CSE502: Foundations of Parallel Programming

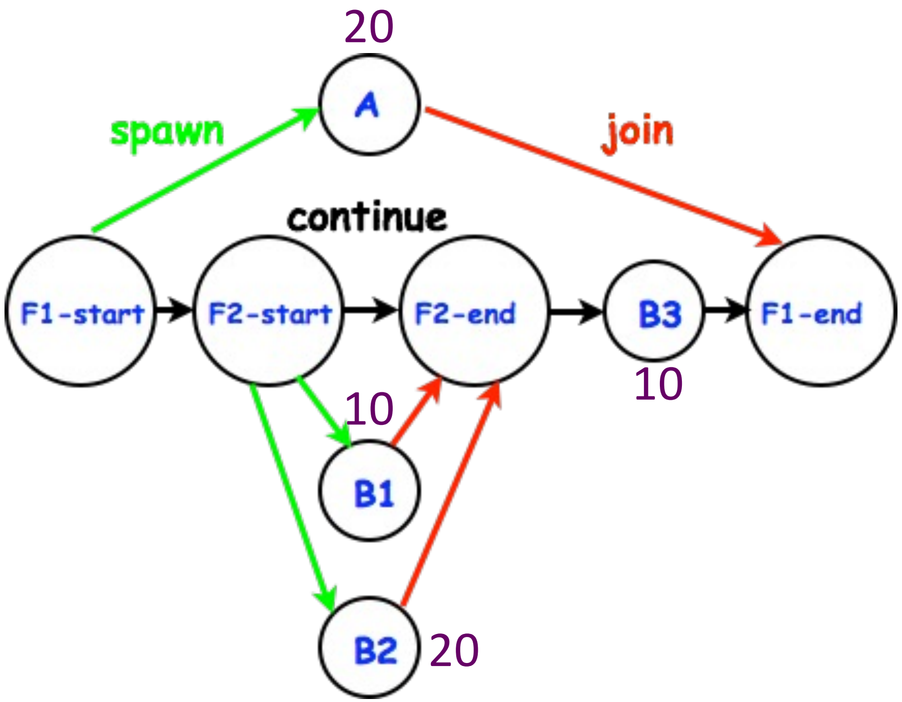
Lecture 07: Greedy Scheduling of Computation Graph on a Fixed Number of Processors

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Class

- Computation graph
  - Ideal parallelism
  - Introduction to data races

```
finish {            // F1
   async A;
   finish {         // F2
      async B1;
      async B2;
   }                // F2
   B3;
}                   // F1
```



Work = 60
CPL = 30
Ideal Parallelism = Work/CPL = 60/30 = 2

A data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1, i.e., S1 and S2 can potentially execute in parallel, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

Today’s Class

- Greedy scheduling of computation graph on fixed number of processors
    - Lower and upper bound on execution time
- Thread pool

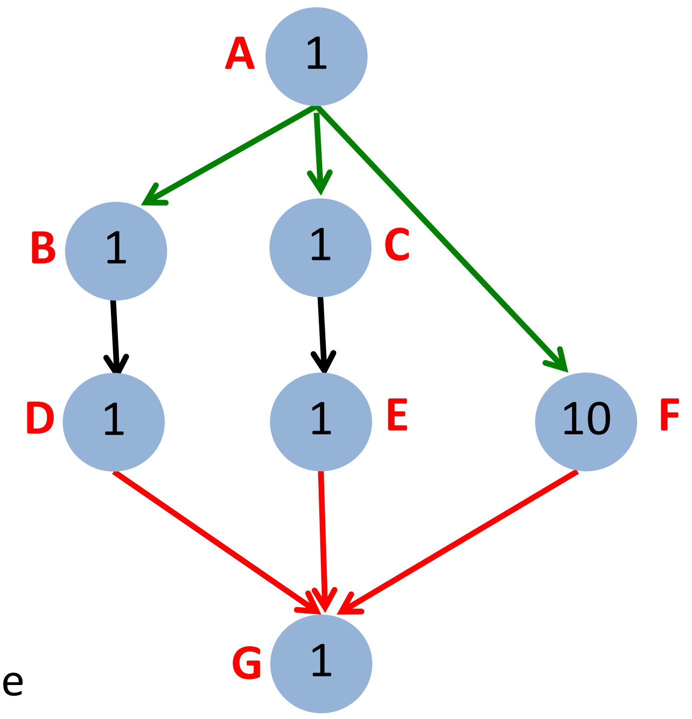
Greedy Schedule

- A greedy schedule is one that never leaves a processor idle when one or more nodes are ready for execution
- A node is **ready** for execution if all its predecessors have been **executed**
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules
- where T_P = execution time of a schedule for computation graph G on P processors

