CSC367 Parallel computing

Lecture 9: Parallel Architectures and Parallel Algorithm Design Continued!

Thanks to Scinet, NSF XSEDE, James Demmel

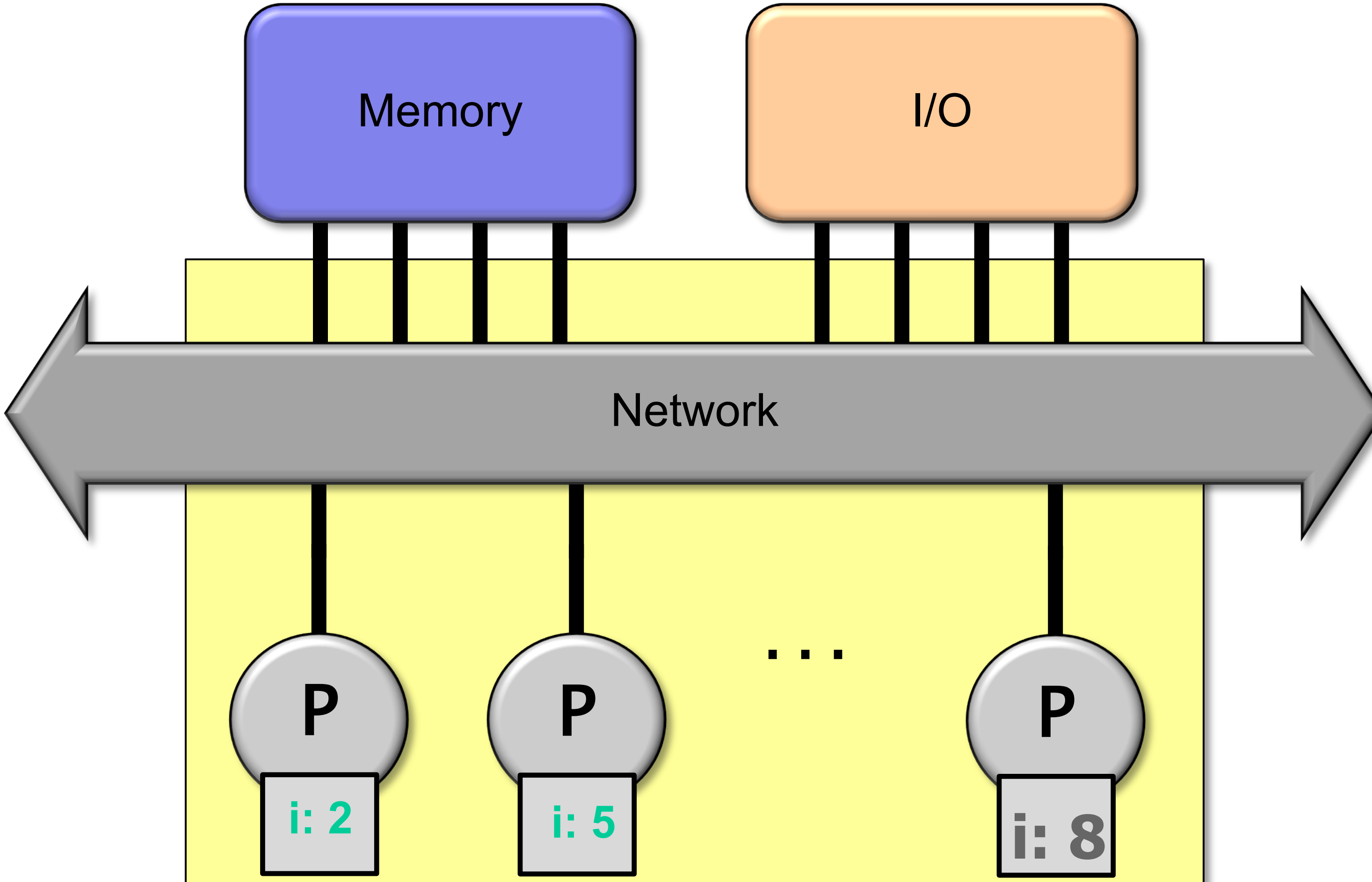
Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Shared Memory Architecture



Chip Multiprocessor (CMP)

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Parallel Programming Models

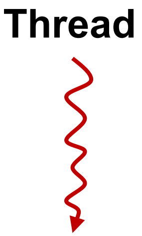
- Programming model is made up of the languages and libraries that create an abstract view of the machine: Pthreads!

