Parallel and Concurrent Programming Introduction and Foundation

Marwan Burelle

marwan.burelle@lse.epita.fr
http://wiki-prog.infoprepa.epita.fr

Outline

Going Parallel

Questions you must ask yourself:

- What kind of threads ? System ? Ligthweight ?
- Relation between threads and processus ?
- Which API ? System one ? Generic/Portable one ?
- What locks and synchronization tools are available ?
- More abstraction ?

Available Tools

POSIX Threads (*pthreads*)

Relatively complete, but unavailable outside of the POSIX world. A little bit old and lack of higher-level constructions. Very good as a back-end for higher-level and portable API.

QT/BOOST/SDL/. . . threads

Third parties libs may provide their own API. More portable than systems' one, they bind you to the whole lib.

C++11/C11

C/C++ standards were recently extended with a real parallelism support and pretty-good threads API. This is probably the better choice: more portable and more stable.

What Do We Need ?

- **Launching threads:** we want to launch threads and give them works to do.
- **Waiting threads:** we want to be able to wait for completion of a thread and eventually retrieve some result.
- **Cancelling threads:** we sometime need to stop a running thread.
- **Locking shared data:** we need some mechanism to lock shared data in order to protect concurrent access.
- **Other synchronization tools:** finally we need some clever tools for synchronization (mostly for correctly waiting.)

Threads

Overview

Multi-threaded Hello World !

hello_world.c

```c
void *hello(void *p) {
  printf("<%zu>: Hello World!\n",(*(size_t*)p));
  pthread_exit(NULL);
}

int main() {
  size_t          ids[] = {1,2};
  pthread_t       th[2];
  pthread_create(th, NULL, hello, ids);
  pthread_create(th + 1, NULL, hello, ids + 1);
  pthread_join(th[0], NULL);
  pthread_join(th[1], NULL);
  return 0;
}
```

Creating A Thread

Syntax: pthread_create(3)

```
int pthread_create(pthread_t *th,
                   const pthread_attr_t *attr,
                   void *(*run) (void *),
                   void *param);
```

- This function create a thread, save the handler in `th` and run function `run(param)`.
- The handler `th` let you wait for, or cancel a thread (and some other things)
- The running function take a pointer and return a pointer.

Waiting Or Canceling

Syntax: pthread_join(3)

```
int pthread_join(pthread_t th, void **ret);
```

Syntax: pthread_cancel(3)

```
int pthread_cancel(pthread_t th);
```

- `pthread_join(th,&ret)` let you wait for the completion of thread `th` and get the returned value in `ret`.
- If you don't care about the returned value, you can pass `NULL` directly.
- `pthread_cancel(th)` let you cancel (kill) a running thread.

Attributes ?

- A thread has several properties that can be set: joinable or detached, scheduling policy . . .
- The *detached state* (joinable or detached) describes the link between the created thread and the main thread: a joinable thread won't last after the end of the main thread.
- Most attributes are intended for specific purpose and you can ignore them, passing `NULL` as the attribute parameter.
- Attributes are created and manipulated using `pthread_attr_init(3)` and other `pthread_attr_*` functions.

Overview

What About C11 ?

- C11 is the new ISO C standard since december 2011.
- C11 standard tries to solve two main issues in C parallel programming: **the need for a portable unified library** and **a memory model aware of concurrency**.
- **C11 Threads API:**
  - Defines a portable API that should relies on a system specific one.
  - It uses almost the same naming as *pthreads*
  - Provides some more modern extensions like **atomic types**.
- **C11 New Memory Model:**
  - It extends the notion of sequence points in concurrent context.
  - Memory access and especially accesses' order are now formally described in a concurrent context.

C11 Memory Model

- Back to previous C standard, the only things you can rely on was sequence points: operations between two sequence points can happen in any order. This let the compiler (and, somehow, the processor) reorder your code in way that fits its optimization strategy.
- C11 Memory Model (and C++11 one) extends this idea two a concurrent context: it specifies sequenciality constraints and memory access (loads and stores.)
- Most common constraints:
  - **Atomicity:** the operation must act as one single step
  - **Release Semantic:** the processor guarantees that all past writes (store operations) have completed and become visible by the time that the release happens
  - **Acquire Semantic:** the processor guarantees that no future reads (load operations) have started yet so that it will see any writes released by other processors
  - **Memory fence:** combine release and acquire.

C11 API

- The API provides operations very similar to *pthreads*
- The idea is to provide an easy transition from system API to portable API.

