Shared-Memory Programming

Jingke Li

Portland State University

Programming Shared Memory Systems

In a shared-memory system, a single address space exists, *i.e.* each memory location is given a unique address, and any memory location is accessible by any of the processors.

Shared-memory systems are typically used in two ways:

- *Executing multiple unrelated programs concurrently* (*aka* "multi-programming")
  - This is a feature of OS, and will not be discussed further.
- *Executing a single application by multiple threads* (*aka* "multi-threading")
  - We'll look into the different approaches for this.

Multi-Threaded Execution

Once multiple threads are created, their execution order will depend upon the system — they may be assigned to different processors, or they may be executed on a single processor in a time-shared fashion. In either case, the statements of individual threads might be *interleaved* in time.

Example:

Thread 1:	A; B; C
Thread 2:	X; Y; Z

Any interleaving execution order of these two statement sequences is a legal execution, *e.g.*

Scenario 1:	A; B; C; X; Y; Z
Scenario 2:	A; B; X; C; Y; Z
Scenario 3:	X; A; Y; B; Z; C
. . .	

Implication: For practical applications, synchronizations are likely needed.

Multi-Threaded Programming

Issues:

- *Thread control and management* — for expressing parallelism.
  - create, terminate, suspend, resume, sleep, wake up
- *Thread synchronization* — for synchronizing shared data access and coordinating computation.
  - locks, semaphores, condition variables, barriers, monitors

Approaches:

1. Explicit control through thread libraries (*e.g.* Pthreads)
2. Implicit control through compiler directives (*e.g.* OpenMP)
3. Implicit control through language support (*e.g.* Algol68, Chapel)

Thread Libraries

Thread libraries provide direct control over all aspects of threads, giving the programmer full programming power.

Example:

- Pthreads — POSIX threads, supported by all versions of Unix/Linux.
- Other libraries exist, *e.g.* Win32 threads, OS/2 Threads, and Java threads.

```
void grandson() {
  printf("grandson\n");
}
void son(int *ip) {
  pthread_t t3;
  pthread_create(&t3, NULL, grandson, NULL);
  printf("son %d\n", *ip);
  pthread_join(t3, NULL);
}
int main() {
  int i=1, j=2;
  pthread_t t1, t2;
  pthread_create(&t1, NULL, son, &i);
  pthread_create(&t2, NULL, son, &j);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Work done\n");
}
```

Compiler Directives

Compiler directives provide a *non-intrusive* approach for shared-memory programming. In this approach, the parallelism information is presented in the form of compiler directives embedded in the user program.

Example:

- OpenMP

```
void Hello() {
  #pragma omp parallel num_threads(4)
  printf("Hello world!\n");
}
```

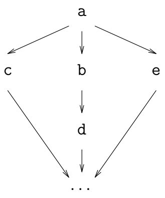
- These directives do not change a program's sequential semantics.
- The information carried by the directives is picked up by special parallelizing compilers.
- The directives can be easily adapted to different host languages.

Language Support

Language support means that the expression of parallelism is through programming languages' constructs.

Example: Algol 68 supports a parallel construct, *parbegin*/*parend*, for creating multiple threads to be executed concurrently.

```
a;
parbegin
  c,
  begin
    b; d
  end,
  e
parend
```

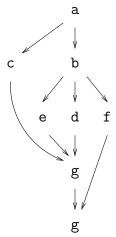


Parbegin/parend can be nested within each other. However, there are restrictions on the parallel threads created by parbegin/parend, for instance, they must share a common stack frame.

Language Support (cont.)

Several languages (e.g. Modular-3) support explicit creation and management of full-scale threads with *fork* and *join* constructs.

```
     a;
     fork L1;
     b;
     fork L2, L3;
     d;
     join L4;
L1:  c; join L4;
L2:  e; join L4;
L3:  f; join L5;
L4:  g; join L5;
L5:  h;
     quit
```



With fork and join, a thread can start at any point and end at any point.

Language Support (cont.)

Chapel has several constructs for supporting thread-based programming:

- The *begin* statement (threads may execute sequentially)

```
begin writeln("hello 1");  // thread 1
begin writeln("hello 2");  // thread 2
writeln("good bye!");      // parent thread (won't wait)
```

- The *cobegin* statement (threads must execute concurrently)

```
cobegin {
  writeln("hello 1");  // thread 1
  writeln("hello 2");  // thread 2
}
writeln("good bye!");  // parent thread (will wait)
```

Thread Synchronizations

Thread synchronizations are needed for two situations:

- *Resolving competition (or achieving mutual exclusion)* —

  When multiple threads try to access the same shared data or execute the same section of code (*i.e.* a critical section), synchronization can ensure that they access the data or enter the critical section one at a time.

