# CSU23016

# Concurrent Systems
# &
# Operating Systems

*Andrew Butterfield*
*ORI.G39, Andrew.Butterfield@scss.tcd.ie*

**Trinity College Dublin**
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

*with thanks to Mike Brady*

# Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinity's statutory duties under the Universities Act, 1997 and in accordance with the University's policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at https://www.tcd.ie/info_compliance/data-protection/

# Overview

- Concurrency
  - What / Why / How,
  - Grappling with Concurrency Issues.
- Operating Systems
  - Operating system architectures,
  - Memory Management (OS perspective),
  - Processes and Thread Management,
  - File Storage (disk I/O and file systems).

# Practical Matters — Assessment

- 20% Assignments
  - Tool Usage 2%
  - Thread Programming 6%
  - Concurrency Modelling 6%
  - OS Coding 6% (Scheduler, Filesystem, ...) 6%
- 80% Examination
  - 2hour exam in a 24 hour window

# Practical Matters — Linux

- Course is based on Linux and the standard C program build toolset, "Build Essentials", POSIX Threads, Mac OS X also works pretty well.
- If you use Linux all will be well with all assignments
- If you don't use Linux (most of you?), then Assignment 1/2 are problematic
  - Suggest you either:
    - Develop your work on MacNeill or one of the LG12 machines
    - consider running Linux in a Virtual Machine - Virtual Box, say.
- Assignment 1 is simple to help smoke out issues
- Assignment 2 is tricky

# POSIX

- POSIX – Portable Operating System Interface
  - for variants of Unix, including Linux
  - IEEE 1003, ISO/IEC9945
- Really, considered a standard set of facilities and APIs for Unix.
  - 1988 onwards
  - Doesn't *have* to be Unix
    - macOS is certified to support POSIX, but ...
    - Windows: partial POSIX compliance with CygWin, Window Subsystem for Linux (WSL)
  - ref: http://en.wikipedia.org/wiki/POSIX

# POSIX Threads

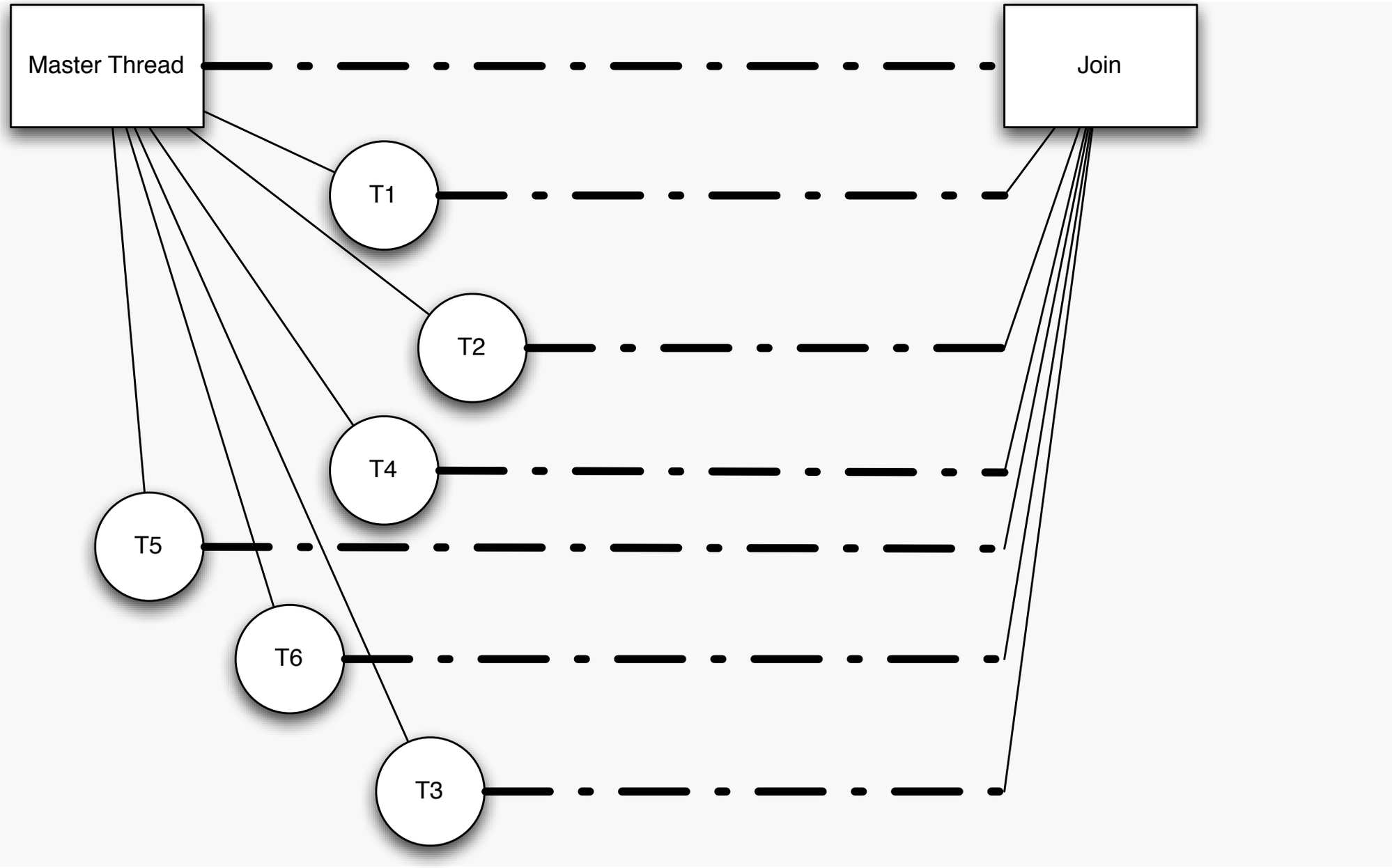
- POSIX Threads aka *'pthreads'*
  - correspond to 'Light Weight Processes' (LWPs) in older literature.
  - *pthreads* live within processes.
    - *processes* have separate memory spaces from one another
      - thus, inter-process communication & scheduling may be expensive
    - *pthreads* (within a process) share the same memory space
      - inter-thread communication & scheduling can be cheap