C11 Threads

```
typedef int (*thrd_start_t)(void *);
int thrd_create( thrd_t *thr, thrd_start_t func,
                 void *arg );
int thrd_join( thrd_t thr, int *res );
int thrd_detach( thrd_t thr );
```

- The API provides locks.

Locking techniques

How to lock ?

- Petterson's Algorithm ensure mutual exclusion and other properties but it's not the best choice.
- What are the available techniques for locking shared ressources ?
  - Memory and interruptions blocking;
  - *Low-level* primitives;
  - *API-level* locking routines;
  - *Higher-level* approach (semaphore, monitor . . .)

Overview

Simple Mutual Exclusion

- Simple shared variables (no complex sync)
- *Transactions I/O*, like multi-lines output that must be group but can't be output in one command.

Readers And Writers

- In this classical problem, threads are separated in two sets: readers that only read the shared data and writers that modify the shared data.
- Multiple readers can access data at the same time while writers must be alone.
- There exists solutions with simple locks, but using dedicated mechanism is far simpler.
- Depending on priority choice, writers can suffer starvation: since readers block access to writers, if you let new readers access data when a writer is waiting for the ressource, the ressource may never be free for writing.

Producers/Consumers

- Another classics, threads share a queue.
- Threads pushing elements are called producers while threads poping elements are called consumers.
- We should maintain the coherence of the data structure.
- When the queue is not available for certain operations (no element to be poped for consumers or no room for new element) threads must be blocked until the ressource is available.
- This problem introduce a new kind of issue: synchronization. Threads wait for new event and not only for exclusive access.

Overview

Memory and interruptions blocking

- Interruptions blocking:
  - A way to ensure *atomicity* of operations is to prevent the current thread to leave active mode and other threads to be active.
  - Processors offer the ability to block interruptions, so a running thread won't be interrupted.
  - Such techniques can't be allowed in userland for obvious security and safety reasons.
  - Interruptions blocking are sometimes used in kernel-space (giant locks.)
  - With multiple processors, interruptions blocking doesn't solve all issues.
- Memory blocking:
  - Memory can also be locked by processor and/or threads.
  - Again, this is not permitted in userland.
- Anyway, locking interruptions or memory imply a *global* synchronization point.

Test and Set

Modern (relatively) processors offer atomic primitives to be used safely in userland like *Test and Set*.
Test and Set: is an atomic operation simulating the following code:

TS

```
/* mem: a shared ressources
 * reg: a thread local variable (ie a register)
 */
void TS(unsigned *mem, unsigned reg)
{
  reg = *mem; // save the value
  *mem = 1;   // set to "true"
}
```

Since, this is performed atomically, we can implement simple *spin-lock*:

Simple spin-lock

```
  TS(mem, reg);   // was it "false"
  while (reg)     // no ? -> loop
    TS(mem, reg); // test again ...
  /* CS */
  *mem = 0;       // set back to "false"
```

Compare and Swap (CAS)

- *Compare and Swap* is a better variation of *Test and Set*: it compare a memory location with a value and, if the test return true, it sets the memory location to a new value. *Compare and Swap* (as *Test and Set*) is atomic.
- *Compare and Swap* is often used for *lock* implementations, but is also primordial for most *lock-free* algorithms.

CAS mimic the following code:

CAS

```
int CAS(int *mem, int testval, int newval)
{
  int                    res = *mem;
  if (*mem==testval)
    *mem = newval;
  return res;
}
```

Concrete CAS

The *ia32* architecture provides various implementation of *Compare And Swap* (for different sizes) but most higher level languages does not provide operators for it (this is changing with last C/C++ standard.) Here is an example on how to implement a CAS in C:

ASM inline CAS

```
void volatile*
cas (void *volatile *mem,
     void *volatile  cmp,
     void *volatile  newval)
{
  void volatile         *old;
  __asm__ volatile ("lock cmpxchg %3, (%1)\n\t"
          :"=a"(old):"r"(mem),"a"(cmp),"r"(newval));
  return old;
}
```

Example: Operator Assign

- We can use CAS to implement an almost atomic kind of *Operator Assign* (OA) instruction like +=
- For OA weed need to fetch the value in a shared cell, perform our operation and store the new value, but only if cell content has not change.

