

Parallelism and Threads

Further reading in Chapter 12.1-12.4 and Parallelism
slides part 1 on Edstem that includes data parallel
SIMD examples

Content based upon Dr. Bernhard Scholz

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Outline

- Forms of parallelism
 - Task-parallelism
 - Data-parallelism
- Examples
- POSIX Threads
 - Introduction
 - Thread creation, termination
 - Passing arguments to threads
 - Thread synchronization upon termination (`pthread_join`)
 - Thread scheduling
 - Basic Synchronisation with mutex

Example: preparing a salad



Tasks to do:



- tear leaves
- wash leaves
- chop leaves



- wash
- slice



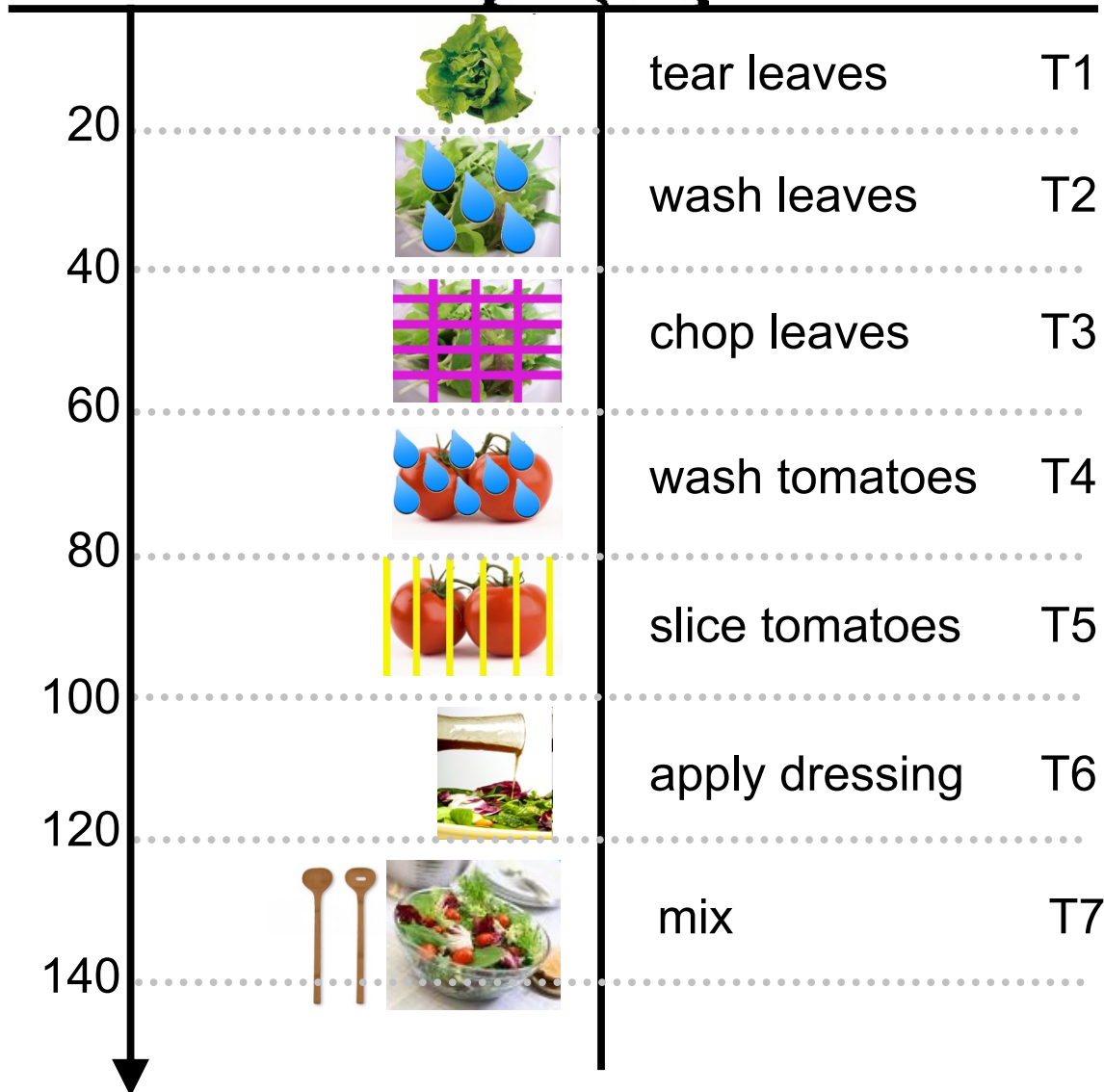
- apply dressing
- mix



execution time
(seconds)



Single Chef Salad (sequential)



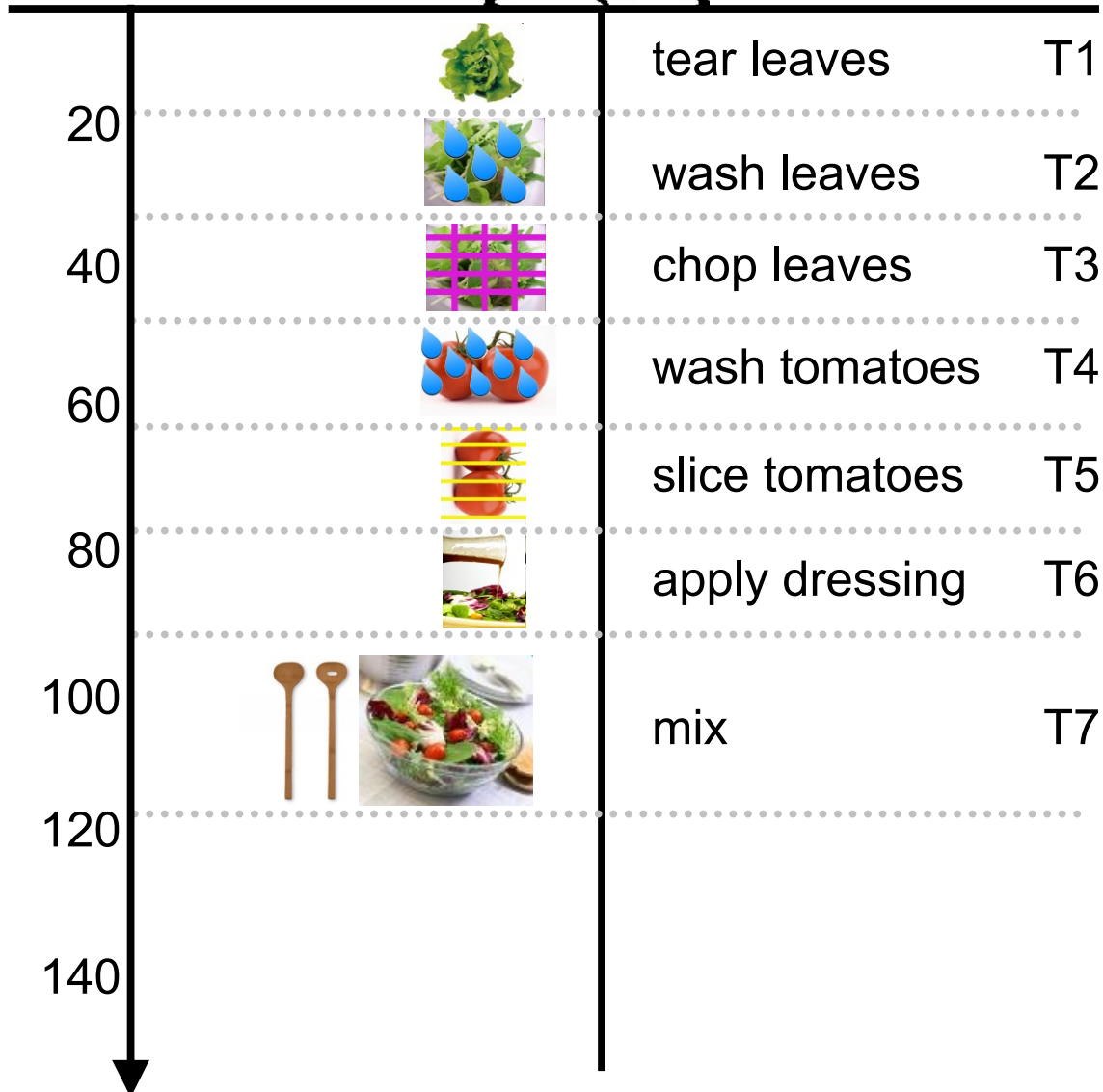
- Preparing a salad consists of 7 tasks (T1-T7).
- A single chef prepares a salad in processing the 7 tasks **sequentially** (one after the other).
- It takes a single chef 142 seconds to prepare a salad.
- We can speed up the process by making the chef work **faster**.