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

Scheduling of a Computation Graph on a fixed number of processors: Example

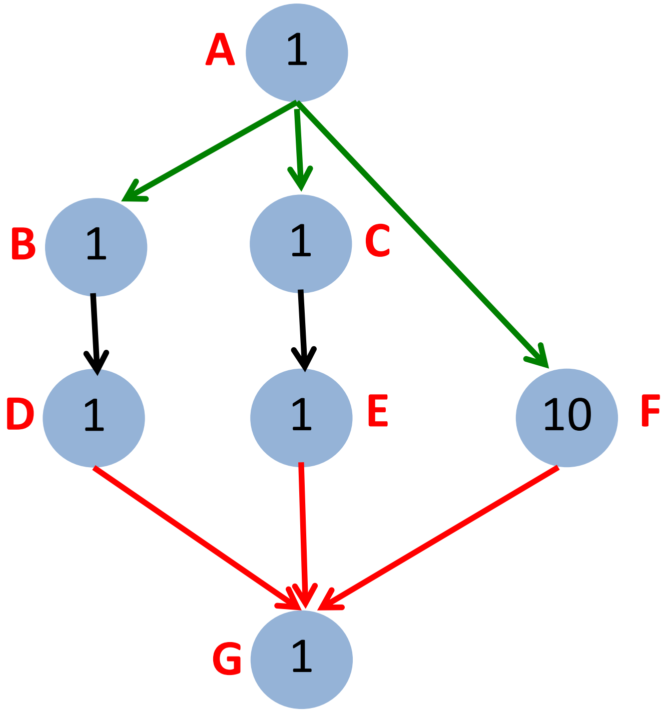
```
A();                  // 1 units
finish {
   async {
      B();         // 1 units
      D();         // 1 units
   }
   async {
      C();         // 1 units
      E();         // 1 units
   }
   async F();   // 10 units
}
G();                 // 1 units
```



- Spawn edge
- Continue edge
- Join edge
- Node label = time(N), for all nodes N in the graph
- CPL (Graph) = 12
- Work (Graph) = 16
- Ideal Parallelism = 16/12 = 1.33

Lower Bounds on Execution Time of Greedy Schedules

- **Best** possible execution time of this computation graph on **two** processors



Start Time	Proc1	Proc2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

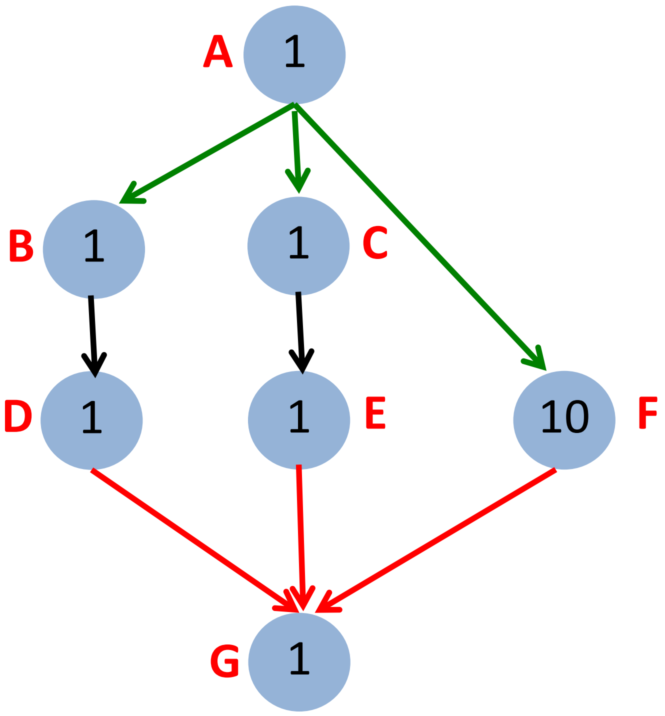
Lower Bounds on Execution Time of Greedy Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - Can be different for different schedules
- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$
- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

