An Introduction to Parallel Systems

Lecture 4 - Shared Resource Parallelism

Martin Brain

University of Bath

December 6, 2007

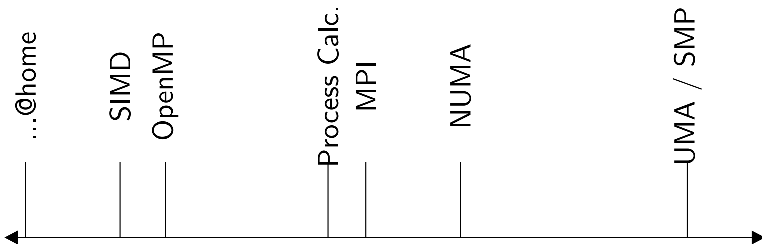
When

- Week 1 Introduction – Who, What, Why, Where, When?
- Week 2 Data Parallelism and Vector Processors
- Week 3 Message Passing Systems
- Week 4 Shared Resource Parallelism

`http://www.cs.bath.ac.uk/~mjb/parallel/`

(sorry I'm a bit behind)

Resource Sharing



Introduction

Threading
- Synchronisation
- Mutexes
- Other Synchronisation Primatives

The Problem with Mutexes...
- Lock Free Algorithms
- STM

Conclusion

What are threads?

Multiple points of (simultaneous) execution that share the same address space.

Shared Code, global variable, heap, kernel resources (i.e. files & sockets)

'Private' Stack, instruction pointer, TLS

Where are threads used?

- Within OS kernel
- In applications using pthreads (DB, Apache, etc.)
- Userspace threading systems

Scheduling Models

But what happens when there are more threads than processors ...

- Pre-emptive vs. co-operative
- Fixed time slice vs. dynamic (tickless)
- Priority scheduling
- (Concurrency)

Using pthreads

Creation `pthread_create(pthread_t *tid, function *start, void *arg)`

Exiting `pthread_exit()`

Killing `pthread_cancel(pthread_t tid)`

Waiting `pthread_join(pthread_t tid)`

Syncronisation Problem 1

```
void *stack[LARGE];
int entries = 0;

void push (void *newItem) {
  stack[entries++] = newItem;
}

void * top () {
  if (entries > 0)
    return stack[entries];
  else
    return NULL;
}

void pop () {
  --entries;
}
```

Syncronisation Problem 1

```
void *stack[LARGE];
int entries = 0;

void push (void *newItem) {
  stack[entries++] = newItem;
}

void * top () {
  if (entries > 0)
    return stack[entries];
  else
    return NULL;
}

void pop () {
  --entries;
}
```

`push()`

1. Read entries
2. Write `stack[entries]`
3. Increment entries
4. Write entries

`access()`

1. Read entries
2. Read `stack[entries]`

`pop()`

1. Read entries
2. Increment entries
3. Write entries

Syncronisation Problem 2

```
node *head;

void push (void *newItem) {
  node *tmp = createNode(newItem);
  tmp->next = head;
  head = tmp;
}

void * pop () {
  node *tmp = head;
  void *r;

  head = tmp->next;
  r = tmp->contents;
  destroyNode(tmp);
  return r;
}
```

`push()`

1. Read `head`
2. Write `head`

`pop()`

1. Read `head`
2. Read `tmp->next`
3. Write `head`

Atomic actions

- Are read and write atomic?

Atomic actions

- Are read and write atomic?
- Compare and swap

```
bool CAS (int *target, int oldValue, int newValue) {
  if (*target == oldValue) {
    *target = newValue;
    return true;
  } else {
    return false;
  }
}
```

Atomic actions

- Are read and write atomic?
- Compare and swap

```
bool CAS (int *target, int oldValue, int newValue) {
  if (*target == oldValue) {
    *target = newValue;
    return true;
  } else {
    return false;
  }
}
```

- Others (test and set, load linked / store conditional, ...)

Atomic actions

- Are read and write atomic?
- Compare and swap

```
bool CAS (int *target, int oldValue, int newValue) {
  if (*target == oldValue) {
    *target = newValue;
    return true;
  } else {
    return false;
  }
}
```

- Others (test and set, load linked / store conditional, ...)
- Key problem is composing atomic operations into the *critical sections* we need.

Things we are going to pretend are true...

- "Read loads from memory"
- "Write outputs to memory"
- "Instructions happen in program order"
- "Reads and write are immediately effective and occur in program order"

Mutexes

- A device for handling MUTual EXclusion.
- Has two states, *unlocked* and *locked*
- Changing between these must be atomic.