*Classic tradeoff: speed vs. stability/ruggedness*

# POSIX Threads (2)

- Portable Threading Library across Unix OSes
  - All POSIX-compliant Unixes implement *pthreads*
    - Linux, Mac OS X, Solaris, FreeBSD, OpenBSD, etc.
- Also Windows:
  - E.g. Open Source: *pthreads-win32*
  - But these are different, so are not suitable for assignments
- OS X is very strict
  - OS X implementation if pthreads is very strict - it can fail ways that Unix pthreads won't.

# Six Separate Threads



# Creating a pthread

```
#include <pthread.h>

int pthread_create(
      pthread_t *thread,            // the returned thread i
      const pthread_attr_t *attr, // starting attributes
      void *(*start_routine)(void*),
                        // the function to run in the thread
      void *arg         // parameter for function
);
```

# Where…

- '*thread*' is the ID of the thread
- '*attr*' is the input-only attributes (NULL for standard attributes)
- '*start_routine*' (can be any name) is the function that runs when the thread is started, and which must have the signature:

```
void* *start_routine (void* arg);
```

- '*arg*' is the parameter that is sent to the start routine.
- returns a status code. '*0*' is good, '*-1*' is bad.

# Wait for a thread to finish

- `int pthread_join( pthread_t thread,`
  `void **value_ptr`
  `);`

- where
  - *'thread'* is the id of the thread you wish to wait on
  - *'value_ptr'* is where the thread's exit status will be placed on exit (`NULL` if you're not interested.)
  - NB: a thread can be joined only to one other thread!

# Thread Code

```
void *PrintHello(void *threadid) {
     printf("\n%d: Hello World!\n", threadid);
     pthread_exit(NULL);
}
```

- `threadid` is a pointer to a number  (printed using %d)
- `pthread_exit` tells pthreads that has finished, and returns a pointer to a return value, if required.