execution time
(seconds)



Fast Single Chef Salad (sequential)



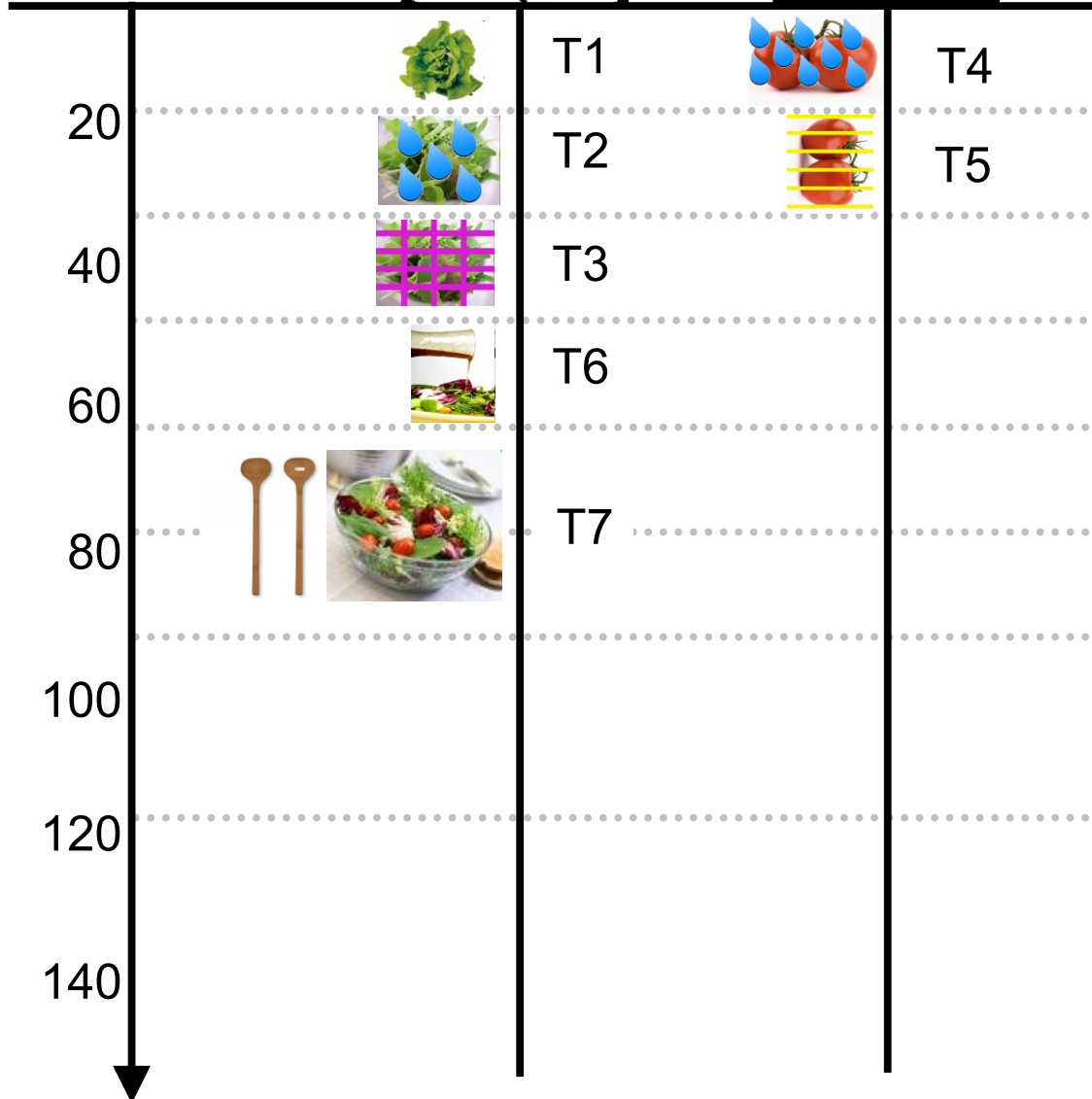
- If we can convince the Chef to finish every task in 18 instead of 20 seconds, we will reduce the process to $7 \times 18 = 126$ seconds.
- The chef agrees for Tasks T1-T6.
- However, he cannot speed up Task T7 without spoiling the entire kitchen.
- He refuses to work any faster than that.
- We can speed up the process by bringing in an additional Chef that works **in parallel** to Chef 1.



execution time
(seconds)