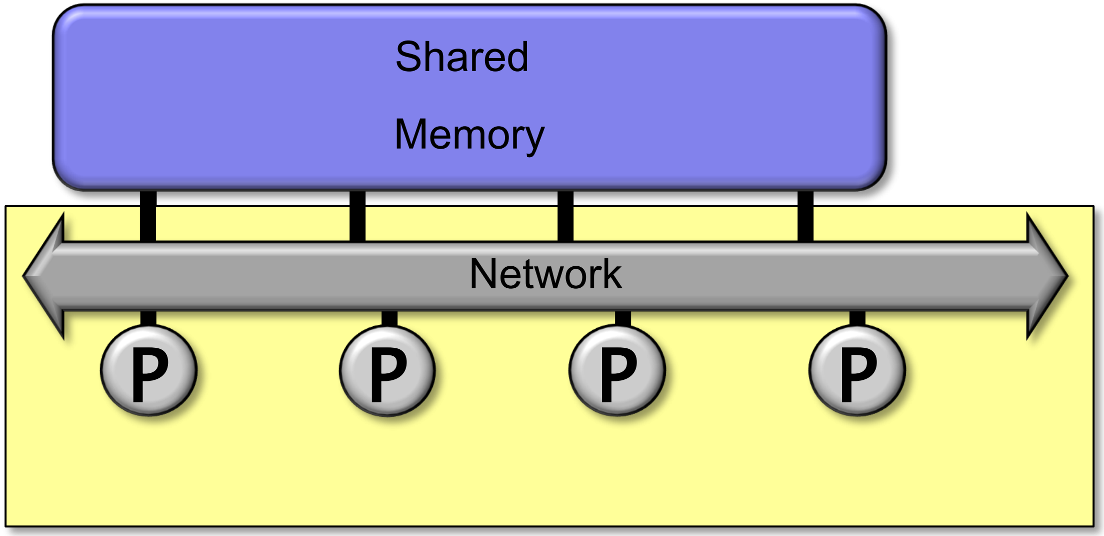
The programming model enables us to identify

- Control
  - How is parallelism created?
  - What orderings exist between operations?
- Data:
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?

Programming Model: Shared Memory

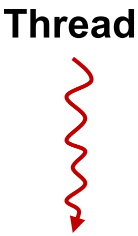
Program is a collection of threads of control, can be created mid-execution.