# Hello World -- Creating Threads

```
int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    for (t=0;t<NUM_THREADS;t++) {
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc);
            exit(-1);
        }
    }
    ...
```

# Hello World -- Waiting for Exit

```
...
// wait for threads to exit
    for(t=0;t<NUM_THREADS;t++) {
        pthread_join( threads[t], NULL);
    }
    return 0;
}
```

# HelloWorld -- Complete

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 6

void *PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    for (t=0;t<NUM_THREADS;t++) {
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc);
            exit(-1);
        }
    }
    ...
    // wait for threads to exit
    for(t=0;t<NUM_THREADS;t++) {
        pthread_join( threads[t], NULL);
    }
    return 0;
}
```

# MacNeill/LG12/Ubuntu

- Compile hello.c as pthreaded executable "hello":

```
cc -o hello hello.c -pthread
```

  - Include the pthread library to allow it to compile and link
  - Sometimes, `-lpthread` works, but `-pthread` is correct
  - pthread library automatically included in Mac OS X build

- To run program "hello":

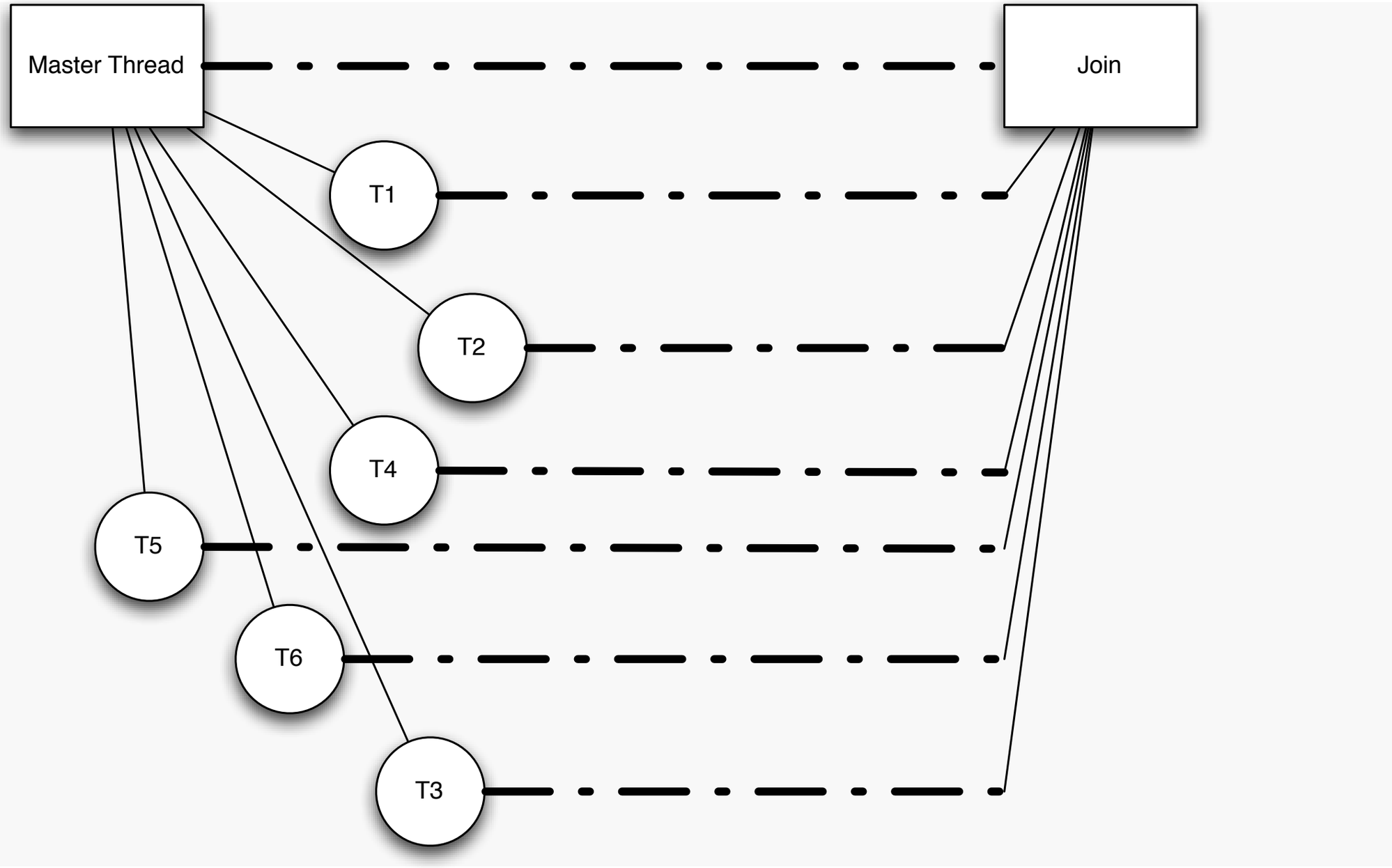
```
./hello
```

  - Let's try it (DEMO)

# HelloWorld -- Running it

- The order in which each thread is created is always the same
  - Determined by the for-loop we use to call `pthread_create`
- The order in which each printf statement executes varies
  - A lot!
  - Because ? .....

# What's curious about this?



# What if ... ?

- The threads interacted in some way?
  - e.g. each thread increments a global variable that tracks the number of threads that ran?
  - What could possibly go wrong?
- Consider a variant of hello.c that does this (`sumofhellos.c`)

sumofhellos.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 6
#define USERNAME "username"

int x; // Global variable !!

void *PrintHello(void *threadid){
    int y ; // (Thread) local variable

    y = x; // read that global
    printf("\n%d: Hello World, from %s!\n",threadid,USERNAME );
    x = y+1; // write that global
    pthread_exit(NULL);
}

int main(int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    x = 0; // Initialise that global !
    for (t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc );
            exit(-1);
        }
    }
    // wait for threads to exit
    for(t=0;t<NUM_THREADS;t++){
        pthread_join(threads[t],NULL);
    }
    // Display that global !
    printf("\nAll threads done by %s, x = %d\n",USERNAME,x);
    exit(0);
}
```