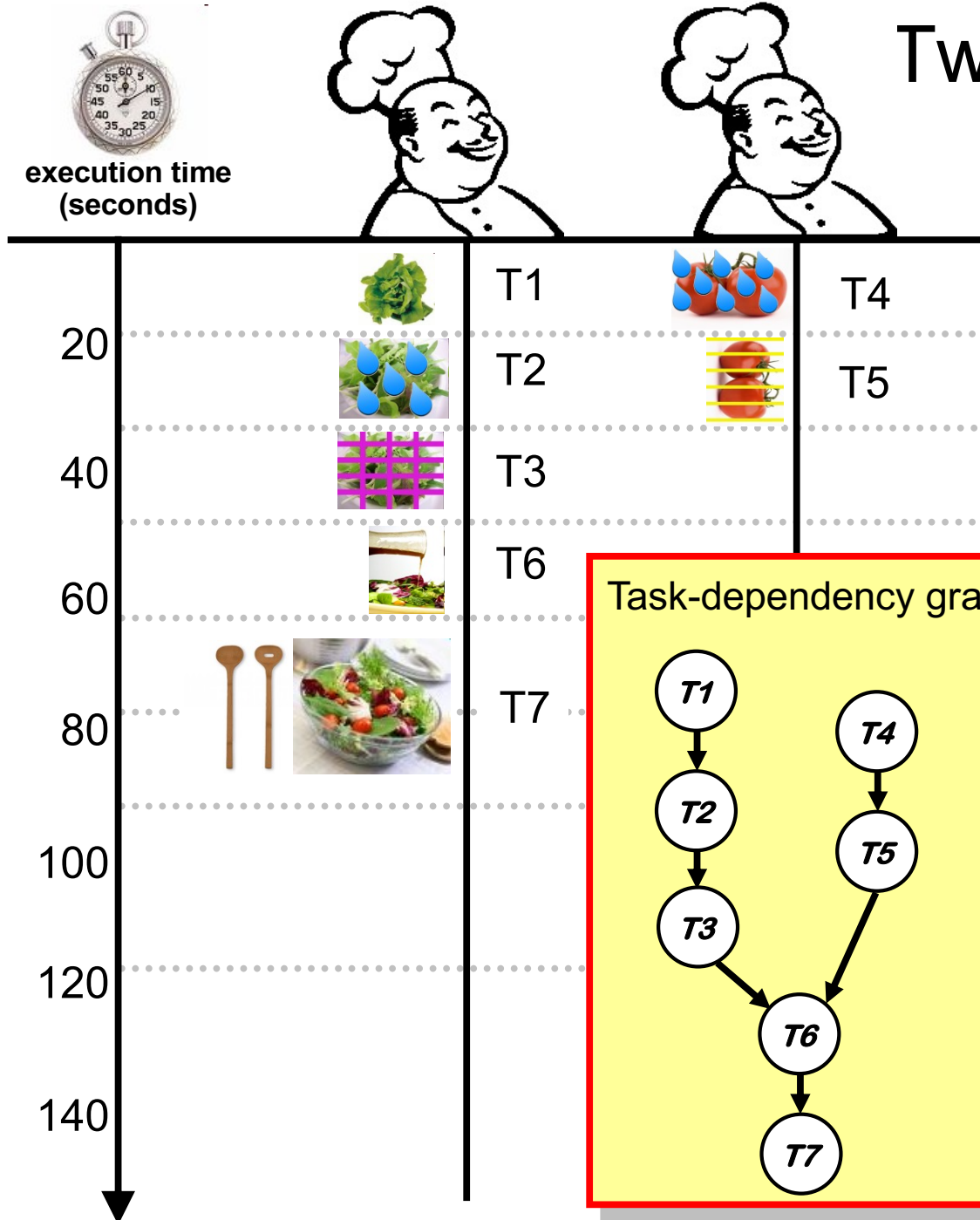


Two Chefs in Parallel Salad

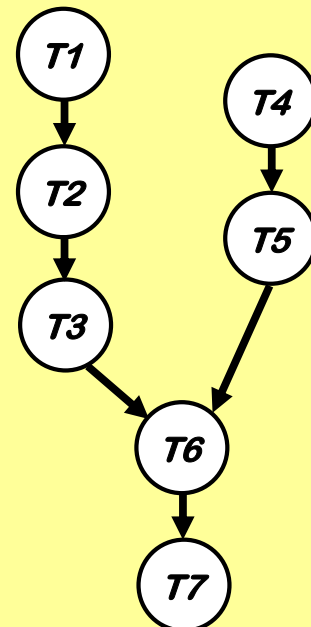


- Chef 2 washes the tomatoes (Task T4) while Chef 1 tears the lettuce (Task T1).
- Task T1 is processed in parallel to Task T4. Processing tasks in parallel is called **task-parallelism**.
- Another form of task-parallelism occurs between tasks T2 and T5.
- There exist **dependencies** between tasks:
 - Vegetables must be washed *before* they are chopped/sliced.
 - Mixing (T7) can only be done *after* the dressing has been applied (T6).

Two Chefs in Parallel Salad

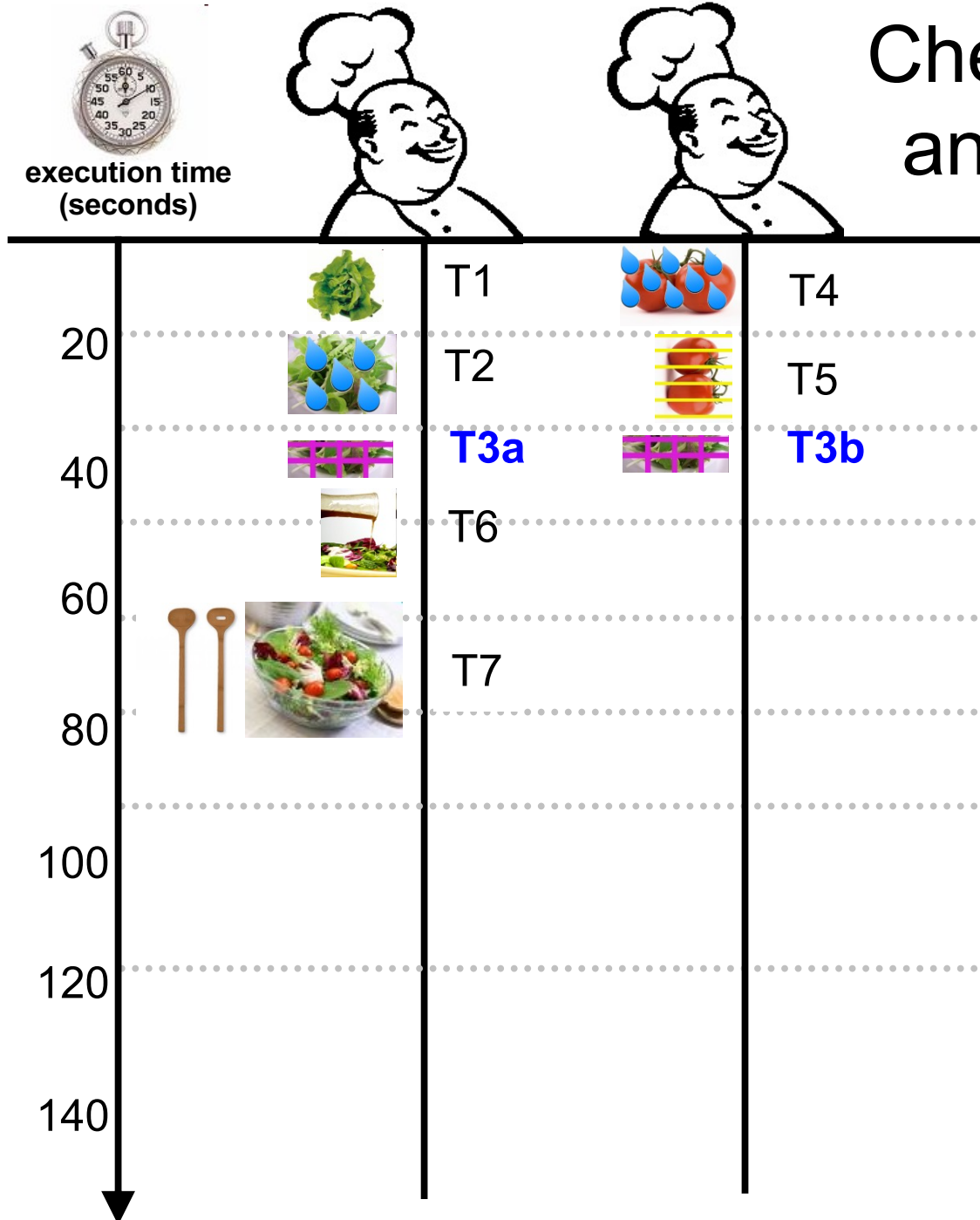


Task-dependency graph

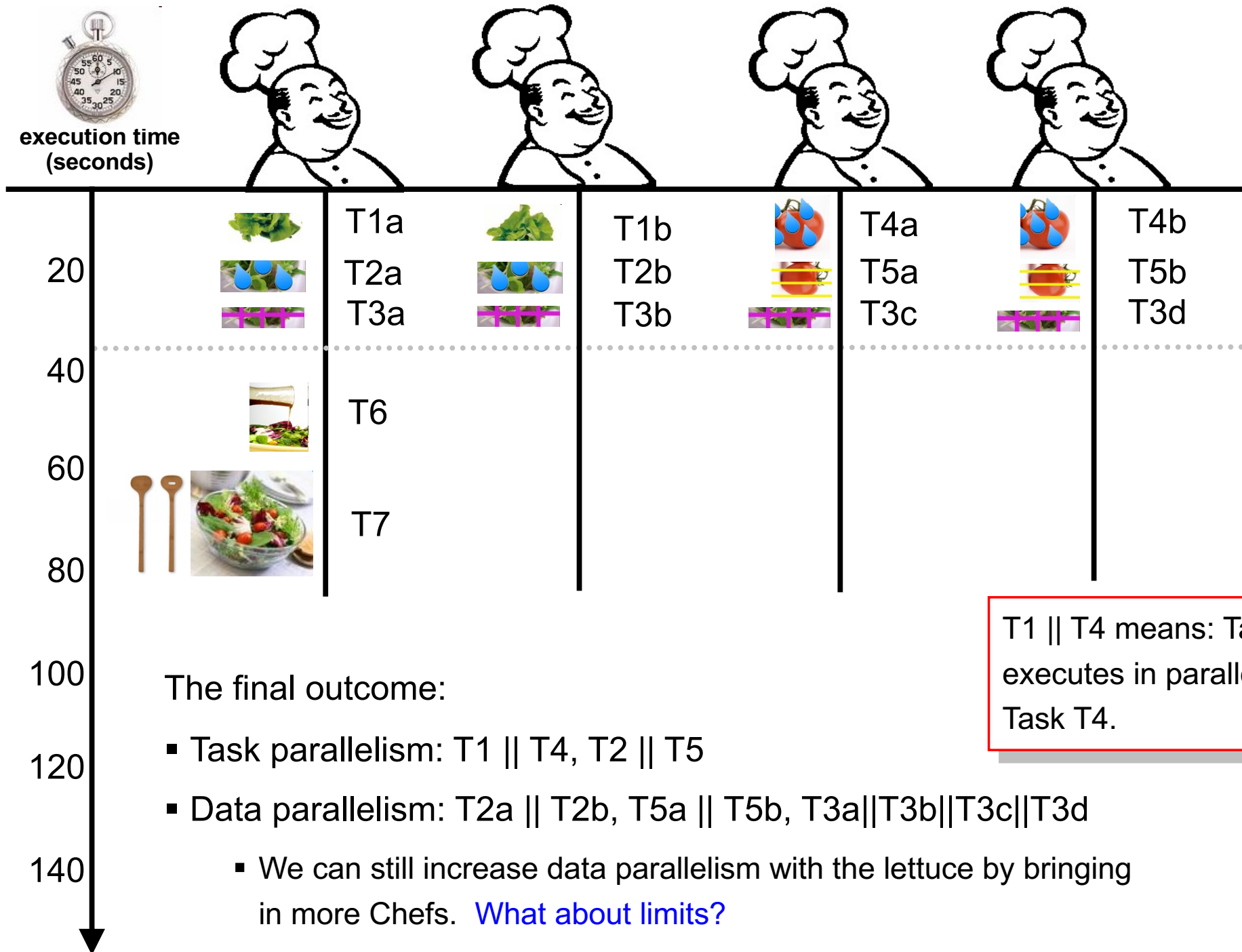


- Dependencies between tasks do not allow further task-parallelism with this example.
- A **task-dependency graph** shows dependencies between tasks.
 - directed acyclic graph (DAG)
 - graph nodes represent tasks
 - directed edge $T_a \rightarrow T_b$ indicates that T_b can only be processed after T_a has completed.
 - See example graph to the left.