OpAssignPlus

```
int OpAssignPlus(int *mem, int val)
{
  int                    tmp;
  tmp = *mem;
  while (CAS(mem, tmp, tmp+val) != tmp)
    tmp = *mem;
  return (tmp + val);
}
```

Overview

Mutex locking

- Mutex provides the simplest locking paradigm that one can want.
- Mutex provides two operations:
  - **lock**: if the mutex is free, lock-it, otherwise wait until it's free and lock-it
  - **unlock**: make the mutex free
- Mutex enforces mutual exclusion of critical section with only two basic operations.
- Mutex comes with several *flavors* depending on implementation choices.
- Mutex is the most common locking facility provides by threading API.

Mutex flavors

- When waiting, mutex can *spin* or *sleep*
- Spinning mutex can use *yield*
- Mutex can be fair (or not)
- Mutex can enforce a FIFO ordering (or not)
- Mutex can be reentering (or not)
- Some mutex can provide a *try lock* operation

To Spin Or Not, To Spin That is the Question

- Spin waiting is often considered as a bad practice:
  - Spin waiting often opens priority inversion issues
  - Spin waiting consumes ressources for doing nothing
  - Since spin waiting implies recurrent test (TS or CAS), it locks memory access by over using atomic primitives.
- On the other hand, passive waiting comes with some issues:
  - Passive waiting means *syscall* and process state modification
  - The cost (time) of putting (and getting it out of) a thread (or a process) in a sleeping state, is often longer than the waiting time itself.
- Spin waiting can be combine with *yield*. Using yield (on small wait) solves most of spin waiting issues.

Implementation

POSIX Threads:

Syntax: pthreads Locks

```
// Passive wait and recursive mutex
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// Spinning non-recursive locks
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

C11 Threads:

Syntax: C11 Threads:

```
// Only one kind, but got a type parameter at creation.
int mtx_lock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

Barrier

- While mutex prevent other threads to enter a section simultaneously, barriers will block threads until a sufficient number is waiting.
- Barrier offers a *phase* synchronization: every threads waiting for the barrier will be awaken simultaneously.
- When the barrier is initialized, we fix the number of threads required for the barrier to open.
- Barrier has one operation: *wait*.
- *Openning* the barrier won't let *latter* threads to pass directly.
- Barrier often provides a way to inform the *last thread* that is the one that make the barrier open.

Implementations

POSIX Threads:

Syntax: Barrier

```
int
pthread_barrier_init(
        pthread_barrier_t           *barrier,
        const pthread_barrierattr_t *attr,
        unsigned                     count
);

int
pthread_barrier_wait(
        pthread_barrier_t           *barrier
);
```

There's no barrier in C11.

Read/Write locks

- The problem: a set of threads are using a shared piece of data, some are only reading it (readers), while others are modifying it (writers.)
- We may let several readers accessing the data concurrently, but a writer must be alone when modifying the shared data.
- Read/Write locks offer a mechanism for that issue: a thread can acquire the lock, only for reading (letting other readers being able to do the same) or acquire for writing (blocking others.)
- A common issue (and thus a possible implementation choice) is whether writers have higher priority than reader:
  - When a writer asks for the lock, it will wait until no reader owns the lock;
  - When a writer is waiting, should the lock be acquired by new readers ?

Implementations

POSIX Threads:

Syntax: Reader/Writer

```
int
pthread_rwlock_rdlock(
  pthread_rwlock_t *rwlock
);

int
pthread_rwlock_wrlock(
  pthread_rwlock_t *rwlock
);
```

There's no read/write locks in C11.

Read Copy Update

- The Read Copy Update (RCU) is a technique to solve issues similar to Reader/Writers problem.
- It is heavily used in Linux kernel, but it is also subject to controversy and patent war . . .
- The idea is simple: the readers are never blocked, when a thread try to update the shared data it must keep a copy of the old data until all readers leave the *critical read section*.
- The RCU mechanism works very well with pointer based data (like linked lists.)
- To use it with buffer like read/write locks, the writer should copy data and change the pointer when all readers pointing to the old data are done.
- Thanks to atomic read of aligned values in most processors, readers don't need any protection.