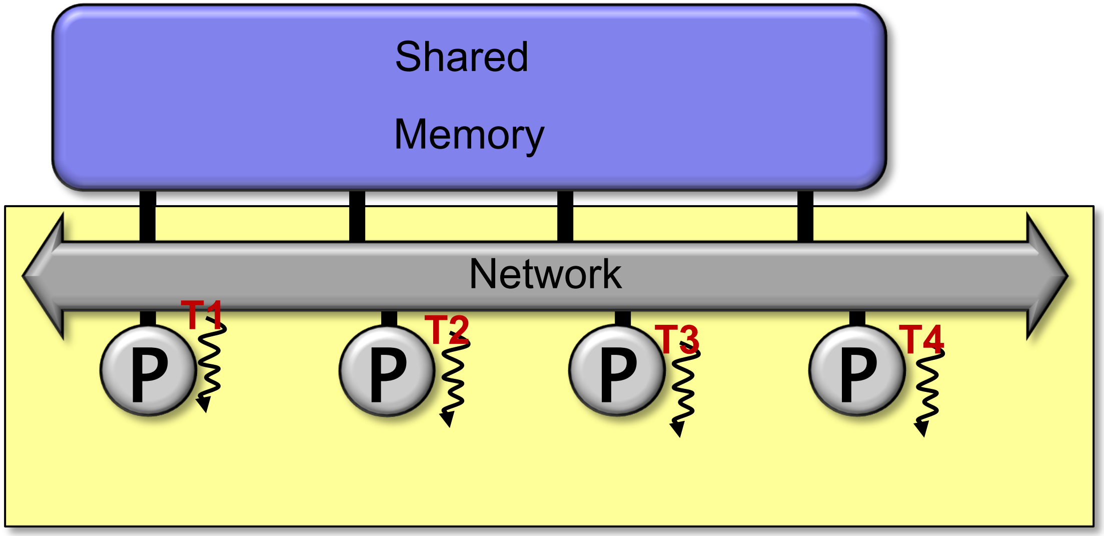




Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

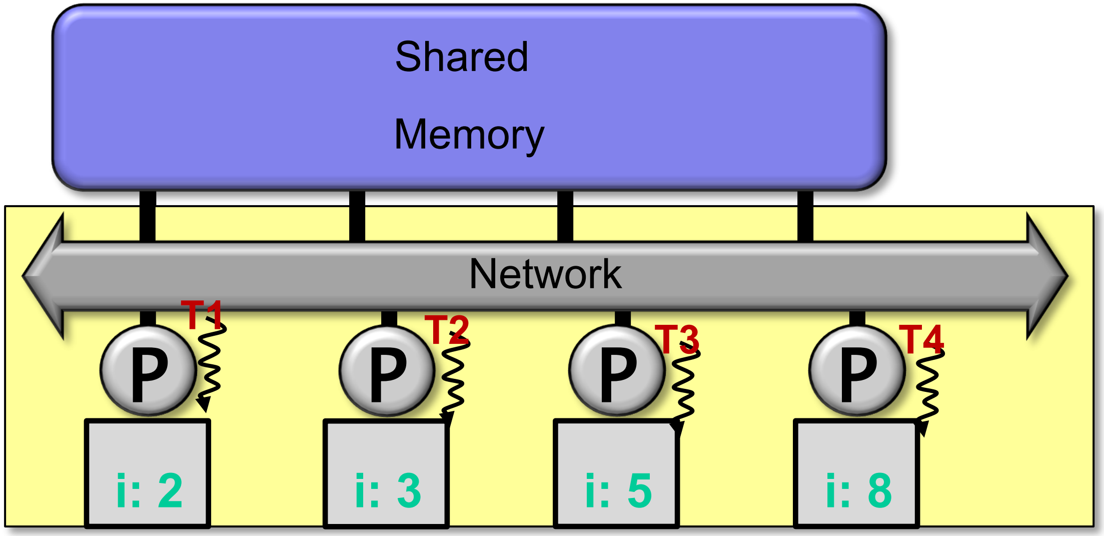




Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.



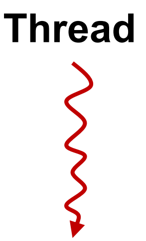
Programming Model: Shared Memory

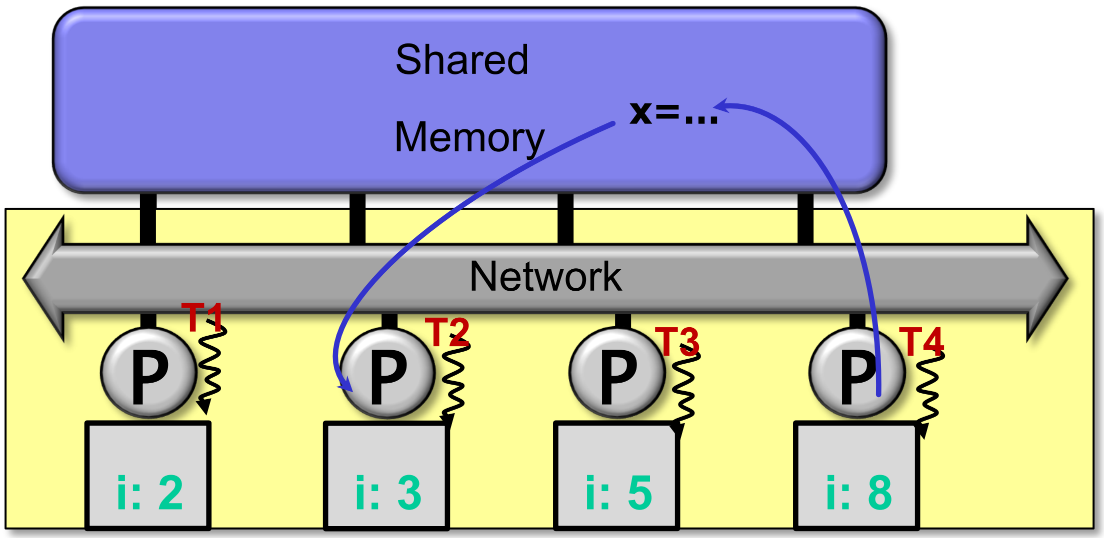
Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.