Chefs in Parallel (task and data parallelism)



- Dependencies between tasks do not allow further task-parallelism with this example.
- However, we can have several Chefs work in parallel on the “data” of a task.
- This form of parallelism is called **data-parallelism**.
 - Example1: Chef 1 and Chef 2 chop one half of the lettuce each (Tasks **T3a** and **T3b**).
 - Example2: On the next slide more data-parallelism is introduced.



Definitions

Task: a computation that consists of a sequence of instructions. The computation is a *distinct* part of a program or algorithm. (That is, programs and algorithms can be de-composed into tasks).

Examples: “washing lettuce”, “initialize data-structures”, “sort array”, ...

Task-parallelism: parallelism achieved from executing different tasks at the same time (i.e., in parallel).

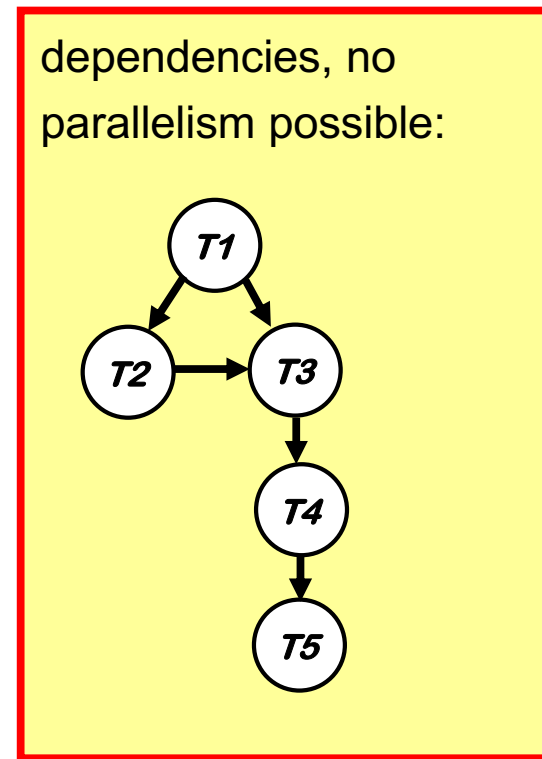
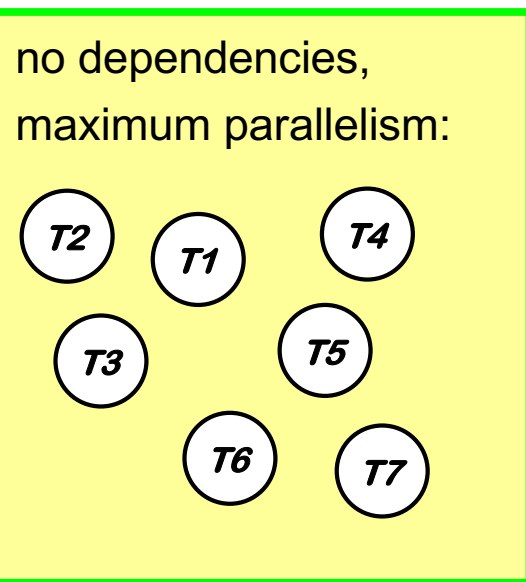
Data-parallelism: performing the same task to different data-items at the same time.

Example: 2 Chefs slicing 1 tomato each. (tomato = data, slicing ~ task).

Definitions (cont.)

Dependencies: an execution order between two tasks T_a and T_b .
 T_a must complete before T_b can execute. Notation: $T_a \rightarrow T_b$.
Dependencies limit the amount of parallelism in an application.

Example task dependency graphs:

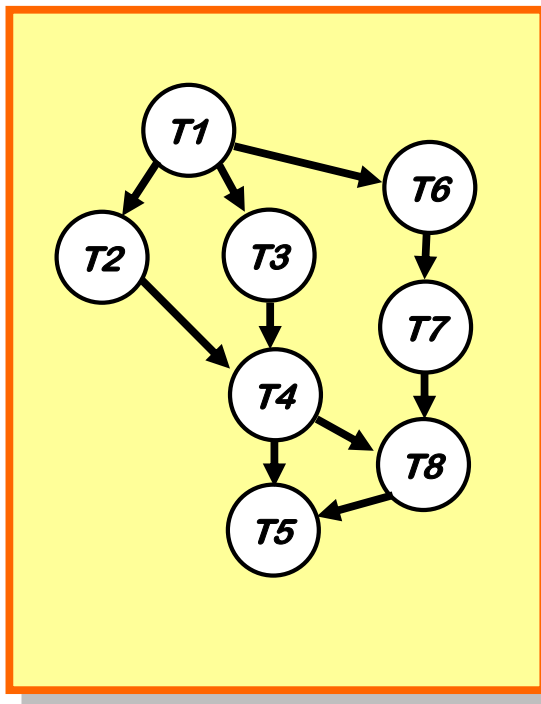


Definitions (cont. cont.)

Dependencies impose a **partial ordering** on the tasks:

Two tasks T_a and T_b can execute in parallel iff

- 1) there is no path in the dependence graph from T_a to T_b
- 2) there is no path in the dependence graph from T_b to T_a

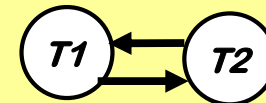


Dependency is transitive:

$T_a \rightarrow T_b$ and $T_b \rightarrow T_c$ implies $T_a \rightarrow T_c$

Say: T_a has to complete before T_b , and T_b has to complete before T_c , therefore T_a has to complete before T_c .

What about this?



Who will do the actual parallelization ?

- The compiler?
 - Would be nice. Programmers could continue writing high-level language programs
 - The compiler would find a task-decomposition for a given multicore processor.
 - Unfortunately this approach does not work (yet).
 - Esp. heterogeneous multiprocessors are difficult to program
 - The speed-up gained from automatic parallelization is limited.
 - Parallelism from automatic parallelization is called **implicit parallelism**.
- The programmer?
 - Yes! (contents of this course)
 - Needs to understand the program to find a task-decomposition.
 - Needs to understand the hardware to achieve a task-decomposition that fits the underlying hardware.
 - Needs to take care of communication & coordination among tasks.
 - Parallelism done by the programmer (her/him)self is called **explicit parallelism**.
 - The research community is working on programming languages and tools that ease this task.

Outline

- Forms of parallelism ✓
 - Task-parallelism ✓
 - Data-parallelism ✓
- Examples

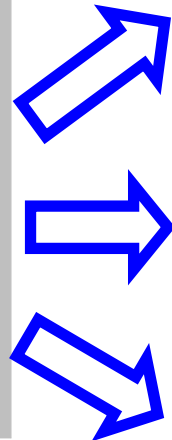
Task Parallelism Example

Example: Assume we have a large data-set in an array and the task is to compute the minimum, average and maximum value. This task can be decomposed into 3 tasks: computing minimum (T1), average (T2), and maximum value (T3).

```
#define maxN 1000000000

int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];

for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
    avrg = avrg + m[i];
    if(m[i] > max)
        max = m[i];
}
avrg = avrg / maxN;
```



```
#define maxN 1000000000
int m[maxN];

int i; int min = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
}

int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
    avrg = avrg + m[j];
}
avrg = avrg / maxN;

int k; int max = m[0];
for(k=1; k < maxN; k++) {
    if(m[k] > max)
        max = m[k];
}
```

T1

T2

T3

Dependence graph:



Note: if T1 – T3 should execute in parallel, then each task needs its own loop index variable (i, j, k)!