Condition Variables

- Condition variables offers a way to put a thread in a sleeping state, until some events occurs.
- Condition offers two operations:
  - *wait*: the calling thread will pause until someone call *signal*;
  - *signal*: wake a thread waiting on the condition (if any.)
- A condition variable is always associated with a lock (mutex): we first lock to test, then if needed we wait. Moving to wait state will free the mutex which will be given back to it after the wait.
- The classical use of a condition variable is:

mutex

```
lock(mutex);                  // we need to be alone
while ( some conditions )     // do we need to wait
  wait(condvar, mutex);       // yes => sleep
...                           // we pass, do our job
unlock(mutex);                // we-re done
```

Condition variables: usecase

- Condition variables are used to solve producer/consumer problem:

consumer

```
void consumer () {
  for (;;) {
    void        *data;
    lock(mutex);
    while (q.is_empty())
      wait(cond, mutex);
    data = q.take();
    unlock(mutex);
    // do something
  }
}
```

producer

```
void producer () {
  for (;;) {
    void        *data;
    // produce
    data = ... ;
    lock(mutex);
    q.push(data);
    unlock(mutex);
    signal(cond);
  }
}
```

Implementations

POSIX Threads:

Syntax: pthread_cond_t

```
int
pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex
);
int
pthread_cond_signal(
    pthread_cond_t *cond
);
```

C11 Threads:

Syntax: cnd_t

```
int cnd_wait(cnd_t* cond, mtx_t* mutex);
int cnd_signal(cnd_t *cond ;
```

Overview

Semaphore: What the hell is that ?

- A semaphore is a shared counter with a specific semantics for the decrease/increase operations.
- Normally, a semaphore maintain a *FIFO* waiting queue.
- The two classic operations are:
  - **P**: if the counter is strictly positive, decrease it (by one), otherwise the calling thread is push to sleep, waiting for the counter be positive again.
  - **V**: increase the counter, waking the first waiting thread when needed.
- Since semaphores use a queue, synchronisation using semaphores can consider *fair*: each thread will wait a finite time for the protected ressource. The property is even more precise, since a waiting thread will see (at least) every other threads accessing the ressource exactly one time before it.

Semaphore's classics

- The counter value of the semaphore can be initialize with any positive integer (zero inclusive.)
- A semaphore with an initial value of 1 can act as a fair *mutex*.
- Semaphore can be used as a condition counter, simplifying classic problems such as *Producer/Consumer*.
- Operations' name **P** and **V** comes from Dijkstra's first Semaphores' presentation and probably mean something in dutch. But, implementations often use more explicit names like *wait* for **P** and *post* for **V**.

Producer/Consumer with semaphores

Shared Symbol

```
semaphore mutex = new semaphore(1);
semaphore size  = new semaphore(0);
```

consumer

```
void consumer () {
  for (;;) {
    void        *data;
    P(size);
    P(mutex);
    data = q.take();
    V(mutex);
    // do something
  }
}
```

producer

```
void producer () {
  for (;;) {
    void        *data;
    // produce
    P(mutex);
    q.push(data);
    V(size);
    V(mutex);
  }
}
```

Draft Implementation of Semaphore

Structures

```
semaphore {
  unsigned          count;
  mutex             m;
  condition         c;
};
```

P

```
void P(semaphore sem){
  lock(sem.m);
  while (sem.count == 0)
    wait(sem.c, sem.m);
  sem.count--;
  unlock(sem.m)
}
```

V

```
void V(semaphore sem){
  lock(sem.m);
  sem.count++;
  unlock(sem.m);
  signal(sem.c);
}
```

Implementations

POSIX semaphores (separated from pthreads . . .)