Upper Bound on Execution Time of Greedy Schedules

- **Worst** possible execution time of this computation graph on **two** processors



Start Time	Proc1	Proc2
0	A	
1	B	C
2	D	E
3	F	
4	F	
5	F	
6	F	
7	F	
8	F	
9	F	
10	F	
11	F	
12	F	
13	G	
14	Time = 14	

Upper Bound on Execution Time of Greedy Schedules

- Define a time step to be **complete** if ≥ P nodes are ready at that time, or **incomplete** otherwise
  - $\#\text{Steps}_{\text{Complete}} \leq \text{WORK}(G)/P$
  - $\#\text{Steps}_{\text{Incomplete}} \leq \text{CPL}(G)$
  - $T_P = \#\text{Steps}_{\text{Complete}} + \#\text{Steps}_{\text{Incomplete}}$
  - $T_P \leq \text{WORK}(G)/P + \text{CPL}(G)$
    - Theorem [Graham'68, Brent'74]

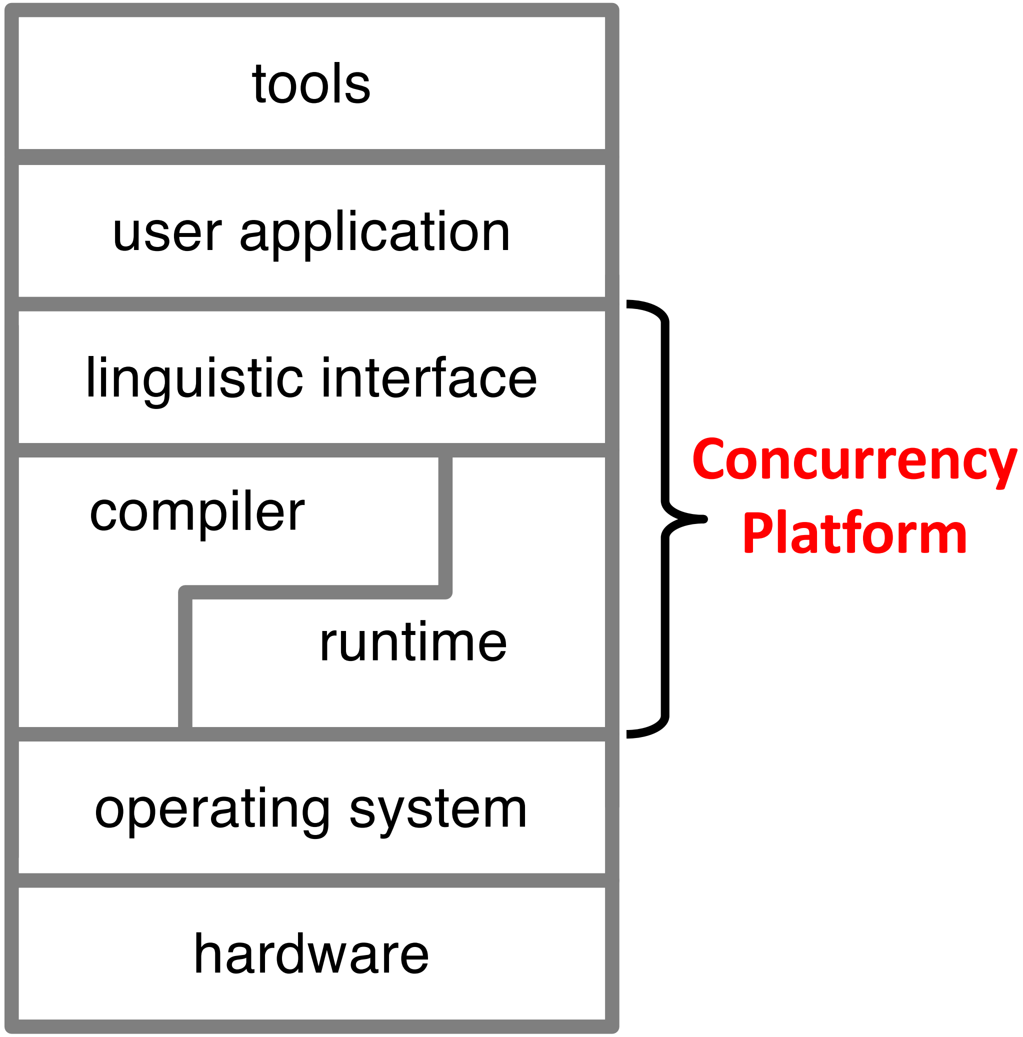
Start Time	Proc1	Proc2
0	A	
1	B	C
2	D	E
3	F	
4	F	
5	F	
6	F	
7	F	
8	F	
9	F	
10	F	
11	F	
12	F	
13	G	
14	Time = 14	