Task Parallelism Example (cont.)

```
#define maxN 1000000000  
int m[maxN];
```

```
int i; int min = m[0];  
for(i=1; i < maxN; i++) {  
    if(m[i] < min)  
        min = m[i];  
}
```

T1

```
int j;  
double avrg = m[0];  
for(j=1; j < maxN; j++) {  
    avrg = avrg + m[j];  
}  
avrg = avrg / maxN;
```

T2

```
int k; int max = m[0];  
for(k=1; k < maxN; k++) {  
    if(m[k] > max)  
        max = m[k];  
}
```

T3

- The problem is now decomposed into three tasks T1, T2, T3.
- However: still sequential (T1, then T2 then T3).
- Need a way to tell the compiler that tasks T1, T2 and T3 shall be executed in parallel.

Outline

- Forms of parallelism ✓
 - Task-parallelism ✓
 - Data-parallelism ✓
- Examples ✓
- POSIX Threads
 - Introduction
 - Thread creation, termination
 - Passing arguments to threads
 - Thread synchronization upon termination (pthread_join)
 - Thread scheduling
 - Basic Synchronisation with mutex

Introduction

- We need a way to tell the compiler that a sequence of instructions (=Task) shall be executed in parallel.
 - Examples: Tasks **T1** (min), **T2** (max).
- We need to be able to specify communication and synchronization between tasks.
 - Example communication: task **T1** needs to communicate the computed minimum value (*min*) back to task **T3** for output.
 - Example synchronization: task **T3** can only output the *min* and *max* results after they have been received from task **T1** and **T2**.

Example:

parallel *min*, *max* computations
on integer array:

```
#define maxN 1000000000  
int m[maxN];
```

```
int i; int min = m[0];  
for(i=1; i < maxN; i++) {  
    if(m[i] < min)  
        min = m[i];  
}
```

T1

```
int k; int max = m[0];  
for(k=1; k < maxN; k++) {  
    if(m[k] > max)  
        max = m[k];  
}
```

T2

```
printf("min %d max %d\n",  
      min, max);
```

T3

Introduction

- Communication with processes require specific IPC
 - The task is so small, why not just share access to the memory of that process to begin with?
- A thread is a series of instructions that are executed in the memory context of a process
- There has always been one thread per process described in this course

To make a task, we can create a new thread

- **A thread shares** the virtual memory of a process with all other threads
 - Shares heap
 - Shares static
 - Shares code
- **A thread has it's own** thread id, program counter, registers, stack

The POSIX threads API

- **API = Application Programming Interface:** a set of functions, procedures or classes provided by an operating system or library for use by application programs.
- Many companies provide vendor-specific APIs to program threads.
- IEEE provides the 1003.1c-1995 POSIX API standard
 - also referred to as Pthreads (for “POSIX threads”).
 - POSIX: Portable Operating System Interface (+X for “UniX”)
- Most operating systems support Pthreads
- **POSIX threads allow us to create tasks**
 - The sequence of instructions of a task becomes a POSIX thread.
 - POSIX threads can communicate and synchronize with each other.
 - The operating system schedules threads (i.e., lets them execute). Works both on uniprocessors and multicores.
- Studying Pthreads will make it easy for you to work with other types of threads.



Apache



MySQL

Our first POSIX Thread

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

pthread_t my_thread;

void* threadF (void *arg) {
    printf("Hello from our first POSIX thread.\n");
    sleep(60);
    return 0;
}

int main(void) {

    pthread_create(&my_thread, NULL, threadF, NULL);

    printf("Hello from the main thread.\n");

    pthread_join(my_thread, NULL);

    return 0;
}
```

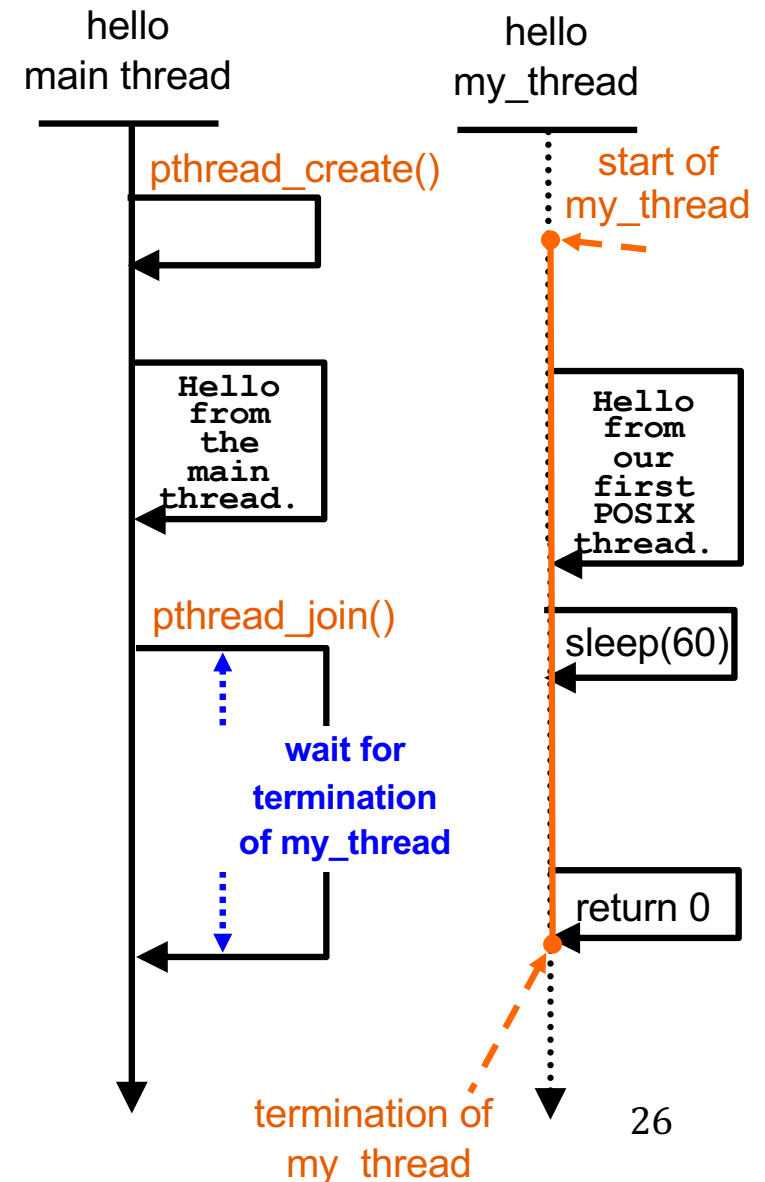
```
[bburg@elc1 ~]$ gcc -o hello hello.c -lpthread
```

- To compile this program, you need to specify **-lpthread**
 - Tells GCC to link in the POSIX thread library.
- `ps -eLf | grep <yourID>` shows 2 threads for hello:
 - the main thread
 - function main()
 - my_thread
 - function threadF()

bburg	11901	11900	11901	0	1	Sep15	pts/6	00:00:00	-bash
bburg	18702	11834	18702	0	2	09:08	pts/4	00:00:00	./hello
bburg	18702	11834	18703	0	2	09:08	pts/4	00:00:00	./hello
bburg	18704	11901	18704	0	1	09:08	pts/6	00:00:00	ps -eLf
bburg	18705	11901	18705	0	1	09:08	pts/6	00:00:00	grep bburg

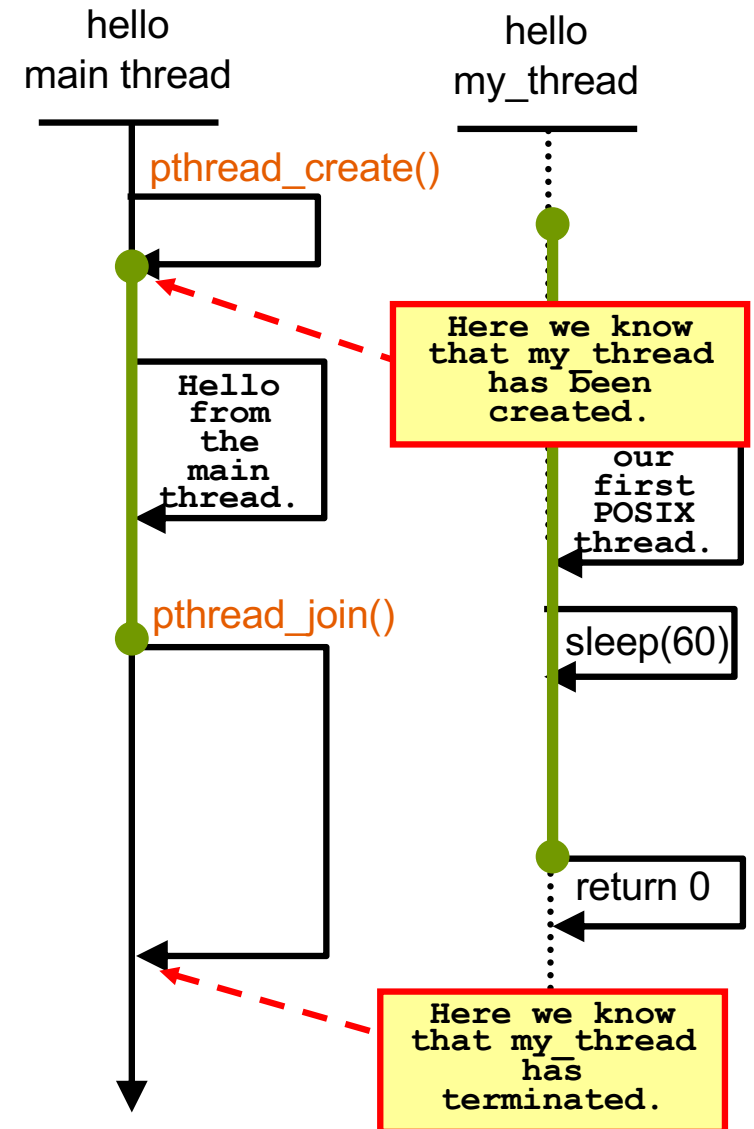
Our first POSIX Thread (cont.)

- The main thread of program hello calls `pthread_create()` to create the POSIX thread 'my_thread'.
- `threadF()` is the pointer to the function that this thread will execute.
- After its creation, the new thread starts executing its thread function (`threadF`).
- `my_thread` and the main thread execute **in parallel**.
- The main thread calls `pthread_join()` to wait for `my_thread` to terminate.
 - The call to `pthread_join()` returns after `my_thread` has terminated.
 - We say that the main thread is **blocked** (has to wait).




Execution Indeterminism

For threads that execute in parallel, no assumption about the statement execution order *between threads* is possible!



pthread_create()