What should be
the final value of x ?

DEMO !

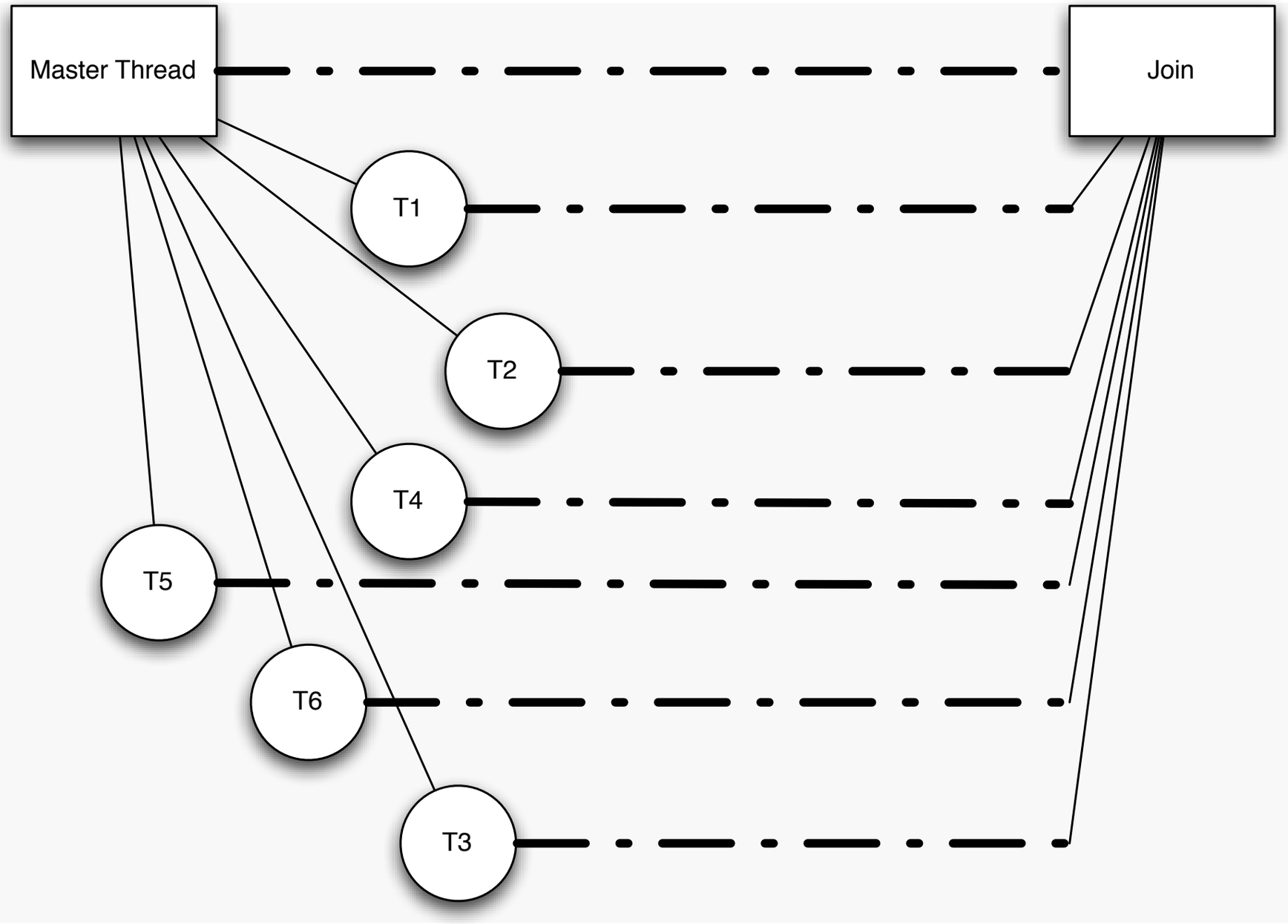
# Woah! Hold on

- The final value of x should have been 6.
  - The final value of x after a run varied, most often 3, 4, or 5
  - Less often, 1,2, or 6.
- What is going on?

# Runtime Behaviour

- The Runtime Behaviour of the Program is no longer under the control of the program.