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

Today’s Class

- Greedy scheduling of computation graph on fixed number of processors
  - Lower and upper bound on execution time
- Thread pool

Concurrency Platforms (Recap Lec04)



tools

user application

linguistic interface

compiler

runtime

operating system

hardware

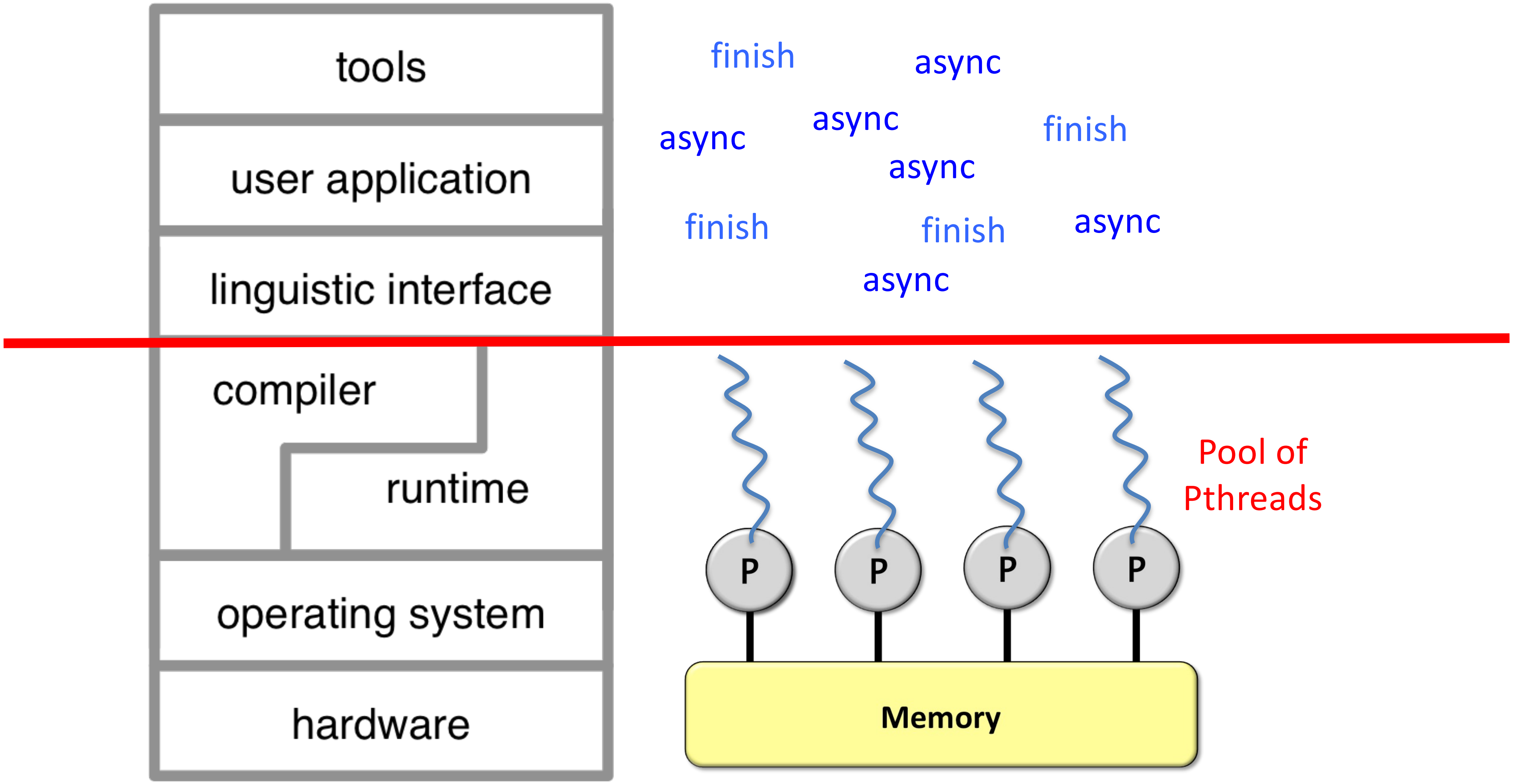
Concurrency Platform

A concurrency platform should provide:

- an interface for specifying the ***logical parallelism*** of the computation;
- a runtime layer to automate scheduling and synchronization; and
- guarantees of performance and resource utilization competitive with hand-tuned code.

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec01_intro.pdf

Thread Pool in Concurrency Platforms



In this course we are only going to consider the case where a thread pool has total number of threads (Pthreads) equal to total number of available cores

Thread Pool in Concurrency Platforms

- A Key component in any concurrency platform that relieves the user from the complexity of mapping tasks to threads (e.g., Pthreads) to achieve maximum performance on a given number of processors (or cores)