Syntax: sem_t

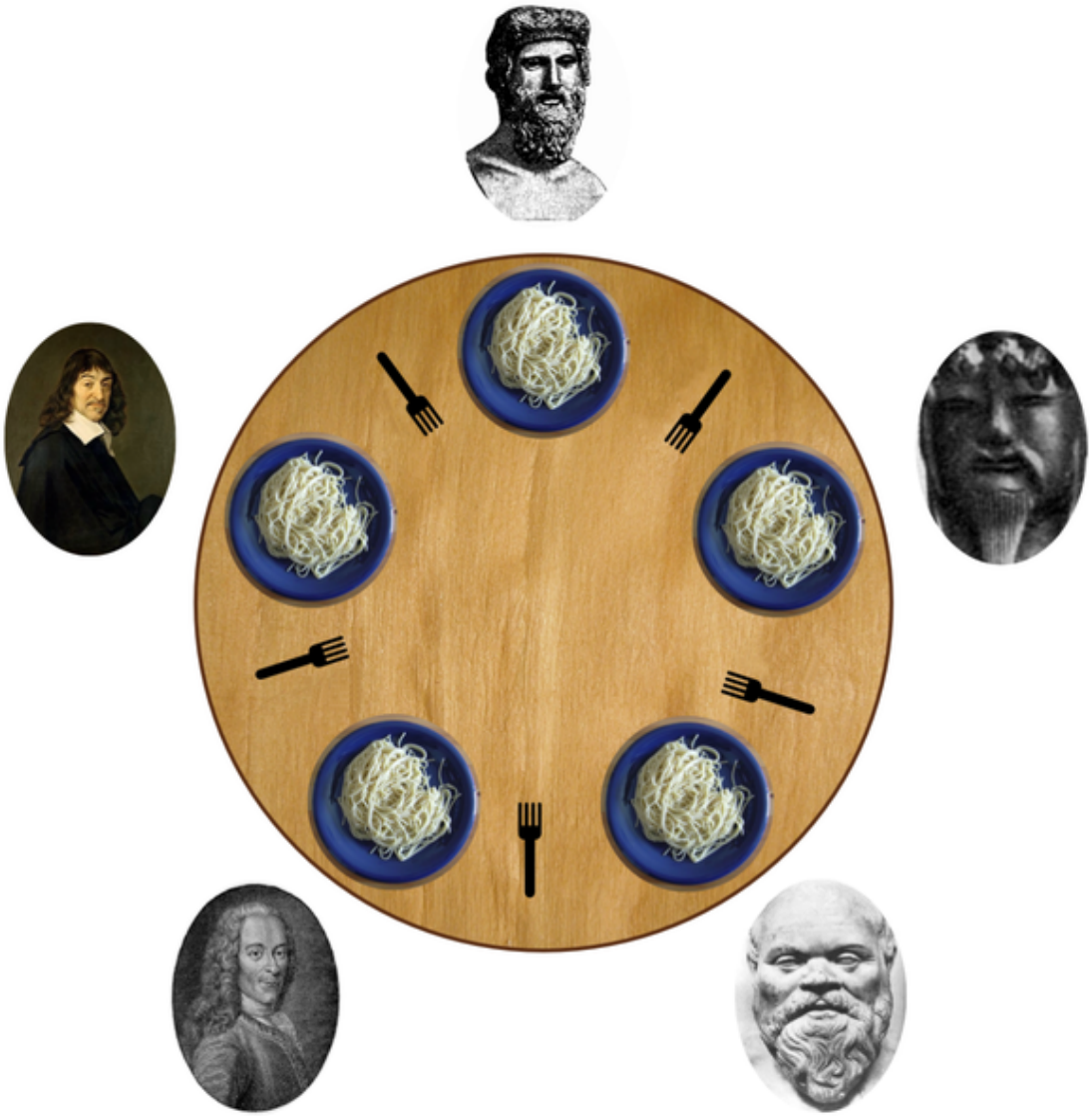
```
// init the semaphore
int
sem_init(
  sem_t *sem,
  int pshared,   // sharing with process ?
  unsigned value
);
// P operation
int sem_wait(sem_t *sem);
// V operation
int sem_post(sem_t *sem);
```

Monitors

- Monitors are abstraction of concurrency mechanism.
- Monitors are more Object Oriented than other synchronization tools.
- The idea is to provide objects where method execution are done in mutual exclusion.
- Monitors come with condition variables
- Modern OO languages integrate somehow monitors:
  - In Java every object is a monitor but only methods marked with `synchronized` are in mutual exclusion.
  - Java's monitor provide a simplified mechanism in place of condition variables.
  - C# and D follow Java's approach.
  - Protected objects in ADA are monitors.
  - ...

Overview

The Dining Philosophers



The Dining Philosophers

- A great *classic* in concurrency by Hoare (in fact a *retold version* of an illustrative example by Dijkstra.)
- The first goal is to illustrate **deadlock** and **starvation**.
- The problem is quite simple:
  - N philosophers (originally $N = 5$) are sitting around a round table.
  - There's only N chopstick on the table, each one between two philosophers.
  - When a philosopher want to eat, he must acquire his left and his right chopstick.
- Naive solutions will cause deadlock and/or starvation.

mutex and condition based solution

Dinning Philosophers

```
/* Dining Philosophers */
#define _XOPEN_SOURCE 600

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>

#define NPHI 5
#define LEFT(k) (((k)+(NPHI-1))%NPHI)
#define RIGHT(k) (((k)+1)%NPHI)

enum e_state {THINKING,EATING,HUNGRY};

typedef struct s_table *table;
struct s_table {
  enum e_state states[NPHI];
  pthread_cond_t        can_eat[NPHI];
  pthread_mutex_t      *lock;
};

struct s_thparams {
  table table;
  pthread_barrier_t    *sync;
  int id;
};
```