  - The order in which work gets done on the machine is not exactly under the control of the program
  - What we observe is an (apparently) arbitrary interleaving of (atomic) actions by each running thread
  - It seems to be a price that's paid for parallelism, but:
    - What errors can it introduce?
    - Can we prevent them / protect against them / design them out?

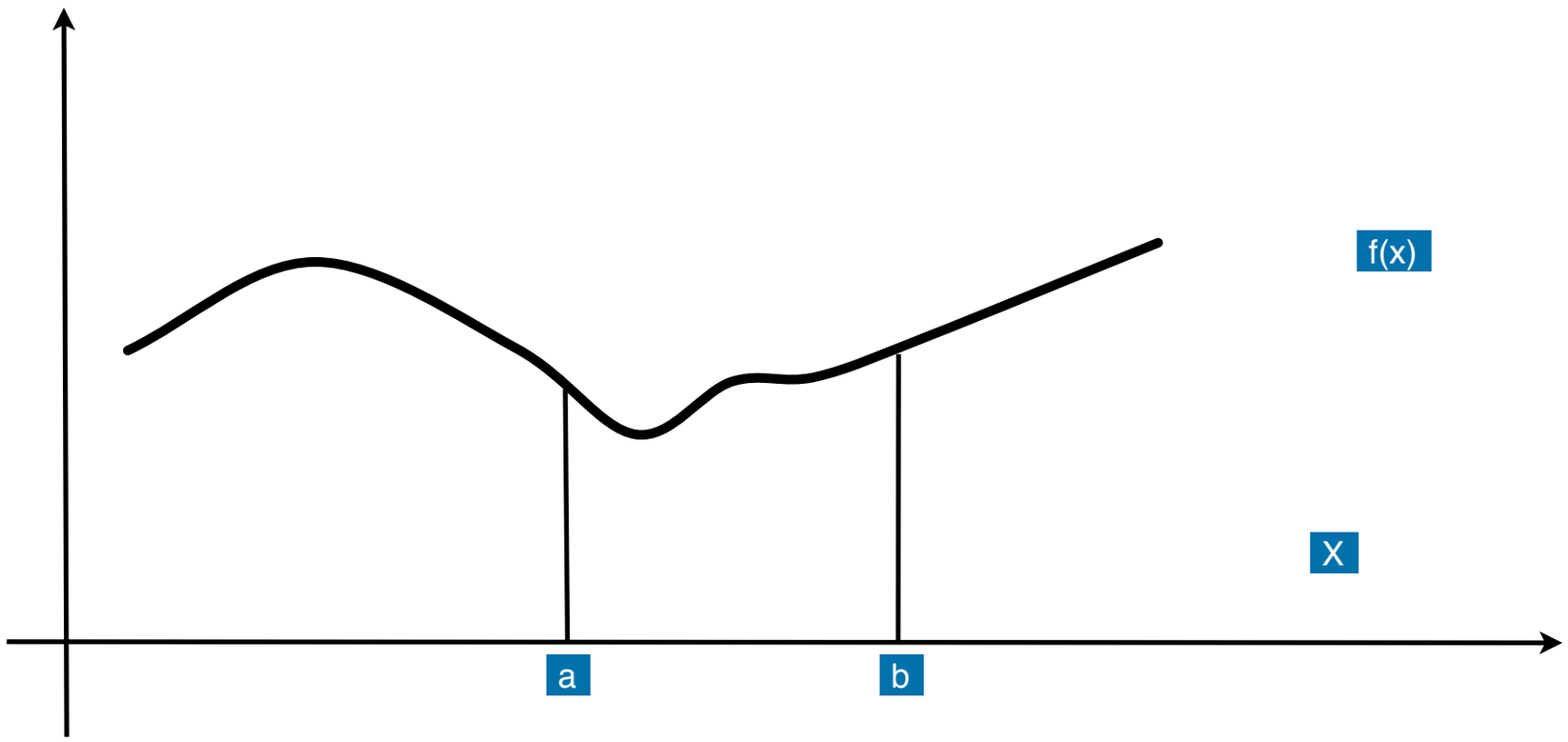
# What's ~~wrong~~ deceptive about this?