  - *Sneak peek: thread pools does not restricts to shared-memory platform (multicore processor), but can also be extended to distributed memory platform (supercomputer)*

Mapping the Linguistic Interface to Thread Pool Runtime

- Compiler based runtimes
  - Cilk, X10, TryCatchWS
  - User code translated to runtime code and then compiled using a native compiler (e.g., gcc)
  - Compiler maintenance is a costly affair and it is not so easy to use new features from mainstream languages
  - Using standard debugger (e.g., gdb) is not possible as the line number information inside the symbol table is w.r.t. the compiler generated code and not w.r.t. the user written code
  - However, compiler based approach provide several opportunities for code optimizations and doing smart things
- Library based runtimes Our focus
  - Java fork/join framework, HClib, HJlib
  - Removes all the drawbacks of a compiler based approach

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime

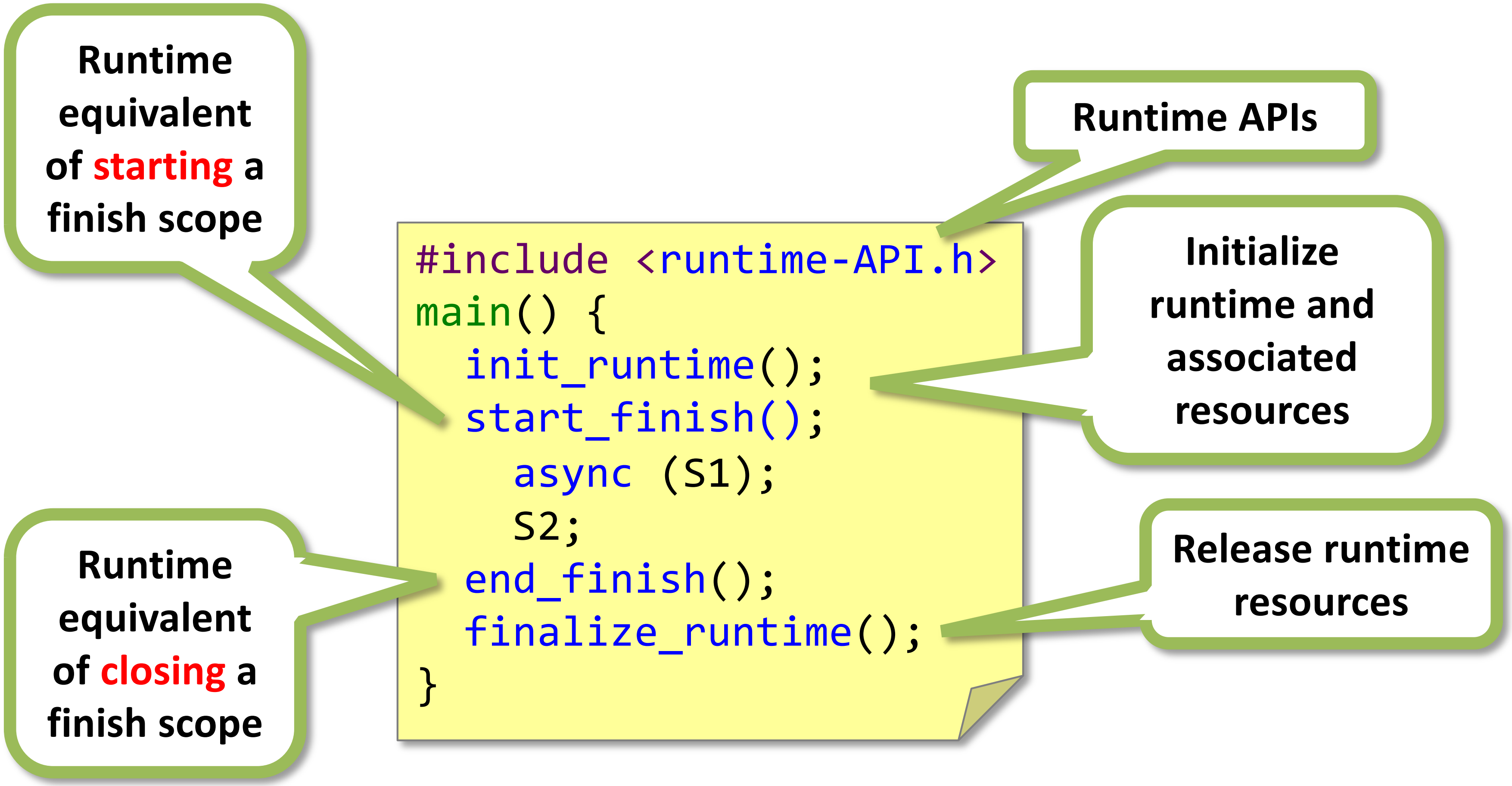
```
#include <runtime-API.h>
main() {
  init_runtime();
  finish {
    async (S1);
    S2;
  }
  finalize_runtime();
}
```

