# Parallel and Concurrent Programming Classical Problems, Data structures and Algorithms

Marwan Burelle

marwan.burelle@lse.epita.fr
http://wiki-prog.kh405.net

# Outline

LSE Security System Laboratory of Epita



# Introduction

# Data and Algorithms

- Classical Algorithmic studies emphasis the importance of data structures against algorithms
- Parallel Algorithms share the same trade-off with a bit more stress on data structures
- Clever data structures are needed for performances but also for consistency and determinism
- We need to understand how to:
  - Correctly manage shared data (mutual exclusion)
  - Synchronize threads
  - Avoid as much as possible contention due to locks
- Once data structures are safe and efficient, we can study algorithms and how to smartly use multiple processors.

# 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 . . . )

# Lower level locks

# 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*.

## Example:

*Test and Set*: is an atomic operation simulating the following code:

```
/* 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*:

```
  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.

**Example:**

CAS mimic the following code:

```
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:

**Example:**

```
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.

## Example:

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

LSE Security System Laboratory of Epita



## Mutex and other usual locks

# 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.

# 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.

# Read/Write locks

- The problem: a set of threads are using a shared peace 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 ?

# 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:

```
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
```

- Sometimes one can use a *broadcast* which will try to wake every thread waiting on the condition.

# Condition variables: usecase

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

**Example:**

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

**Example:**

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

# Higher Level: Semaphore and Monitor

# 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

## Example:

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

## Example:

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

## Example:

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

# Draft Implementation of Semaphore

## Example:

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

## Example:

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

## Example:

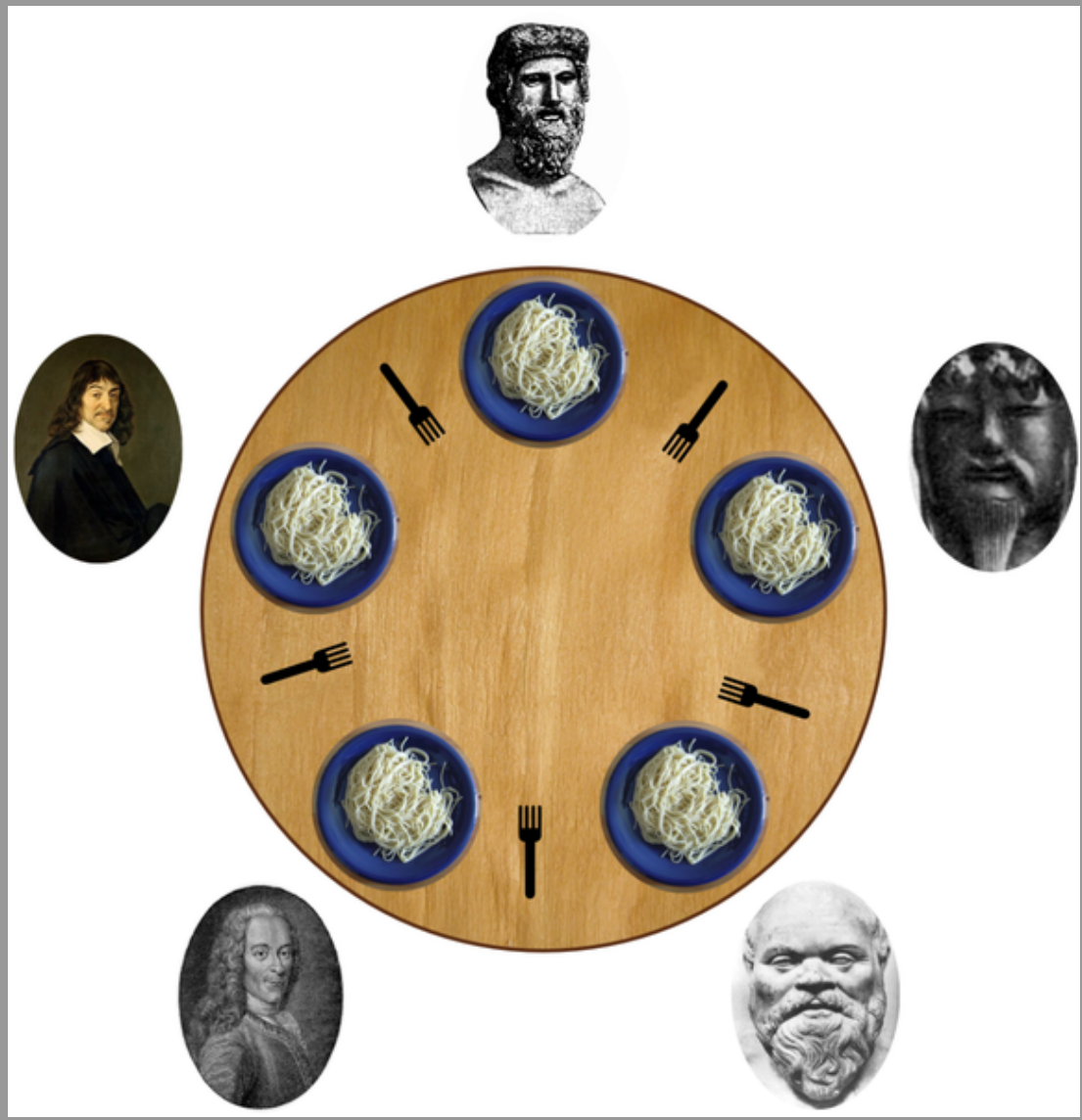
```
void V(semaphore sem){
  lock(sem.m);
  sem.count++;
  unlock(sem.m);
  signal(sem.c);
}
```