- *Facilitating cooperation* —

  When one thread needs the data produced by another thread (*e.g.* a consumer-producer pair), synchronization can provide ways for the producer to inform the consumer that data is ready; and for the consumer to tell the producer that more data is needed.

Locks

Locks are the simplest mechanism for ensuring mutual exclusion of critical sections.

- A lock is just a 1-bit variable — the two values 1 and 0 represent the "locked" and "unlocked" states, respectively.
- A thread can only enter a critical section when lock is unlocked, and it locks the lock before entering.

```
while (lock==1)     // check the lock; if not available,
  wait;             // wait for the lock to open
lock = 1;           // lock; enter critical section
<critical section>
lock = 0;           // leave critical section; unlock
```

Locks (cont.)

Note that the lock-checking statement and the actual-locking statement in the previous code must be implemented as one inseparable *atomic* operation. Otherwise, the lock may fail.

In other words, what we need is

```
Lock(lock);         // atomic operation: check and lock
                    // the lock; wait if not available
<critical section>
Unlock(lock);       // unlock
```

Question: How to implement an atomic operation?

Answer:

1. Use normal loads and stores — complicated and expensive
2. Use special synchronization primitives — simple and efficient

Hardware Synchronization Primitives

Most modern computer architectures provide a variant of the `read-modify-write` (`RMW`) operation as a synchronization primitive:

- `get-and-set(v)` — read and return the old value, and set the new value `v`
- `get-and-increment()` — read and return the old value, and set the incremented value
- `test-and-set(v)` — test the old value, and set the new value `v` if the old value passes the test; return a boolean
- `load-linked()/store-conditional()` — like `test-and-set()`, but with two separate read and write instructions; still guarantees atomicity

The common theme: A non-interruptible instruction or instruction sequence capable of atomically retrieving and changing a value.

Common Types of Locks

Based on strategies for handling waiting, locks can be categorized as:

- *Spin Locks:*
  - If the lock is locked, the thread repeatedly tests the lock status, until it becomes unlocked.
- *Blocking Locks:*
  - If the lock is locked, the requesting thread gets blocked and goes into sleep. (This strategy requires a waking-up mechanism.)
- *Combination:*
  - If the lock is locked, the thread spins for a specified period, then block. (This is useful if the thread knows that the thread holding the lock is about to release it.)

There are also

- *Readers/Writer Locks:*
  - Multiple threads can *concurrently read* the data protected by an RW lock; but any thread to write to the data must have exclusive access.

Support User-Level Spin Locks

Supporting user-level spin locks is not a trivial task.

First try: Using the synchronization primitives directly at the user level.

```
void Lock(boolean lock) {
  while (get-and-set(lock,true)) {}
}
void Unlock(boolean lock) {
  set(lock, false);
}
```

Problem:

- Excessive memory accesses. Each `get-and-set()` call translates to a memory read and a memory write. When number of competing threads increases, the situation worsens quickly.
- It can take exponential (or more) number of memory accesses for n competing threads to each get the lock.

Support Spin Locks, Take Two

Taking advantage of the cache coherence mechanism.

```
void Lock(boolean lock) {
  while (true) {
    while (get(lock)) {}         // spin on a cache copy
    if (!get-and-set(lock,true)) // one memory access
      return;
  }
}
```

Remaining Problem:

- When the lock is released, every waiting thread will issue a `get-and-set()` call, even though only one will succeed.
- It can take $O(n^2)$ number of memory accesses for n competing threads to each get the lock.

Support Spin Locks, Take Three

Using the "exponential back-off" strategy — if a `get-and-set()` call fails, double the waiting time.

```
void Lock(boolean lock) {
  int delta = <a small-random-init-value>;
  while (true) {
    while (get(lock)) {}         // spin on a cache copy
    pause delta;
    if (!get-and-set(lock,true)) // one memory access
      return;
    delta = delta * 2;
  }
}
```

Support Spin Locks, Take Four

Using a queue to organize the waiting threads; letting only one thread to call `get-and-set()` when the lock is released.

The queue can be implemented either in software or in hardware.

- In software — with the use of an array
- In hardware — with queuing functionality added to directory controller on directory-based multiple processors

Deadlock

- Deadlock can occur with two threads when each requires a resource held by the other. For example,

  Thread 1: requests A; requests B; uses A and B
  Thread 2: requests B; requests A; uses A and B

  Deadlock occurs when thread 1 holds A, thread 2 holds B, and each wants to get another resource.
- Deadlock can also occur in a circular fashion with several threads:

  Thread 1: holds A_1; requests A_2
  Thread 2: holds A_2; requests A_3
  ...
  Thread k: holds A_k; requests A_1
- Deadlock can be avoid if all threads request resources in the same order.