Runtime APIs

Initialize runtime and associated data-structures

Release runtime resources

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime



```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

Runtime equivalent of starting a finish scope

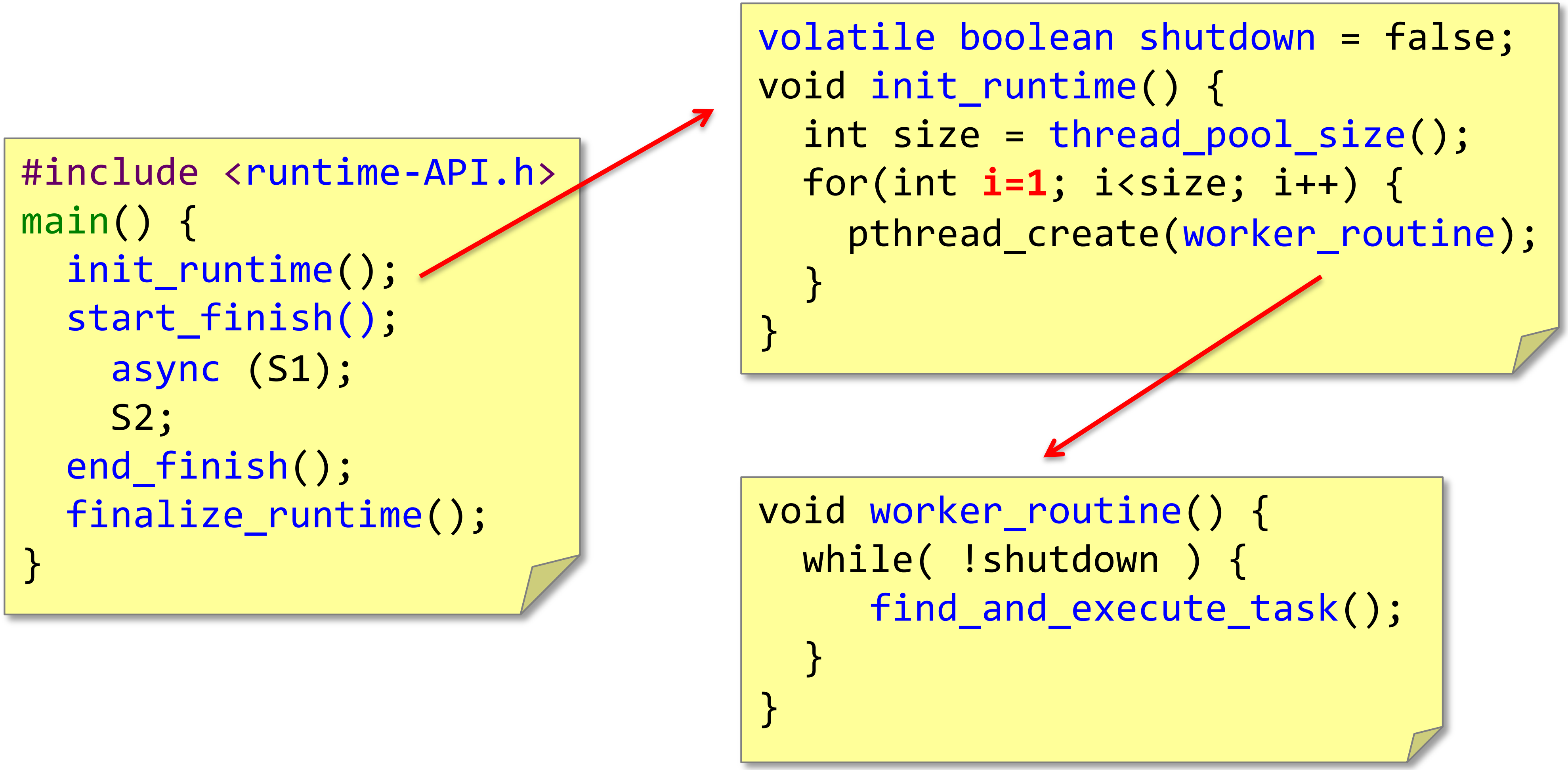
Runtime APIs

Initialize runtime and associated resources

Runtime equivalent of closing a finish scope

Release runtime resources

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime



```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
volatile boolean shutdown = false;
void init_runtime() {
  int size = thread_pool_size();
  for(int i=1; i<size; i++) {
    pthread_create(worker_routine);
  }
}
```

```
void worker_routine() {
  while( !shutdown ) {
     find_and_execute_task();
  }
}
```

Note: here the workers are continuously spinning, but in some implementation they might sleep if no tasks are available

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
volatile int finish_counter = 0;
void start_finish() {
  finish_counter = 0; //reset
}
```

Note: in case of nested finish (e.g., Fibonacci), we need a better way to manage finish scopes. Recall, in Fibonacci every fib(n) call created a new finish, which ultimately creates a tree of finishes

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
void async(task) {
  lock_finish();
  finish_counter++;//concurrent access
  unlock_finish();
  // copy task on heap
  void* p = malloc(task_size);
  memcpy(p, task, task_size);
  //thread-safe push_task_to_runtime
  push_task_to_runtime(&p);
  return;
}
```

Note: Runtime stores pointer to the tasks passed in the async. To ensure valid pointer during task execution, we heap allocate the task and store pointer to the task on heap.

Note: there are better ways to increment finish counter rather than doing it inside locks

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
void end_finish() {
  while(finish_counter != 0) {
    find_and_execute_task();
  }
}
```

```
void find_and_execute_task() {
   //grab_from_runtime is thread-safe
   task = grab_task_from_runtime();
   if(task != NULL) {
      execute_task(task);
      free(task);
      lock_finish();
      finish_counter--;
      unlock_finish();
   }
}
```

Note: there are better ways to decrement finish counter rather than doing it inside locks

Mapping the Linguistic Interface to **Library** Based Thread Pool Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
void finalize_runtime() {
   //all spinning workers
   //will exit worker_routine
   shutdown = true;
   int size = thread_pool_size();
   // master waits for helpers to join
   for(int i=1; i<size; i++) {
     pthread_join(thread[i]);
   }
}
```

How to Store Tasks in Runtime ?

- push_task_to_runtime()
- grab_task_from_runtime()

Data-structures for storing tasks in a thread pool based runtime plays a very important role in determining the scalability and performance of the runtime

Next Lecture

- Work-sharing and work-stealing task scheduling