# 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.
  - ...

# The Dining Philosophers Problem

# 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

```c
/* 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;
};
```

```c
/* 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

```
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);
 }
}
```

```
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

```
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;

 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*.

# Data Structures

LSE Security System Laboratory of Epita



# Concurrent Collections

# Producers and Consumers Classical Problem

- When using a shared collection, we face two issues:
  - Concurrent accesses;
  - What to do when collection is empty.
- Usual solution for a queue (or any other *push-in/pull-out* collection) is to implement the Producers/Consumers model:
  - The collection is accessed in mutual exclusion;
  - When the collection is empty *pull-out* operations will block until data is available.
- Producers/Consumers is quite easy to implement using semaphores or using mutex and condition variables.
- Producers/Consumers can also be extended to support bounded collections (*push-in* operations may wait until a place is available.)

# Producers and Consumers Seminal Solution

## Example:

```
void push(void *x, t_queue q)
{
  pthread_mutex_lock(q->m);
  q->q = _push(x,q->q);
  pthread_mutex_unlock(q->m);
  sem_post(q->size);
}

void *take(t_queue q)
{
  void                  *x;
  sem_wait(q->size);
  pthread_mutex_lock(q->m);
  x = _take(&q->q);
  pthread_mutex_unlock(q->m);
  return x;
}
```

# Locking Refinement

- Global locking of the collection implies more synchronisation (and thus, less parallelism !)
- Let's consider a *FIFO* queue:
  - Unless there's only one element in the queue, *push-in* and *pull-out* can occur at the same time (careful implementation can also accept concurrent accesses when there's only one element.) [2]
  - The traditionnal circular list implementation of queue can not be used here.
  - The solution is to build the queue using a structures with two pointers (head and tail) on a simple linked list.
- Better locking strategies leads to more parallelism, but as we can see usual implementations may not fit.

# Loose Coupling Concurrent Accesses

- When using *map* collections (collections that map keys to values), we can again improve our locking model.
- When accessing such collection we have two kind of operations: read-only and create/update.
- The idea is to see a *map* as a collection of pairs: all operations on the *map* will get a pair (even the create operation) and locking will only impact the pair and not the whole collection.
- In ordrer to support concurrent read we prefer read/write lock.
- Insertion operations can also be seperated in two distinct activities:
  - We create the cell (our pair) give back the pointer to the caller (with appropriate locking on the cell itself.)
  - Independently, we perform the insertion on the structure using a tasks queue and a seperate worker.
- The later strategy minimize even more the need of synchronisation when accessing our collection.

# Data Structures Concurrent Friendly

- Some data structures are more *concurrent friendly* than others.
- The idea is again to minimize the impact of locking: we prefer structures where modifications can be kept local rather than global.
- Tree structures based are good candidate: most modification algorithm (insertion/suppression/update) can be kept local to a sub-tree and during the traversal we can release lock on *unimpacted* sub-tree.
  - For example, in *B-tree*, it has been proved that read operations can be perfomed without any locks and Write locks are located to modified block [1].
- Doubly linked lists are probably the worst kind of data structures for concurrent accesses: the nature of linked lists implies global locking to all elements accessible from the cell, so any operations on doubly linked lists will lock the whole list.