Dining Philosophers

```
/* return 1 after receiving SIGINT */
int is_done(int yes) {
 static pthread_spinlock_t *lock=NULL;
 static int                 done=0;
 if (!lock) {
  lock=malloc(sizeof(pthread_spinlock_t));
  pthread_spin_init(lock,
        PTHREAD_PROCESS_PRIVATE);
 }
 pthread_spin_lock(lock);
 if (yes) done = yes;
 pthread_spin_unlock(lock);
 return done;
}

/* where all the magic is !  */
/* test if we are hungry and */
/* our neighbors do no eat   */
void test(table t, int k) {
 if (t->states[k] == HUNGRY
     && t->states[LEFT(k)] != EATING
     && t->states[RIGHT(k)] != EATING) {
  t->states[k] = EATING;
  pthread_cond_signal(&(t->can_eat[k]));
 }
}
```

mutex and condition based solution

Dining Philosophers

```
void pick(table t, int i) {
 pthread_mutex_lock(t->lock);
 t->states[i] = HUNGRY;
 printf("Philosopher %d: hungry\n",i);
 test(t,i);
 while (t->states[i] != EATING)
  pthread_cond_wait(&t->can_eat[i],
                    t->lock);
 printf("Philosopher %d: eating\n",i);
 pthread_mutex_unlock(t->lock);
}

void put(table t, int i) {
 pthread_mutex_lock(t->lock);
 t->states[i] = THINKING;
 printf("Philosopher %d: thinking\n",i);
 test(t,LEFT(i));
 test(t,RIGHT(i));
 pthread_mutex_unlock(t->lock);
}

void thinking() {
 struct timespec        reg;
 reg.tv_sec = random()%6;
 reg.tv_nsec = 1000000*(random()%1000);
 if (nanosleep(&reg,NULL) == -1) {
  if (errno != EINTR || is_done(0))
  pthread_exit(NULL);
 }
}
```

Dining Philosophers

```
void eating() {
 struct timespec        reg;
 reg.tv_sec = random()%2;
 reg.tv_nsec = 1000000*(random()%1000);
 nanosleep(&reg,NULL);
}

void *philosopher(void *ptr) {
 struct s_thparams     *p;
 p = ptr;
 pthread_barrier_wait(p->sync);
 printf("Philosopher %d:thinking\n",p->id);
 while (!is_done(0)) {
  thinking();
  pick(p->table, p->id);
  eating();
  put(p->table, p->id);
 }
 pthread_exit(NULL);
}

void handle_int(int sig) {
 is_done(1);
 signal(sig,handle_int);
}
```

mutex and condition based solution

Dining Philosophers

```
int main(int argc, char *argv[]) {
 table                t;
 struct s_thparams    *p;
 pthread_t            th[NPHI];
 pthread_mutex_t      lock;
 pthread_barrier_t    sync;
 size_t               i, seed=42;

 signal(SIGINT, handle_int);

 if (argc>1)
  seed = atoi(argv[1]);
 srandom(seed);

 t = malloc(sizeof (struct s_table));
 pthread_barrier_init(&sync,NULL,NPHI);
 pthread_mutex_init(&lock,NULL);
 t->lock = &lock;
```

Dining Philosophers

```
 for (i=0; i<NPHI; ++i) {
  t->states[i] = THINKING;
  pthread_cond_init(&t->can_eat[i],NULL);
 }

 for (i=0; i<NPHI; ++i) {
  p = malloc(sizeof (struct s_thparams));
  p->table = t;
  p->sync = &sync;
  p->id = i;
  pthread_create(th+i,NULL,philosopher,p);
 }

 for (i=0; i<NPHI; ++i)
  pthread_join(th[i], NULL);

 return 0;
}
```

Sharing Resources

- The dining philosophers problem emphasizes the need of synchronisation when dealing with shared resources.
- Even with a simple mutex per chopstick, the execution may not (will probably not) be correct, ending with either a global deadlock or some philosophers in starvation.
- It is easy to see that no more than half of the philosophers can eat at the same time: **sharing resources implies less parallelism !**
- This kind of situation is what we want to avoid: *a lot of dependencies between threads*.
- A good parallel program try to avoid shared resources when possible. A good *division* of a problem for parallel computing will divide the global task into *independant tasks*.