Also a set of shared variables, e.g., static variables.

- Threads communicate implicitly by writing and reading shared variables.





Slide Source: Demmel

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Overview of POSIX Threads

- POSIX: ***P**ortable **O**perating **S**ystem **I**nterface*
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *, const pthread_attr_t *,  void * (*)(void *), void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute; &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attributes: minimum stack size, priority
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

“Simple” Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Synchronization

- Threads interact in a multiprogrammed system
  - To share resources (such as shared data)
  - To coordinate their execution
- Arbitrary interleaving of thread executions can have unexpected consequences
  - We need a way to restrict the possible interleavings of executions
  - Scheduling is invisible to the application => cannot know when we lose control of the CPU and another thread/process runs
- Synchronization is the mechanism that gives us this control

Motivating Example

```
EggRun(fridge *f) {
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
        eggs_left = buy_carton();
        f->egg_count += eggs_left;
    }
}
```

```
EggRun(fridge *f) {
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
        eggs_left = buy_carton();
        f->egg_count += eggs_left;
    }
}
```

- Separate threads, which may run concurrently; eggs_left is local to each thread while the f->egg_count is shared
- Assume fridge has no eggs initially
- Think about potential schedules for these two threads

Interleaved Schedules

- The execution of the two threads can be interleaved:

T1:	T2:
`int eggs_left = f->egg_count;` `if(eggs_left == 0) {`	
	`int eggs_left = f->egg_count;` `if(eggs_left == 0) {`

`    eggs_left = buy_carton();`
`    f->egg_count += eggs_left;`
`}`	

	`    eggs_left  = buy_carton();`
	`    f->egg_count += eggs_left;`
	`}`

time

Interleaved Schedules

- The execution of the two threads can be interleaved:

T1:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

T2:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

T1:

```
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

T2:

```
    eggs_left  = buy_carton();
    f->egg_count += eggs_left;
}
```

time

We end up buying **two** cartons of eggs

Interleaved Schedules

- The execution of the two threads can be interleaved:

T1:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

time

T2:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

Interleaved Schedules

- The execution of the two threads can be interleaved:

T1:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

T2:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

time

We end up buying **one** carton of eggs

Race conditions and synchronization

- What happens when 2 or more concurrent threads manipulate a *shared resource* (e.g., a piece of data) without any synchronization?
  - The outcome depends on the order in which accesses take place!
  - This is called a ***race condition*** f->egg_count += (unsynchronized write to shared memory)
- We need to ensure that only one thread at a time can manipulate the shared resource
  - So that we can reason about correct program behavior

=> We need ***synchronization***

How do we handle this?

- How about whoever gets to check first, locks the fridge and takes the sole key, for the duration of the entire grocery run?
    - Nobody else can unlock the shared resource until the key owner unlocks it

Mutual Exclusion

- Given:
  - A set of *n* threads, $T_0, T_1, \ldots, T_{n-1}$
  - A set of resources shared between threads
  - A segment of code which accesses the shared resources, called the *critical section, CS*
- We want to ensure that:
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

Mutex locks

- Typically associated to a resource, to ensure one access at a time, to that resource
- Ensure mutual exclusion to a critical section
- For Mutexes, a thread go to sleep when they see the lock is busy.

Spinlock Implementation

- There are two operations on locks: *acquire()* and *release()*