Semaphores

Semaphores are an extension to locks. A semaphore, s, is a non-negative integer operated upon by two operations:

- $P(s)$ — waits until s is > 0 and then decrements s by one and allows the thread to continue.
- $V(s)$ — increments s by one to release one of the waiting threads (if any).

The P and V operations are performed atomically. A mechanism for activating waiting threads is also implicit in the operations.

Threads delayed by $P(s)$ are kept in sleep until released by a $V(s)$ on the same semaphore.

A Semaphore Example

Semaphore can be used to solve the producer-consumer problem:

```
// a global task queue
queue_t *queue = ...;

void producer() {
  task_t *task;
  while (true) {
    task = create_task();
    add_task(queue, task);
    V(queue.len);
  }
}
```

```
void consumer() {
  task_t *task;
  while (true) {
    P(queue.len);
    task = remove_task(queue);
    compute(task);
  }
}
```

Condition Variables

Condition variables are a further extension to semaphores.

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

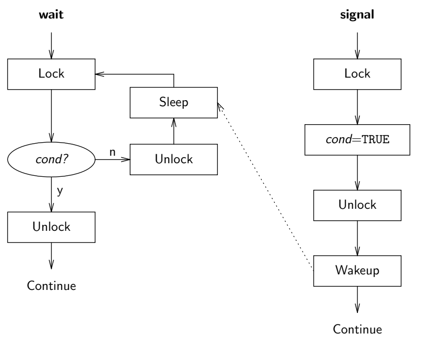
- With locks, the global variable would need to be examined at frequent intervals ("polled") within the critical section. This is a very time-consuming and unproductive exercise.
- Semaphores, on the other hand, can only handle a limited form of conditions, *e.g.* a counter reaching 0.

This problem can be overcome by using condition variables.

Three operations are defined for a condition variable:

`Wait(cond)` — wait for a condition to occur
`Signal(cond)` — signal that the condition has occurred
`Status(cond)` — return the number of waiting threads

A Graphic View of Condition Variables



Condition Variable Example

Consider one or more threads designed to take action whenever a counter, x, reaches a value that is a multiple of 5. (A separate thread is responsible for incrementing the counter.)

```
void action() {
  lock();
  while (x % 5 == 0)
    wait(s);
  unlock();
  take_action();
}
```

```
void counter() {
  lock();
  x++;
  if (x % 5 == 0)
    signal(s);
  unlock();
}
```

- The `wait` operation will automatically release the lock, to allow another thread to alter the condition.
- The `signal` operation will automatically wake up any or all waiting threads (implementation dependent).
- The waking-up threads will try to lock the lock again, and one of them will succeed.

Monitors

Monitors are introduced for providing object-oriented style of access control — Essentially the protected data and the operations (*monitor procedures*) that can operate upon it are encapsulated inside a structure. Reading and writing to the data can only be done by using monitor procedures, and only one thread can use a monitor procedure at any instant.

A monitor procedure could be implemented using a semaphore to protect its entry; i.e.,

```
void monitor_procedure() {
  P(monitor_semaphore);
  <monitor body>
  V(monitor_semaphore);
  return;
}
```

Monitor Example

The concept of monitor exists in Java. The keyword `synchronized` in Java makes a method thread safe, preventing more than one thread executing the method at the same time.

```
public class Adder {
  public int[] array;
  private int sum=0, i=0;
  public Adder() { ... } // constructor

  public synchronized int nextIndex() {
    return (i < 1000) ? i++ : -1;
  }

  public synchronized void addSum(int psum) {
    sum += psum;
  }
}
```

Monitor Example (cont.)

```
class AdderThread extends Thread {
  Adder adder;
  public AdderThread(Adder adder) { this.adder = adder; }

  public void run() {
    int i, psum;
    while ((i=adder.nextIndex()) != -1)
      psum = psum + adder.array[i];
    adder.addSum(psum);
  }
}

public static class Driver {
  public static void main(String args[]) {
    Adder adder = new Adder();
    for (int i=0; i<10; i++)
      new AdderThread(adder, i).run();
  }
}
```

Desirable Synchronization Properties

The above synchronization mechanisms provide basic access control on shared data. For many applications, stronger policies are required on top of them, such as

- *Safe* — A safe policy is one that enforces deterministic results, i.e. access order is always the same no matter how many times the program run.
- *Fair* — A fair policy guarantees access to all tasks that have requested access, before second or third accesses are granted to any pending tasks.
- *Deadlock-Free* — A deadlock-free policy guarantees that no deadlock can ever happen no matter what order or speed the requests are generated.