C++11 Threads And Locks API

Disclaimer

- You need to put an hand on a working compiler, I personally use `clang++` but a recent version of `g++` should do.
- If your using clang be sure to use `libc++` and not `libstdc++`.
- Be sure to add the `std=c++11` flag.

Overview

Do You Know Lambda ?

- To run threads in C++11, you can use function pointers, but it's far simpler to use lambdas.
- Using lambdas really simplify the way you can transmit state to your threads.
- The idea is to pass static state as value or at least *const ref* directly rather than using parameters.

lambda example

```
int example(int const tab[], int res[], size_t len) {
  std::thread  *worker[8]; size_t step = len/8;
  for (size_t i=0; i < 8; ++i)
    worker[i] = new std::thread
      ( [=](size_t start) {
        int const       *cur = tab + start;
        int const       *end = tab + start + step + 1;
        int     *target = res  + start;
        for (; cur != end; cur++, target++)
          *target = *cur * 2; }, i * step);
  return 0;
}
```

Do You Know Lambda ?

lambda example

```
[=](size_t start) {
  int const       *cur = tab + start;
  int const       *end = tab + start + step + 1;
  int     *target = res  + start;
  for (; cur != end; cur++, target++)
  *target = *cur * 2;
}
```

- [=] indicates that symbols borrowed from the context should be copied in the closure. If you need reference rather than copy, you can declared your lambda with [&]
- For the rest, it's just an anonymous function !

The `std::thread` class

- Similar in spirit with the Java's `Thread` class.
- You give a *function* to the constructor and function's arguments.
- You can use a lambda or any function pointer.
- The object you get can be used for *joining* or *detaching* the thread.

this_thread namespace

- Provides operations acting on the current thread (the one executing the current code.)
  - `std::this_thread::yield()`: give back the remaining time in the slice to the scheduler.
  - `std::this_thread::get_id()`: obtain thread id (supposed to be unique)
  - `std::this_thread::sleep_for()`: put the thread to sleep for the given amount of time (using `std::chrono::duration`)
  - `std::this_thread::sleep_until()`: like the previous one, but using a specific time point.

Overview

Returning Values

- In traditionnal *pthread* model, you can return value and grab it during joins.
- C++11 does not provide such a mechanism.
- Instead we use shared containers through the mean of promises and future.
- You build a promise and give it to the thread, then trough that promise you get the future.
- A future will let you retrieve the value when you need it (or when it is available.)

Promise and Future

```
int example() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::thread th([&](){ p.set_value(42); } );
  return f.get();
}
```

Using Promises And Futures

- The promise should live *inside* the thread you want value from.
- The caller that want to get result(s) must retrieve the future at some point (using the `get_future()` method.)
- Beware that futures and promises have a deleted copy-constructor (but they got a move constructor) and thus can't be copied.
- The `get()` method of the future is blocking: it will only return when the holder of the promise use the `set_value()` method.
- This approach gives you more freedom than the traditionnal *join* mechanism where you can only get a value when the thread is dead.
- The API offers more operations (liked bounded waiting or tests on availability . . .)

Overview

Using std::async

std::async

```
uint64_t ack(uint64_t m, uint64_t n) {
  return (m?(n?(ack(m-1,ack(m,n-1))):ack(m-1,1)):n+1);
}

uint64_t fibo(uint64_t n) {
  return n > 1 ? fibo(n-1) + fibo(n-2): n;
}

int main() {
  std::future<uint64_t> a = std::async(ack,3,12);
  std::future<uint64_t> b = std::async(fibo,45);
  std::cout << a.get() + b.get() << std::endl;
  return 0;
}
```

More On Asynchronous Calls

- `std::async` provides an easy way to call *functions* (using functions in the sense of STL) asynchronously (*i.e.* later, at least between now and when you ask for the result.)
- You get the result of the function through a future, you can even obtain raised exception.
- `std::async` may run your code in a separated thread or lazily depending on the policy (first, but optional parameter):
  - Launching with a thread:
    `std::async(std::launch::async, func, args...)`
  - Launching lazily:
    `std::async(std::launch::deferred, func, args...)`

A Little Trap

Fail !

```
int main() {
  std::async(std::launch::async, []() {f();});
  // the destructor will probably wait for f()
  // and thus next line won't run in parallel.
  std::async(std::launch::async, []() {g();});
  return 0;
}
```

Overview

Simple Locks

- C++11 provides two way of using mutexes:
  - The classical lock/unlock style
  - And the *Resource Acquisition Is Initialization* style
- We got four kind of standard mutex objects:
  - `std::mutex` : classical simple non-recursive mutex.
  - `std::timed_mutex` : implement lock with a timeout.
  - `std::recursive_mutex` : support locking on already owned locks.
  - `std::recursive_timed_mutex` : guest what ?
- All mutexes provides `lock()`, `unlock()` and `try_lock()`
- Mutex classes can not be copied.