Mutexes (cont.)

```
int lock;

void unlock () {
  lock = UNLOCKED;
}

bool lock () {
  if (lock == UNLOCKED) {
    if (CAS(*lock,UNLOCKED,LOCKED))
      return true;
  }

  // ???
  // ? Profit ?
}
```

Mutex Variants

We could ...

- **Classic** Sleep until it becomes available
  - **Error Checking** (or fail if it is locked by us)
  - **Recursive** (or increment a counter if it is locked by us)
- **`timedlock`** Sleep up to a time limite
- **`trylock`** Return false
- **Spin lock** Wait a bit and try again

Syncronisation Problem 1

```
void *stack[LARGE];
int entries = 0;
pthread_mutex_t m;

void push (void *newItem) {
  pthread_mutex_lock(&m);
  stack[entries++] = newItem;
  pthread_mutex_unlock(&m);
}

void pop () {
  pthread_mutex_lock(&m);
  --entries;
  pthread_mutex_unlock(&m);
}
```

```
void * top () {
  void *tmp;

  pthread_mutex_lock(&m);
  if (entries > 0)
    tmp = stack[entries];
  else
    tmp = NULL;
  pthread_mutex_unlock(&m);

  return tmp;
}
```

Syncronisation Problem 2

```
node *head;
pthread_mutex_t m;

void push (void *newItem) {
  node *tmp = createNode(newItem);

  pthread_mutex_lock(&m);
  tmp->next = head;
  head = tmp;
  pthread_mutex_unlock(&m);

}
```

```
void * pop () {
  node *tmp;
  void *r;

  pthread_mutex_lock(&m);
  tmp = head;
  head = tmp->next;
  pthread_mutex_unlock(&m);

  r = tmp->contents;
  destroyNode(tmp);
  return r;
}
```

Semaphores

A generalised version of the mutexes.

- `init(v)` Creates a semaphore with a `count = v`
- `wait()` Wait until `count > 0` and then decrement count
- `post()` Increment count

Read / Write locks

Multiple reads can be done in parallel...

rdlock()	Lock for a read, can be done in parallel
wrlock()	Lock for a write, myst be exclusive
unlock()	Releases either lock

Using read/write locks

```
void *stack[LARGE];
int entries = 0;
pthread_rwlock_t l;

void push (void *newItem) {
  pthread_rwlock_wrlock(&l);
  stack[entries++] = newItem;
  pthread_rwlock_unlock(&l);
}

void pop () {
  pthread_rwlock_wrlock(&l);
  --entries;
  pthread_rwlock_unlock(&l);
}
```

```
void * top () {
  void *tmp;

  pthread_rwlock_rdlock(&l);
  if (entries > 0)
    tmp = stack[entries];
  else
    tmp = NULL;
  pthread_rwlock_unlock(&l);

  return tmp;
}
```

Implementing read/write locks using semaphores

```
int n;   /* The number of threads */
sem_t s;
__thread bool areWriter = true;

sem_init(s,0,n);

rdlock () {
  sem_wait(s);
}

wrlock () {
  for (int i = 0; i < n; ++i) {
    sem_wait(s);
  }
  areWriter = true;
}
```

```
unlock () {
  if (areWriter == false) {
    sem_post(s);
  } else {
    for (int i = 0; i < n; ++i) {
      sem_post(s);
    }
    areWriter = false;
  }
}
```

Implementing read/write locks using semaphores (correct)

```
int n;
sem_t s;
sem_t w;
pthread_t writer;

sem_init(s,n);
sem_init(w,1);

wrlock () {
  sem_wait(w);
  writer = pthread_self();
  for (int i = 0; i < n; ++i) {
    sem_wait(s);
  }
}
```

```
rdlock () {
  sem_wait(s);
}

unlock () {
  if (writer != pthread_self()) {
    sem_post(s);
  } else {
    for (int i = 0; i < n; ++i) {
      sem_post(s);
    }
    sem_post(w);
    writer = 0;
  }
}
```

See also ...

- Condition Variables
- Monitors
- Barriers
- Futexes

Introduction

Threading
- Synchronisation
- Mutexes
- Other Synchronisation Primatives

The Problem with Mutexes...
- Lock Free Algorithms
- STM

Conclusion

Mutexes are correct but ...

