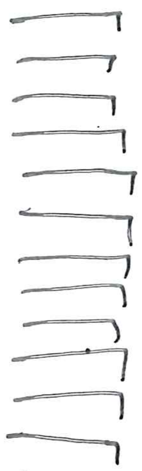
Intensity $< 127 \rightarrow$ Black $\underset{0}{}$

Sequential



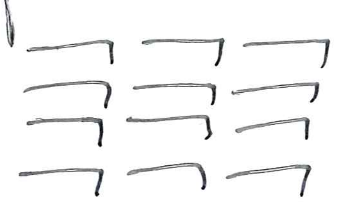
$\geq 12$ sec

parallel with 12 threads



$\geq 1$ sec

parallel with 4 threads



$\geq 3$ secs

# Code in paragraphs

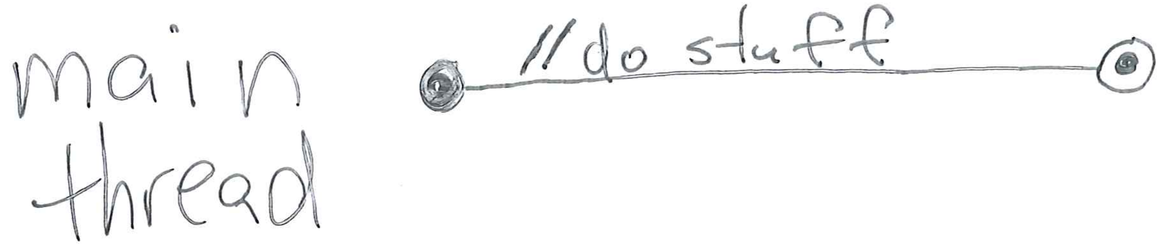
```
// Find pixel offset in new image
```

```
// copy pixel: b, g, r
```

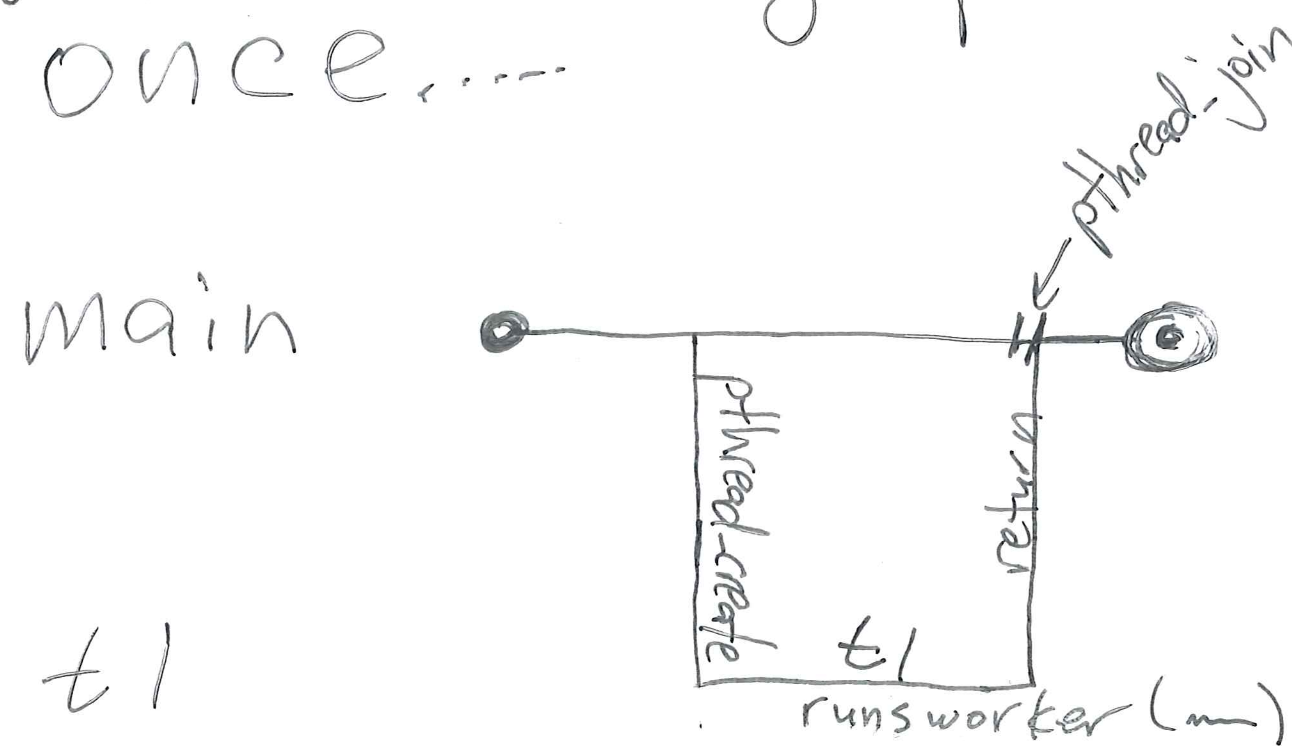
Feel Smarter!