# “Massively” Parallel Problems

- A problem is said to be "Massively" Parallel if:
  - there are almost no interaction between parallel threads calculating independent parts of the solution.
  - So, in theory, everything can be run at the same time.

# Example: Numerical Integration




Numerically integrate from a to b

$$\int_0^4 (16 - x^2)dx = 42.6666\ldots$$

- Consider integrating f(x) = 16 - x^2 between 0 and 4
- The analytic solution is 128/3 or 42.666…(optional exercise: check my math)
- We break the interval [0..4] into a thousand vertical slices (`H`=0.004),
  - use the "trapezoidal rule" to calculate the `area` of each slice
  - sum all of these `areas` into an `answer` variable.
- Running code in `seq-integrate.c` gives the answer: 42.666656

# integration preamble

```
#include <stdio.h>

#define NUM_SLICES 1000
#define H 4.0/NUM_SLICES

double answer;

double f(double x) { return (16.0 - x*x) ; }

double trapezoid(double a, double b) { return H*(f(a)+f(b))/2.0; }
```

# seq-integrate.c body

```
int main (int argc, const char * argv[]) {
  answer    = 0.0;
  double a ;
  double b ;
  int i ;
  for (i=0;i<NUM_SLICES;i++) {
    a = (int)i * H ;
    b = a + H;
    answer += trapezoid(a,b);
   }
  printf("\nAnswer is %f\n",answer);
    return 0;
}
```

# Integration with threads (attempt 1)

- We might use 1000 threads, one for each of the trapezoid calculations.

# nom-integrate.c thread

```
void *IntegratePart(void *i) {

  double a,b,area;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with no mutex !!!!!
  answer=answer+area;

    pthread_exit(NULL);
}
```

# nom-integrate body

```
int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_SLICES];
    long rc,t;

  answer = 0.0;

    for (t=0;t<NUM_SLICES;t++) {
        rc = pthread_create(&threads[t],NULL,IntegratePart,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %ld\n",rc);
            exit(-1);
        }
    }
    for(t=0;t<NUM_SLICES;t++) {
        pthread_join( threads[t], NULL);
    }
  printf("%f\n",answer);
    return 0;
}
```

# Integration with threads (attempt 1)

- When I ran this I got 42.666656, most of the time
  - (e.g. 46 times out of 52)
- However sometimes I got an answer in the range 42.6026 .. 42.6038 !
  - (e.g. 6 times of of 52)
  - Approximately 11% of the time I got an answer that underestimates by -0.15%
    - on my office iMac, at least.
- Hard to explain this result. May differ on different machines.
- Let's give it a go! (DEMO)