RAII

- The simplest way for exception safe locking code.

RAII

```
#include <thread>
#include <mutex>
struct shared { int a, b;
  shared(int x, int y) : a(x), b(y) {}
};
shared shr(0,0);          std::mutex mtx;

void safe() {
  std::lock_guard<std::mutex>   lock(mtx);
  shr.a += 1; shr.b -= 1;
}
int main() {
  std::thread th1(safe);          std::thread th2(safe);
  th1.join();                     th2.join();
  return 0;
}
```

Overview

Condition Variables

Producer/Consumer

```
#include <thread>
#include <mutex>
#include <condition_variable>
struct PCQueue {
  std::queue<int>                 q;
  std::mutex                      m;
  std::condition_variable         cond;
  int consume() {
    std::lock_guard<std::mutex> lock(m);
    while (!q.empty())
      cond.wait(lock);
    return q.pop();
  }
  void produce(int x) {
    std::lock_guard<std::mutex> lock(m);
    q.push(x);
    cond.notify_all();
  }
};
```

Overview

Atomic ?

- An atomic type is a scalar type implemented such that usual operations are intended to be atomic.
- This the only types for which concurrent behavior are defined.
- Most interesting operations are defined over integral types (*i.e.* the ones you can do arithmetics on.)
- You can use atomics for:
  - Simple variable mutual exclusion (without locks)
  - Basis to implement fine grain locking or lock-free algorithms.
- For load/store operations you can define the memory barrier policy (release, acquire, full barrier . . .)

Basic Counter (broken)

Broken Example

```
int main() {
  uint64_t       counter = 0;
  std::future<uint64_t>  f1 =
    std::async([&]() -> uint64_t {
        uint64_t i;
        for (i = 0; i < 65536; ++i) ++counter;
        return i;
      });
  std::future<uint64_t>  f2 =
    std::async([&]() -> uint64_t {
        uint64_t i;
        for (i = 0; i < 65536; ++i) ++counter;
        return i;
      });
  std::cout << "local counters: "
            << f1.get() + f2.get() << std::endl;
  std::cout << "shared counter: " << counter << std::endl;
  return 0;
}
```

Basic Counter (working)

Working Example

```
int main() {
  std::atomic<uint64_t>  counter(0); // Using atomic !
  std::future<uint64_t>  f1 =
    std::async([&]() -> uint64_t {
        uint64_t i;
        for (i = 0; i < 65536; ++i) ++counter;
        return i;
      });
  std::future<uint64_t>  f2 =
    std::async([&]() -> uint64_t {
        uint64_t i;
        for (i = 0; i < 65536; ++i) ++counter;
        return i;
      });
  std::cout << "local counters: "
            << f1.get() + f2.get() << std::endl;
  std::cout << "shared counter: " << counter << std::endl;
  return 0;
}
```

Results

Example:

```
# Broken Version:
un_shell> ./exatomic
local counters: 131072
shared counter: 81035
un_shell> ./exatomic
local counters: 131072
shared counter: 70738

# Working Version
un_shell> ./exatomic
local counters: 131072
shared counter: 131072
un_shell> ./exatomic
local counters: 131072
shared counter: 131072
```

Overview

What ?

- C++11 provides a way to be sure that a function is ran only once by a bunch of threads.
- Threads shares a special flag, and use the template function `call_once()` to run their code.
- One *call* is selected to be run and all other wait until execution is complete.
- There's no control upon which function will be run.
- The function is ran in the thread that invoke it.

Call Me Maybe

call_me_maybe.cc

```cpp
#include <mutex>
#include <thread>
#include <iostream>

std::once_flag  flag;
void itsme() {
  call_once(flag, []() {
      std::cout << "Thread " << std::this_thread::get_id()
                << " won !" << std::endl; });
}

int main() {
  std::thread th1(itsme);       std::thread th3(itsme);
  std::thread th2(itsme);       std::thread th4(itsme);
  th1.join(); th2.join(); th3.join(); th4.join();
  return 0;
}
```