```
int pthread_create(           // Purpose: create a new thread.
    pthread_t *tid,          // pointer to thread ID
    const pthread_attr_t *attr, // pointer to thread attributes
    void * (*start_routine) (void *), // pointer to function that the thread will execute
    void *arg                 // pointer to argument
);
```



run-time argument passing

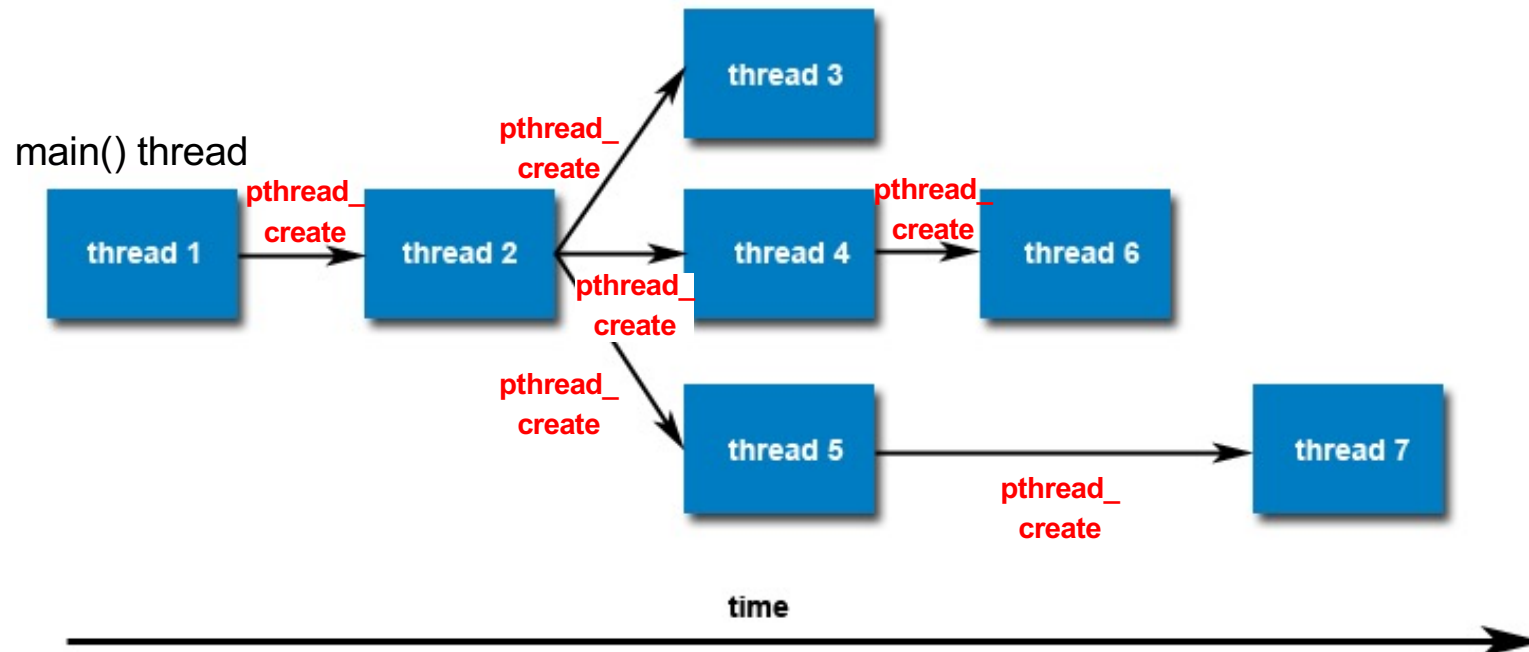
Arguments:

1. The thread ID variable that will be used for the created thread.
 - (We need a way to refer to the thread, e.g., when we want to join it).
2. The thread's attributes (explained on subsequent slides).
 - NULL specifies default attributes.
3. The function that the thread will execute once it is created (start_routine).
 - `void * (*start_routine) (void *)` is a function pointer to a function that
 - has `void *` as its return value
 - accepts one `void *` argument
4. The argument that will be passed to `start_routine()` at **run-time**.

pthread_create()

Once created, threads are peers

- may create other threads
- no implied hierarchy or dependency between threads
- No assumption on execution order of siblings possible.
 - Example: assume thread2 creates thread3 and then thread4.
 - You must not assume that thread3 will start execution before thread 4.
 - Except when applying Pthread scheduling mechanisms



Passing Arguments to a Thread Function

- `pthread_create()` allows us to pass *one* argument to a thread.

```
1  #include <stdio.h>
2  #include <pthread.h>

3  #define NUMT ...
4  pthread_t threadIDs[NUMT]; /* thread IDs */

5  void * tfunc (void * p) {
6      // thread work function
7  }

8  int main(void) {
9      int i;
10     for (i=0; i < NUMT; i++) {
11         pthread_create(&threadIDs[i], NULL,
12                       tfunc, ??? );
13     }
14     for (i=0; i < NUMT; i++) {
15         pthread_join(threadIDs[i], NULL);
16     }
17     return 0;
18 }
```

- Inside the loop several threads are created.
- The same thread function (tfunc) is used for each thread.
 - Common scenario with data parallelism.
- Thread argument passing is used to differentiate between threads.
- Example:
 - tell each thread on which part of a data array it shall work.
 - Tell each cook which tomato to slice.

Passing Arguments to a Thread Function (cont.)

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  int args[NUMT];           /* argument array */
5  pthread_t threadIDs[NUMT]; /* thread IDs */

6  void * tfunc (void * p) {
7      printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         args[i] = i; /* init arg for thread #i */
13         pthread_create(&threadIDs[i], NULL,
14             tfunc, (void *) &args[i] );
15     }
16     for (i=0; i < NUMT; i++) {
17         pthread_join(threadIDs[i], NULL);
18     }
19     return 0;
20 }
```

- In Line 4 we declare an array of integer variables. Each thread has its own “private” array element as argument.