1. Can deadlock / livelock
2. Priority inversion
3. Thundering hurd problem
4. Traffic light effect
5. Composition

Lock Free Algorithm

- Reduce the critical section to a single read/write
- Using atomic operations to make this change (assuming nothing has changed)
- Retry if things have changed

Syncronisation Problem 2

```
node *head;

void push (void *newItem) {
  node *tmp = createNode(newItem);

  do {
    tmp->next = head;
  } while (!CAS(*head,head,tmp));
}
```

```
void * pop () {
  node *tmp;
  void *r;

  do {
    tmp = head;
  } while (!CAS(*head,head,tmp->next));

  r = tmp->contents;
  destroyNode(tmp);
  return r;
}
```

Pros and Cons

+ Faster
+ More scalable
+ Can't deadlock, etc.
- Each thread may do more work in the contended case
- The ABA problem
- Ties the design of data structure and synchronisation primatives

Software Transactional Memory

Like subversion for RAM...

Prepair Read from objects without synchronisation, noting their version number, prepair changes.

Commit Lock every touch object, if version numbers have changed abort, else make changes and increment version numbers

Pros and Cons

+ Faster
+ More scalable
+ Composes
- Difficult to use
- Difficult to implement

Introduction

Threading
- Synchronisation
- Mutexes
- Other Synchronisation Primatives

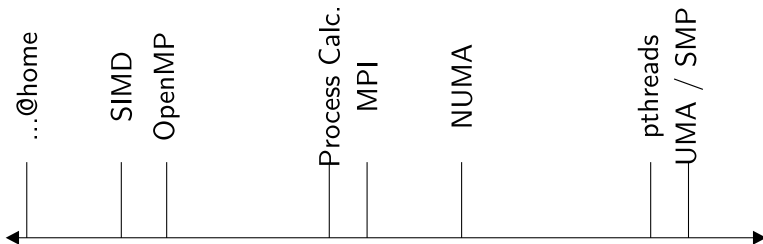
The Problem with Mutexes...
- Lock Free Algorithms
- STM

Conclusion

Classification

- Asynchronous
- Homogenous
- Dynamic
- Reliable
- Scalability - ???
- Granularity - 100s or 1000s of instructions
- Concurrent only on multi-processor machines

Resource Sharing

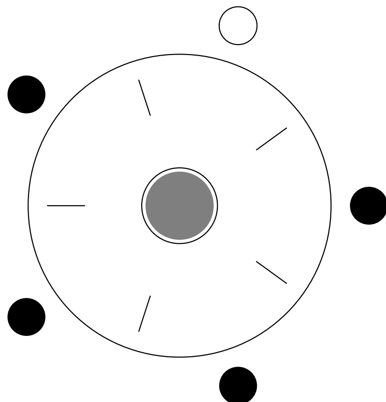


Conclusion

- Shared memory / resource programming is the most flexible but most complicated parallel system
- Key problem is composing atomic operations
- Mutexes give a correct system but not necessarily a fast one

The Dining Philosopher's Problem

... finished

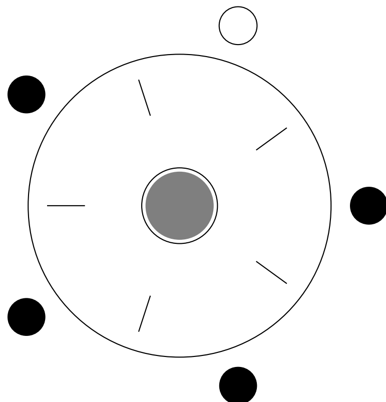


```
0.0 : think
0.1 : hand down | pick up right
0.2 : hand down | pick up left
0.3 : eat
0.4 : hand over | put down left
0.5 : hand over | put down right
0.6 : goto 0.0

1.0 : think
1.1 : hand down | pick up left
1.2 : hand down | pick up right
1.3 : eat
1.4 : hand over | put down right
1.5 : hand over | put down left
1.6 : goto 1.0
```

The Dining Philosopher's Problem

... finished ?



```
0.0 : think
0.1 : hand down | pick up right
0.2 : hand down | pick up left
0.3 : eat
0.4 : hand over | put down left
0.5 : hand over | put down right
0.6 : goto 0.0

1.0 : think
1.1 : hand down | pick up left
1.2 : hand down | pick up right
1.3 : eat
1.4 : hand over | put down right
1.5 : hand over | put down left
1.6 : goto 1.0
```

Questions?

Questions?

Made using only Free Software