```
boolean lock;

void acquire(boolean *lock) {
        while(test_and_set(lock));
}

void release(boolean *lock) {
        *lock = false;
}
```

When false, we know that we've acquired it

To release, simply turn it to false.

- This is a *spinlock*
  - Uses *busy waiting* - thread continually executes *while* loop in *acquire()* , consumes CPU cycles

Spinlocks vs Mutex

- Spinlocks are built on machine instructions and because *busy waiting* they will waste CPU cycles.
- Mutexes are usually the better choice because a thread goes to sleep if the lock is busy allowing another thread to execute on that core/CPU.
- Putting threads to sleep is expensive so for very short tasks it might not be beneficial to use a mutex.
- However, most modern systems will allow a mutex to spinlock for a very short amount of time, if this seems beneficial.

Next up ...

- Using locks for synchronization
- Common mistakes, potential correctness problems
- Coarse-grained vs. fine-grained locking
- Deadlocks

POSIX mutex API

- Pthreads library has builtin mutexes
    - You've seen these in the labs already
- Basic API:
    - pthread_mutex_t mutex;
    - pthread_mutex_init(pthread_mutex_t *mutex);
    - pthread_mutex_lock(pthread_mutex_t *mutex);
    - pthread_mutex_unlock(pthread_mutex_t *mutex);

Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {
    int egg_count;
    double milk_qty;
    pthread_mutex_t lock;
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {
    pthread_mutex_lock(f->lock);
    if(f->egg_count > 0) {
        f->egg_count --;
    }
    else {
        printf("Plan B: cereal\n");
    }
    pthread_mutex_unlock(f->lock);
}
```

```
EggRun(fridge *f) {
    pthread_mutex_lock(f->lock);
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
      eggs_left = buy_carton();
      f->egg_count += eggs_left;
    }
    pthread_mutex_unlock(f->lock);

    printf("Eggs refilled: %d remaining!", f->egg_count);
}
```

Problem!

No lock around printf in the yellow box so possible bogus output of

`Eggs refilled: 0 remaining!`

not as expected i guess

Potential correctness problems

- Careful about losing track of a lock without unlocking
  - e.g., what happens here:

```
bool CanEatEggs(fridge *f) {
   pthread_mutex_lock(f->lock);
   int eggs_left = f->egg_count;
   if(eggs_left == 0){
       printf("Oh no!\n");
       return false;      lock not released !
   }
   printf("Yummy, eggs!\n");
   pthread_mutex_unlock(f->lock);
   return true;
}
```

- If a thread never releases a lock, all other waiting threads are stuck
  - Such concurrency bugs are called deadlocks! (more on this later...)

Locking – coarse vs fine-grained

- Locking large sections of code may not be efficient => limits concurrency
- What if T1 wants to do a MilkRun, while T2 does an EggRun?
    - Locking the fridge for the EggRun won't allow a MilkRun to happen
- Solution: fine-grained locking
    - Use smaller locks, lock only what is needed...
- Advantage: reduces unnecessary waiting/blocking, more parallelism

Example

- Separate locks => can run in parallel, higher degree of concurrency

```
typedef struct {
    int egg_count;
    pthread_mutex_t egg_lock;

    double milk_qty;
    pthread_mutex_t milk_lock;
} fridge;
```

```
MilkRun(fridge *f) {
    pthread_mutex_lock(f->milk_lock);
    double milk_left = f->milk_qty;
    if(milk_left == 0) {
       milk_left = buy_carton();
       f->milk_qty += milk_left;
    }
    pthread_mutex_unlock(f->milk_lock);
}
```

```
EggRun(fridge *f) {
    pthread_mutex_lock(f->egg_lock);
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
       eggs_left = buy_carton();
       f->egg_count += eggs_left;
    }
    pthread_mutex_unlock(f->egg_lock);
}
```

Careful with fine-grained locking

- Morning routine includes eating eggs and drinking milk
- Must have both eggs and milk to eat breakfast, otherwise breakfast is ruined
- MorningRoutine thread executes concurrently with other threads that perform other breakfast routines => Locks are used (see code)
- Problem? Two threads might try to eat the same egg!

```
MorningRoutine(fridge *f, int e, double m) {
   pthread_mutex_lock(f->egg_lock);
   int eggs_left = f->egg_count;
   if(eggs_left > 0) {
      pthread_mutex_unlock(f->egg_lock);

      pthread_mutex_lock(f->milk_lock);
      double milk_left = f->milk_qty;
      if(milk_left > 0) {
         pthread_mutex_unlock(f->milk_lock);

         pthread_mutex_lock(f->egg_lock);
         f->egg_count -= e;
         pthread_mutex_unlock(f->egg_lock);

         pthread_mutex_lock(f->milk_lock);
         f->milk_qty -= m;
         pthread_mutex_unlock(f->milk_lock);
      } else {
         printf("Breakfast is ruined\n");
         pthread_mutex_unlock(f->milk_lock);
      }
   } else {
      printf("Breakfast is ruined\n");
      pthread_mutex_unlock(f->egg_lock);
   }
}
```

Fine-grained locking and atomicity

- Must be aware of the program semantics to correctly use fine-grained locking and guarantee atomicity where race conditions are possible
- Solutions:
    - Restructure the code if possible
    - Lock larger sections to guarantee atomicity
- Let’s fix the example...