- Each thread will receive a pointer to its “private” array element as argument.
 - In Line 14, `&args[i]` determines the address of the *i*th integer variable.
 - Taking the address of an integer variable gives us an integer pointer
→ type-cast to a void *
 - In Line 7, the thread function casts back the void * to an `int *`.
 - Dereference to get the actual argument value.

What will not work...

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  pthread_t threadIDs[NUMT];      /* thread IDs */

5  void * tfunc (void * p) {
6      sleep(10);
7      printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         pthread_create(&threadIDs[i], NULL,
13                        tfunc, (void *) &i );
14     }
15     for (i=0; i < NUMT; i++) {
16         pthread_join(threadIDs[i], NULL);
17     }
18     return 0;
19 }
```

- In Line 12, a pointer to the loop index variable is passed as argument to each thread.
- Instead of setting up a unique parameter for each thread, the loop index variable is the shared argument between all threads.
- Leads to disaster quickly...
 - The threads may not access the parameter fast enough to read the value “assigned” to them.
 - See, e.g., sleep (10)...

What will not work... (cont.)

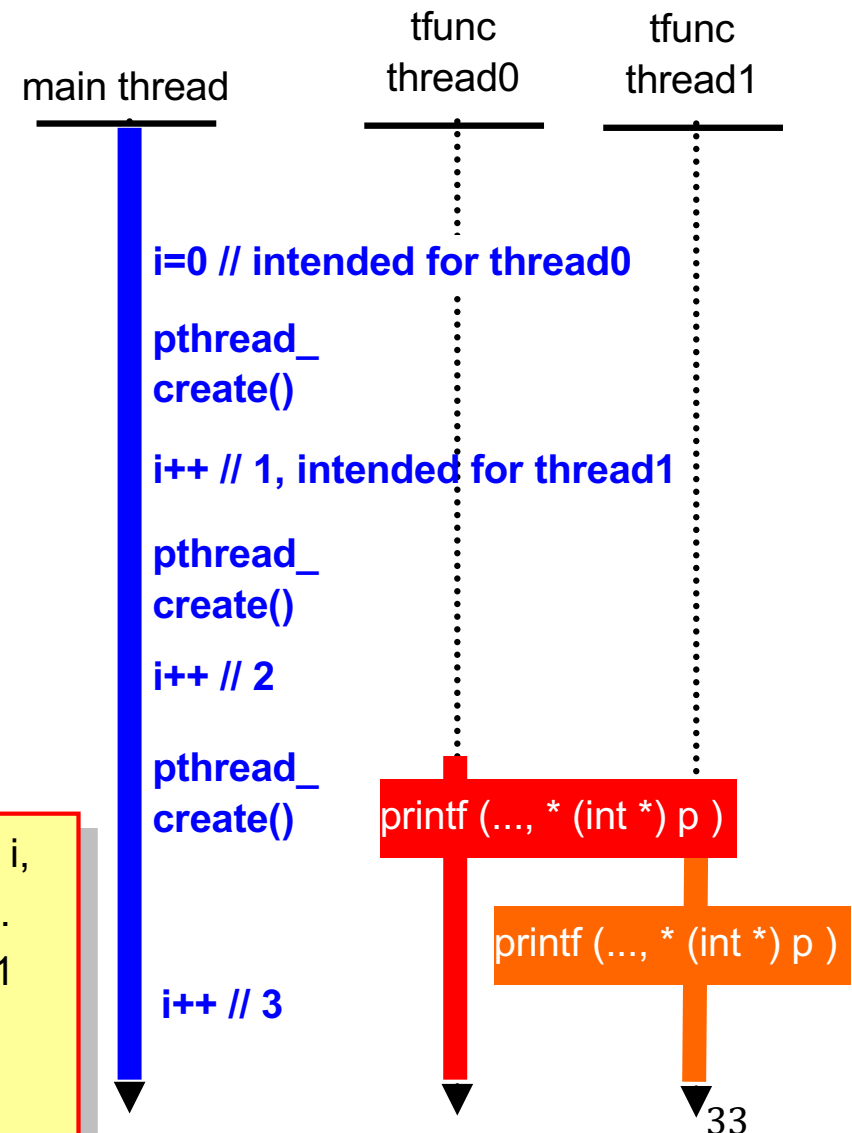
```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  pthread_t threadIDs[NUMT];      /* thread IDs */

5  void * tfunc (void * p) {
6      sleep(10);
7      printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         pthread_create(&threadIDs[i], NULL,
13                       tfunc, (void *) &i );
14     }
15     for (i=0; i < NUMT; i++) {
16         pthread_join(threadIDs[i], NULL);
17     }
18     return 0;
19 }
```

By the time thread0 dereferences the pointer to variable i, the value has already been changed by the main thread. Thread0 reads the wrong value (2 instead of 0). Thread1 also reads 2 → two chefs start chopping tomato nr. 2. Ouch!



Our first Race-condition

- The correctness of the previous program depends on how fast the created threads manage to read variable *i* (reading must happen before the main thread updates *i*, otherwise a wrong value is read).
 - (Assume there was no `sleep(10)` statement).
 - Execution speed among threads is determined by the operating system that schedules the threads.
 - Depends on system load also.

Race Condition: *“a situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution”.*

From the previous example:

- shared data item: variable *i*
- final result: what value thread 0 reads as its parameter (0 or >0).
- relative timing: when and how fast thread 0 and the main thread execute relative to each other.

Passing more than one argument to a thread

```
1  #include<stdio.h>
2  #include<pthread.h>

3  pthread_t threadID;

4  struct multi_arg {
5      int range;
6      int tomato;
7  };

8  void * tfunc (void * p) {
9      struct multi_arg * ptr = (struct multi_arg *) p;
9      printf("range: %d, tomato %d\n",
10             ptr->range, ptr->tomato);
11 }

12 int main(void) {

13     struct multi_arg ma;
14     ma.range = 1;
15     ma.tomato = 2;

16     pthread_create(&threadID, NULL, &tfunc,
17 (void *) &ma );
18     pthread_join(threadID, NULL);
19     return 0;
20 }
```

To pass more than one argument to a thread...

- Create a struct that can hold multiple arguments (Lines 4-7).
- Create a variable of that struct type (lines 13-15).
- Pass a pointer to this struct variable to pthread_create (Line 17).
- In the thread function, cast the void * back to a pointer to the struct (Line 9).

pthread_join()