# Non blocking data structures

- Normally spin waiting is a bad idea, but careful use of spin waiting can increase parallelism in some cases.
- The idea of non-blocking data structures is to interleave the waiting loop with the operation we want to perform.
- Good algorithm for that kind of data structures are harder to implement (and to verify) but offers a more dynamic progression: no thread idle by the system should block another when performing the operation.
- Non blocking operations relies on hardware dependent atomic operations

# Non Blocking Concurrent Queue

- We'll study an implementation of a non blocking queue describes in [2]
- The two classical operations provides progression and all expected safety
- The algorithm uses double CAS (see next slides) to solves the ABA problem.
- Basic idea: when accessing tail or head of the queue, we fetch the front pointer, and in order to update the structure we use a CAS, if it fails we retry from the begining.
- The second interesting point is to finish work of other threads when possible.

# The ABA issue

- When manipulating value using CAS a particular issue can arise: the so called ABA problem.
- The idea is quite simple: the fetched pointer can change several time during between the original fetch and the CAS, and for example we can fetch a A, it can be replaced by a B then by a A again.
- When manipulating data structure this means that the fetched values are incoherent.
- The simpliest way to solve the issue is to use a *double-CAS*: the pointer is concatened to a counter incremented each time we perform a CAS.

# Concurrent Data Model

# Using Data in A Concurrent Context

- Once we have chosen a good data structures, we need to manage concurrent accesses.
- Classical concurrent data structures define locking to enforce global data consistency but problem driven consistency is not considered.
- Most of the time, consistency enforcement provide by data structures are sufficient, but more specific cases requires more attention.
- Even with non-blocking or (almost) lock-free data structures, accessing shared data is a bottleneck (some may call it a *serialization point*.)
- When dealing with shared data, one must consider two major good practices:
  - Enforcing high level of abstraction;
  - Minimize locking by deferring operations to a *data manager* (*asynchronous upates*.)

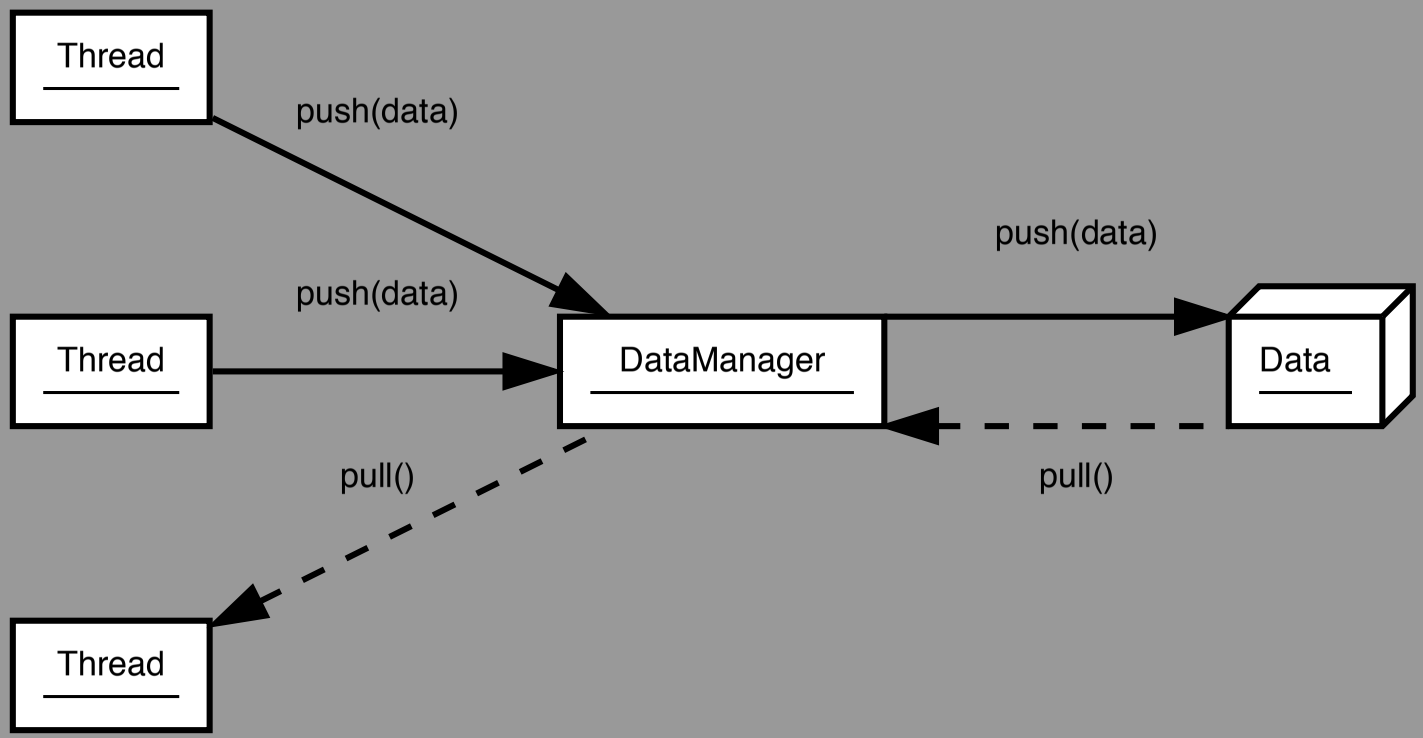
# Data Authority and Concurrent Accesses

- **Enforcing high level of abstraction:**
  - Encapsulation of the operations minimize exposition of the locking policy and thus enforce correct use of the data.
  - When possible, using *monitor* (object with native mutual exclusion) can simplify consistency and locking.
  - As usual, abstraction (and thus encapsulation) offers more possibility to use clever operations implementations.

# Data Authority and Concurrent Accesses

- **Deferring operations to a kind of *data manager*:**
  - Deferring operations can improve parallelism by letting a different worker performs the real operations: the calling thread (the one that issue the operation) won't be blocked (if possible), the data manager will take care of performing the real operations.
  - Since the data manager is the only entity that can perfom accesses to the data, it can work without any lock, nor any blocking stage.
  - Data manager can *re-order* operations (or simply discard operations) to enforce algorithm specific constraint.

# Data Manager



# The Future Value Model

- Future are concurrent version of lazy evaluation in functionnal languages.
- Futures (or *promises* or *delays*) can be modeled in many various ways (depending on language model.)
- In short, a future is a variable whose value is computed independently. Here's a simple schematic example (pseudo language with *implicit future*):