Fine-grained locking and atomicity

```
MorningRoutine(fridge *f, int e, double m) {
             pthread_mutex_lock(f->egg_lock);
   int eggs_left = f->egg_count;
   if(eggs_left > 0) {

      pthread_mutex_lock(f->milk_lock);
      double milk_left = f->milk_qty;
      if(milk_left > 0) {

         f->egg_count -= e;
         pthread_mutex_unlock(f->egg_lock);

         f->milk_qty -= m;
         pthread_mutex_unlock(f->milk_lock);
      } else {
         printf("Breakfast is ruined\n");
         pthread_mutex_unlock(f->milk_lock);
      }
   } else {
      printf("Breakfast is ruined\n");
      pthread_mutex_unlock(f->egg_lock);
   }
}
```

a bit more coarser parallelism

The egg_lock might never get unlocked if a thread makes it to this line

Other problems with fine-grained locking

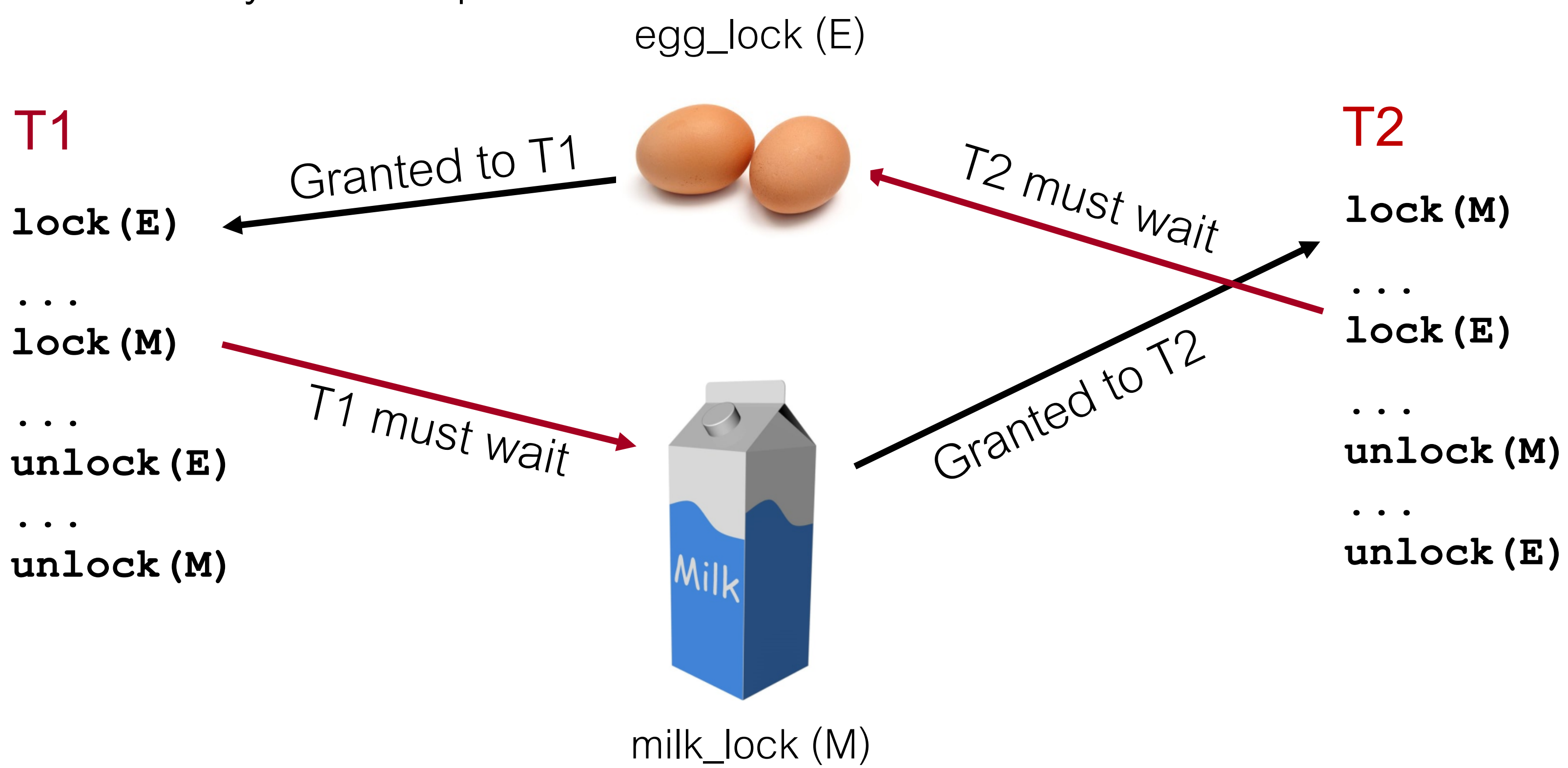
- If we have two morning routines, eating egg then milk, eating milk then egg

```
MorningRoutine_1(fridge *f, int e, double m) {
   pthread_mutex_lock(f->egg_lock);
   int eggs_left = f->egg_count;
   if(eggs_left > 0) {

      pthread_mutex_lock(f->milk_lock);
      double milk_left = f->milk_qty;
      if(milk_left > 0) {

         f->egg_count -= e;
         pthread_mutex_unlock(f->egg_lock);

         f->milk_qty -= m;
         pthread_mutex_unlock(f->milk_lock);

      } else {
         printf("Breakfast is ruined\n");
         pthread_mutex_unlock(f->milk_lock);
         pthread_mutex_unlock(f->egg_lock);
      }

   } else {
      printf("Breakfast is ruined\n");
      pthread_mutex_unlock(f->egg_lock);
   }
}
```

```
MorningRoutine_2(fridge *f, int e, double m) {
   pthread_mutex_lock(f->milk_lock);
   double milk_left = f->milk_qty;
   if(milk_left > 0) {

      pthread_mutex_lock(f->egg_lock);
      int eggs_left = f->egg_count;
      if(eggs_left > 0) {

         f->milk_qty -= m;
         pthread_mutex_unlock(f->milk_lock);

         f->egg_count -= e;
         pthread_mutex_unlock(f->egg_lock);

      } else {
         printf("Breakfast is ruined\n");
         pthread_mutex_unlock(f->milk_lock);
         pthread_mutex_unlock(f->egg_lock);
      }

   } else {
      printf("Breakfast is ruined\n");
      pthread_mutex_unlock(f->milk_lock);
   }
}
```