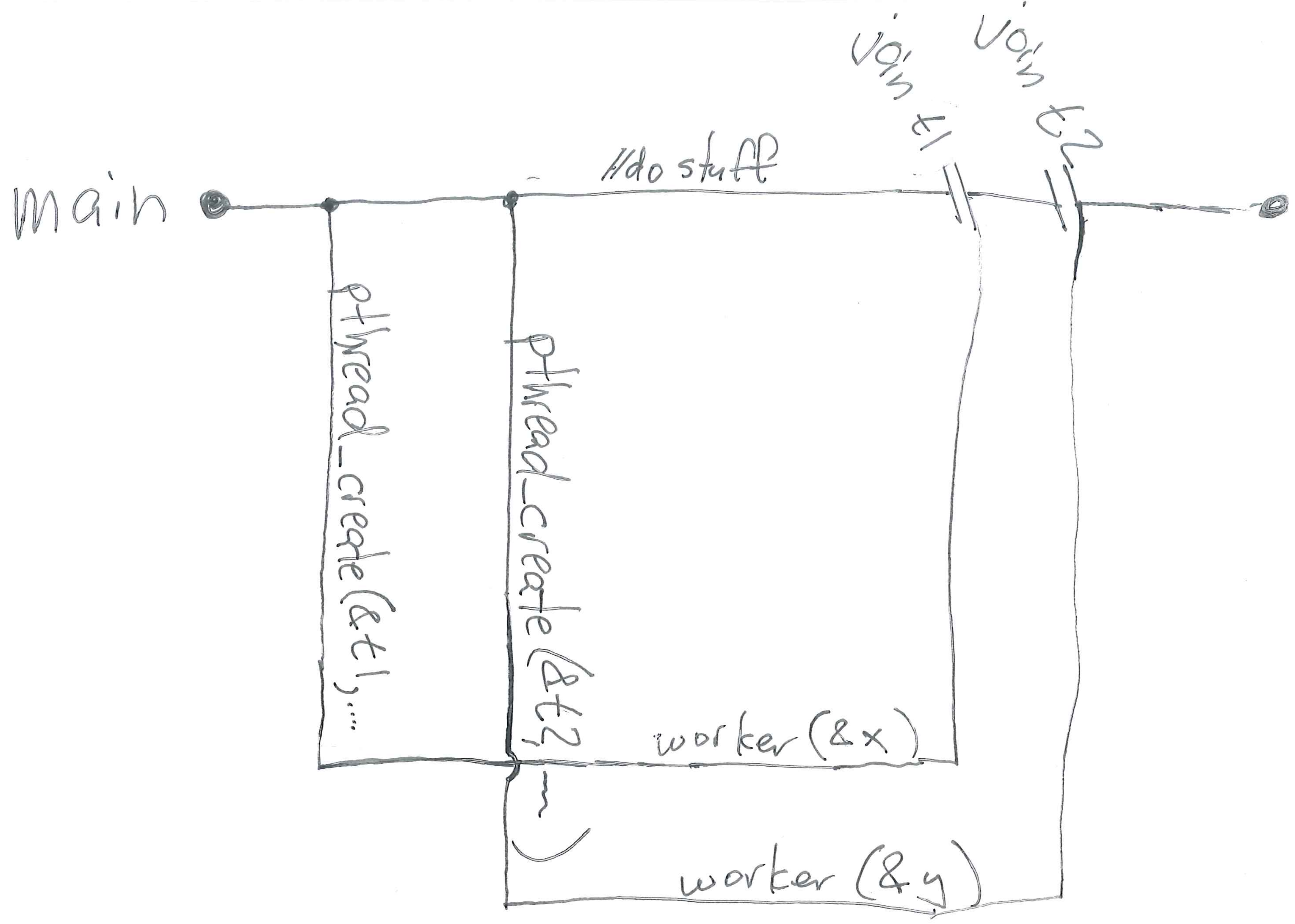
```
float calc_inverse(float n){
    return 1.0/n;
}
```

Until now ....



With calling pthread_create(...) once....





Each thread has its own stack, but memory is accessible.

Your job:

Ensure that no two threads write to the same data "at the same time"