```
// Defer computation to another thread
future int                  v = <expr>;
// some computations
// ...
// We now need access to the future value
x <- 42 + v
```

# Future for real . . .

- Java has future (see `java.util.concurrent.Future`);
- Futures will normaly be part of C++0x;
- Futures exists in sevral *Actor based* languages, functionnal languages (rather natively like in Haskell or AliceML, or byt the means of external libs like in OCaml) and pure object oriented languages (Smalltalk, AmbientTalk, E ... )
- Implementing simple future using pthread is quite simple: the future initialization create a new thread with a pointer to the operation to perform and when we really need the value we perform a *join* on the thread.
- One can also implements future using a tasks based systems.

# Futures' issues

There are several issues when implementing futures. Those issues depend on the usage made of it:

- When we use futures to perform blocking operations or intensive computations, tasks systems may induce important penality.
- Creating a thread for each future induces important overhead.
- In object oriented languages, one have to solve whether message passes should wait on the futures result or not:

```
future int                    v = acker(...);
// ...
v.add(1)
// we can block here waiting for v to complete
// but we can also send the message and let the
// future handle it in time.
```

# Tasks Systems

# Direct Manipulation of Physical Threads

- Physical (*system*) threads are not portable
- Most of the time, physical threads are almost independant process
- Creating, joining and cancelling threads is almost as expensive as process manipulations
- Synchronisation often implies kernel/user context switching
- Scheduling is under system control and doesn't take care of synchronisation and memory issues
- Data segmentation for parallel computing is problem **and** hardware driven:
  - Data must be split in order to respect memory and algorithm constraints
  - Number of physical threads needs to be dependant of the number of processors/cores to maximize performances
- ...

# Light/Logical Threads

- One can implement threads in full user-space (*light threads*) but we loose physical parallelism.
- A good choice would be to implement *logical threads* with scheduling exploiting physical threads.
- Using logical threads introduces loose coupling between problem segmentation and hardware segmentation.
- *Local* scheduling increase code complexity and may introduce overhead.

# Tasks based approach

- A good model for logical threads is a tasks system.
- A task is a sequential unit in a parallel algorithm.
- Tasks perform (*sequential*) computations and may spawn new tasks.
- The tasks system manage scheduling between *open* tasks and available physical threads.
- Tasks systems often use a *threads pool*: the system start a bunch of physical threads and schedule tasks on available threads dynamically.

# Simple tasks system: waiting queue.

- *Producer* schedule new *tasks* by pushing it to the queue.
- *Consumer* take new *tasks* from the queue.
- *Producer* and *Consumer* are physical threads, we call them **worker**.
- Each worker may play both role (or not.)
- Tasks can be input values or data ranges for a fixed task's code.
- It is also possible to implement tasks description so producer can push any kinds of task.
- For most cases, we need to handle a kind of *join*: special task pushed when computation's results are ready, in order to closed unfinished tasks (think of a parallel reduce or parallel Fibonacci numbers computation.)

# Tasks Sytems For Real

- Java `Executor` provides a task-based threading approach
- Intel's TBB (Threading Building Blocks) is completely based on this paradigm:
  - High level tools (such as parallel for) are based on a task and the librairy provides a scheduling mechanism to efficiently executes task.
  - You can also directly use the task system and build you're own partitionning.
  - TBB provides also pipeline mechanism

# Algorithms and Concurrency

# Easy Parallelism

# Problems with simple parallel solutions

- A lot of problems can be solved easily with parallelism: for example when computing a Mandelbrot Set, we can perform the iteration for each pixel independently.
- The remaining issue of *easy parallelism* is scheduling: for our Mandelbrot set we can't start a new thread for each pixel.
- Using tasks systems and range based scheduling offers a good tradeoff between scheduling overhead and efficient usage of physical parallelism.
- Modern tools for parallel computing offers *intelligent* parallel loop constructions (*parallel for*, *parallel reduce* ... ) based on range division strategy statisfying hardware constraints (number of processors, cache affinity ... )

# Parallel *trap*

# Parallel not so parallel

- Some times, parallel version are not so fast, even with multi-processor.
- It is important to keep in mind that speed-up is bound by the real degrees of parallism (Amdahl's law.) Take an example:
  - We have a set of vectors and want to compute the average of each vector;
  - Simple parallel version consiste of running a thread per vector;
  - This does not implies good speed-up (in fact, sequential version runs almost as fast);
  - A better solution is to perform (smart) parallel sums for each vector, the parallel part will be more significant and thus you can have a good speed-up.

**Parallel or not, Parallel that is the question !**

# Parallelism and classical algorithms

- Some classical algorithms won't perform well in parallel context: for example depth first traversal is inherently not parallel.
- Optimizations in classical algorithms can also induce a lot of synchronisation points.
- Backtrack based algorithms can be improved with parallelism, but we must take care of scheduling: if the algorithms have a lot of backtrack point, we have to find a *strategy* to choose which point can be scheduled for parallel execution.

# Bibliography


P. L. Lehman and S. B. Yao.
Efficient locking for concurrent operations on B-trees.
*ACM Trans. on Database Sys.*, 6(4):650, December 1981.

Maged Michael and Michael L. Scott.
Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.
In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–267, Philadelphia, Pennsylvania, USA, 23–26 May 1996.