Deadlocks

- The mutual blocking of a set of threads (or processes)
- Each process/thread in the set is blocked, waiting for a lock which can only be unlocked by another process/thread in the set



- Simplest way to break the deadlock: always acquire locks in the same order!
    - Must enforce the same ordering in every piece of code where we acquire more than 1 lock

Next up...

- Overheads of locking
- Barrier construct

Overheads of locking

- When threads access the same locks => lock contention!
- If lots of threads are contending for the same lock, impacts performance
- Example: simple access to a shared counter by a handful of threads

```
void* do_work(void* arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    counter++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

- Even when no lock contention, acquiring a lock has overheads
  - Try with 1 thread and lots of iterations, with and without the mutex

"Localize" your computations

- Idea: Compute as much as possible locally, use synchronization scarcely
  - Mind you, do not break mutual exclusion when needed for correctness!
- Example:

```
void* do_work(void* arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    counter++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

- Idea: use local counter, update the global shared counter much more rarely...

Other synchronization primitives

- Semaphores
- Condition variables
- We'll focus only on the latter - powerful semantics

Barriers

- Threads that reach the barrier stop until all threads have reached it as well
    - If execution has stages, barrier ensures that data needed from a previous stage is ready
- POSIX has built-in barrier implementation
    - int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier_attr_t *attr,

      unsigned count);
    - int pthread_barrier_wait(pthread_barrier_t *barrier);
    - int pthread_barrier_destroy(pthread_barrier_t* barrier);
    - Check pthread documentation for more details...