```
int pthread_join(           // Purpose: wait for a thread to terminate.
    pthread_t tid,         // thread ID we are waiting for
    void ** status,        // exit status
);
```

Note: with pthread_create() we are passing a pointer to a thread ID as argument. With pthread_join() it is the thread ID itself!

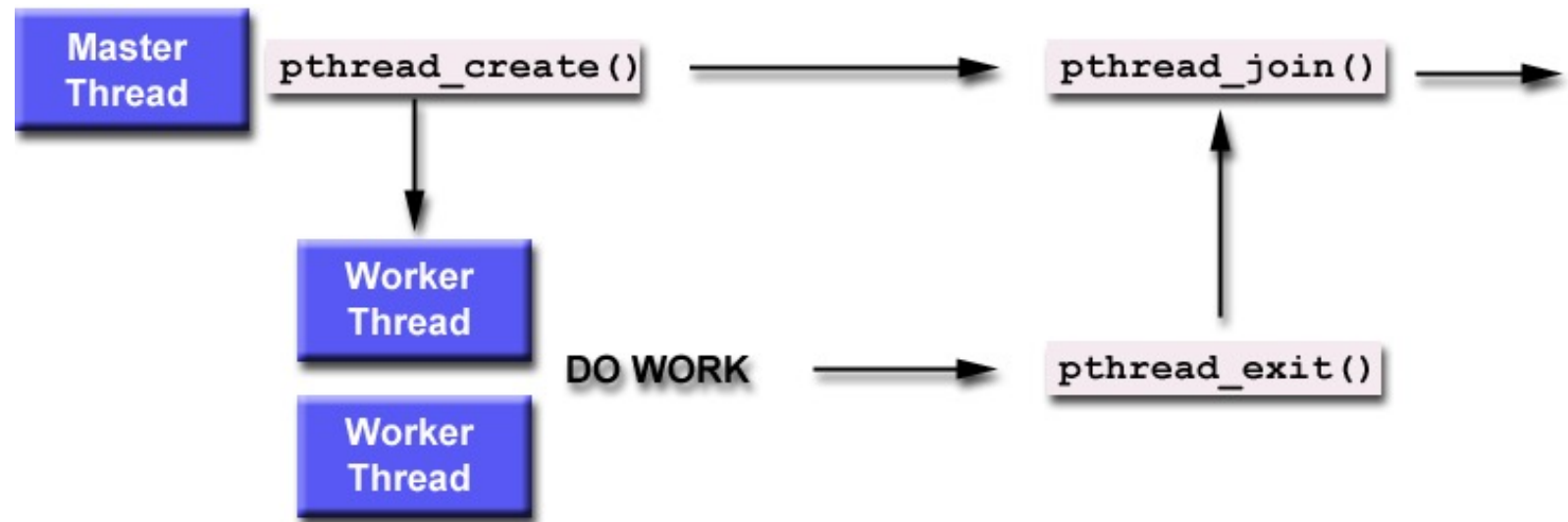
Arguments:

1. The thread ID of the thread we are waiting for.
2. The completion status of the exiting thread will be copied into *status, unless status is NULL, in which case the completion status is ignored.

Note:

- Once a thread is joined, the thread no longer exists. Its thread ID is no longer valid, and it cannot be joined again, e.g. with another thread!

pthread_join() (cont.)



- Joining threads is a mechanism to accomplish synchronization between threads.
- `pthread_join` blocks the calling thread until the specified thread terminates.
- A thread can only join one `pthread_join` call. It is a logical error to attempt to join a thread several times, e.g., from different threads.
- Further synchronization methods will be discussed later
 - mutexes, condition variables

Thread termination

There are three ways to terminate a thread:

- 1) a thread can return from his start routine
- 2) a thread can call `pthread_exit()`
- 3) a thread can be cancelled by another thread.

In each case, the thread is destroyed and his resources become unavailable.

Gotcha: terminating a thread does not release

- dynamic memory allocated by the thread (`malloc`),
- close files that have been opened by the thread.

Cleaning up those resources is the responsibility of the programmer!

Thread termination (cont.)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%d!\n", tid);
    pthread_exit(NULL);
}

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

Distinction between pthread_exit() and exit():

- `pthread_exit()` is used to explicitly exit a thread.
 - Example: called after a thread has completed its work and is no longer needed.
- `exit()` terminates the **whole** program, including all threads.
 - Example: used when a program encounters a severe error condition which makes it impossible to continue.
- Both can be called from the main thread or any Pthread.

Thread termination (cont.)

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void * Hello(void *)
5  {
6      sleep(20);
7      printf("Hello World!\n");
8      pthread_exit(NULL);
9  }
10
11 int main (int argc, char *argv[])
12 {
13     pthread_t thread;
14
15     pthread_create(&thread, NULL,
16                   Hello, NULL);
17
18     pthread_exit(NULL);
19 }
```

- When **main()** terminates, all threads are terminated.
 - This is equivalent to calling `exit()` at the end of **main()**.
 - Example: the Hello thread created in Line 15 won't be able to print its message, because right after thread creation, main terminates...
- Calling `pthread_exit()` in main (Line 18) will allow the other threads to continue.
 - Example: now the Hello thread can sleep for 20 seconds, print its message, and terminate. At this point, the whole program will terminate.

Thread termination (cont.)

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void * Hello(void *)
5  {
6      sleep(20);
7      printf("Hello World!\n");
8      pthread_exit( (void *) 22 );
9  }
10
11 int main (int argc, char *argv[])
12 {
13     void * rPtr;
14     pthread_t thread;
15
16     pthread_create(&threads[t], NULL,
17                   Hello, NULL);
18
19     pthread_join ( thread, &rPtr );
20
21     printf("Thread returned %d\n",
22           (long) rPtr);
23
24 }
```

- `pthread_exit()` can be used to pass an exit status to any thread that will join the exiting thread.
- The value of the exit status must be **void ***.
 - We can cast any value to void *.
 - See example, Line 8.
- `pthread_join()` requires a void ** as argument for returning the void * value of the exit status.
 - Void ** is a void * passed by-reference!
- Example:
 - In Line 13 we declare a void * rPtr.
 - Passing rPtr by reference means passing a pointer to rPtr (&rPtr), Line 19.
 - The void * value in rPtr must then be cast back to its original type (long in this example).

Synchronising with Mutex

- Change from parallel execution to serial execution of instructions

- Consider a section of code

- Allow at most one thread to execute the section of code
- Other threads are blocked from executing the section of code
- Programmer defines the section and controls access using pthread function calls

Instruction1

Instruction2

Begin section

Instruction3

Instruction4

Instruction5

End section

Instruction6

Instruction7

...

Synchronising with Mutex

- *Change from parallel execution to serial execution of instructions*

- Example:

- N threads executes instruction 1-7 in sequence
- Mutex -> Only 1 thread can execute 3,4,5 at any given time

Instruction1

Instruction2

Begin section

Instruction3

Instruction4

Instruction5

End section

Instruction6

Instruction7

...

- Why is this helpful?

- Instructions access memory
- hence we can serialise memory access!
- Only one thread can access that specific memory at a time

Synchronise shared address space between threads

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

int tick_tock = 0;

pthread_t my_thread;

void* threadF (void *arg) {
    printf("Thread worker is %d.\n", tick_tock++);
}

int main(void) {
    tick_tock = 1;
    pthread_create(&my_thread, NULL, threadF, NULL);
    printf("main worker is %d.\n", tick_tock++);
    pthread_join(my_thread, NULL);
    printf("tick tock is %d.\n", tick_tock);
    return 0;
}
```

- Threads share the address space.
- Variable i is visible by both threads
 - main thread
 - my_thread
- Both a blessing and a curse
 - (will become apparent soon)

Synchronise shared address space between threads

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

char *message = "Chocolate microscopes?";
int mindex = 0;
size_t message_len = 0;

pthread_t my_thread;

void* threadF(void *arg) {
    while (1) {
        if (mindex < message_len) {
            printf("%c", message[mindex]);
            mindex++;
        } else
            break;
    }
    printf("Thread end at %d.\n", mindex);
    return NULL;
}
```

```
int main(void) {
    message_len = strlen(message);
    pthread_create(&my_thread, NULL, threadF, NULL);

    while (1) {
        if (mindex < message_len) {
            printf("%c", message[mindex]);
            mindex++;
        } else {
            break;
        }
    }
    printf("main end at %d.\n", mindex);
    pthread_join(my_thread, NULL);
    printf("\n");
    printf("all threads ended: %d.\n", mindex);
    return 0;
}
```

Synchronise shared address space between threads

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <pthread.h>
4
5 char *message = "Chocolate microscopes?";
6 int minindex = 0;
7 pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
8 size_t message_len = 0;
9
10 pthread_t my_thread;
11
12 void* threadF(void *arg) {
13     while (1) {
14         pthread_mutex_lock(&lock);
15         if (minindex < message_len) {
16             printf("%c", message[minindex]);
17             minindex++;
18         } else {
19             pthread_mutex_unlock(&lock);
20             break;
21         }
22         pthread_mutex_unlock(&lock);
23     }
24     printf("Thread end at %d.\n", minindex);
25     return NULL;
26 }
```

```
27
28 int main(void) {
29     message_len = strlen(message);
30     pthread_create(&my_thread, NULL,
31                   threadF, NULL);
32
33     while (1) {
34         pthread_mutex_lock(&lock);
35         if (minindex < message_len) {
36             printf("%c", message[minindex]);
37             minindex++;
38         } else {
39             pthread_mutex_unlock(&lock);
40             break;
41         }
42         pthread_mutex_unlock(&lock);
43     }
44     printf("main end at %d.\n", minindex);
45     pthread_join(my_thread, NULL);
46     printf("\n");
47     printf("all threads ended: %d.\n", minindex);
48     return 0;
49 }
```

Outline

- Forms of parallelism ✓
 - Task-parallelism ✓
 - Data-parallelism ✓
- Examples ✓
- POSIX Threads ✓
 - Introduction ✓
 - Thread creation, termination ✓
 - Passing arguments to threads ✓
 - Thread synchronization upon termination (pthread_join) ✓
 - Thread scheduling ✓
 - Basic Synchronisation with mutex ✓