Programming for Multi-Core CPUs: Locking and Transactional Memory

Huiyang Zhou

Slides 1-18 are from Professor Umakishore Ramachandran @ GaTech

Example use of threads - 1

(a) Sequential process
(b) Multithreaded process
Example use of threads - 2

Programming Support for Threads

- creation
  - `pthread_create`(top-level procedure, args)
- termination
  - `return` from top-level procedure
  - explicit `kill`
- rendezvous
  - creator can `wait` for children
    - `pthread_join`(child_tid)
- synchronization
  - `mutex`
  - `condition` variables

<table>
<thead>
<tr>
<th>Main thread</th>
<th>main thread</th>
<th>foo thread</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>thread_create(foo, args)</code></td>
<td><code>thread_create(foo, args)</code></td>
<td><code>thread_create(foo, args)</code></td>
</tr>
</tbody>
</table>

(a) Before thread creation  (b) After thread creation
Sample program – thread create/join

```c
int foo(int n)
{
    ....
    return 0;
}

int main()
{
    int f;
    thread_type child_tid;
    ....
    child_tid = thread_create (foo, &f);
    ....
    thread_join(child_tid);
}
```

Programming with Threads

- synchronization
  for coordination of the threads
- communication
  for inter-thread sharing of data
  threads can be in different processors
  how to achieve sharing in SMP?
  - software: accomplished by keeping all threads in the same address space by the OS
  - hardware: accomplished by hardware shared memory and coherent caches
Need for Synchronization

digitizer()
{
    image_type dig_image;
    int tail = 0;
    loop {
        if (bufavail > 0) {
            grab(dig_image);
            frame_buf[tail mod MAX] = dig_image;
            tail = tail + 1;
            bufavail = bufavail - 1;
        }
    }
}

tracker()
{
    image_type track_image;
    int head = 0;
    loop {
        if (bufavail < MAX) {
            track_image = frame_buf[head mod MAX];
            head = head + 1;
            bufavail = bufavail + 1;
            analyze(track_image);
        }
    }
}

Problem?
Synchronization Primitives

- lock and unlock
  - mutual exclusion among threads
  - busy-waiting Vs. blocking
  - `pthread_mutex_trylock`: no blocking
  - `pthread_mutex_lock`: blocking
  - `pthread_mutex_unlock`

Fix number 1 – with locks

digitizer()
{
    image_type dig_image;
    int tail = 0;
    loop {
        thread_mutex_lock(buflock);
        if (bufavail > 0) {
            grab(dig_image);
            frame_buf[tail mod MAX] = dig_image;
            tail = tail + 1;
            bufavail = bufavail - 1;
        }
        thread_mutex_unlock(buflock);
    }
}

tracker()
{
    image_type track_image;
    int head = 0;
    loop {
        thread_mutex_lock(buflock);
        if (bufavail < MAX) {
            track_image = frame_buf[head mod MAX];
            head = head + 1;
            bufavail = bufavail + 1;
            analyze(track_image);
        }
        thread_mutex_unlock(buflock);
    }
}

Problem?
**Fix number 2**

```c
digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop {
        grab(dig_image);
        while (bufavail == 0) do nothing;
        frame_buf[tail mod MAX] = dig_image;
        tail = tail + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_mutex_unlock(buflock);
    }
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop {
        thread_mutex_lock(buflock);
        while (bufavail == MAX) do nothing;
        track_image = frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_mutex_unlock(buflock);
        analyze(track_image);
    }
}
```

Problem?

---

**Fix number 3**

```c
digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop {
        grab(dig_image);
        while (bufavail == 0) do nothing;
        frame_buf[tail mod MAX] = dig_image;
        tail = tail + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_mutex_unlock(buflock);
    }
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop {
        while (bufavail == MAX) do nothing;
        track_image = frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_mutex_unlock(buflock);
        analyze(track_image);
    }
}
```

Problem?
• condition variables
  - `pthread_cond_wait`: block for a signal
  - `pthread_cond_signal`: signal one waiting thread
  - `pthread_cond_broadcast`: signal all waiting threads

---

**Wait and signal with cond vars**

(a) Wait before signal

(b) Wait after signal (T1 blocked forever)
Fix number 4 – cond var

digitizer()
{
    image_type dig_image;
    int tail = 0;
    loop |
        grab(dig_image);
        thread_mutex_lock(buflock);
        if (bufavail == 0)
            thread_cond_wait(buf_not_full, buflock);
        thread_mutex_unlock(buflock);
        frame_buf[tail mod MAX] = dig_image;
        tail = tail + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_cond_signal(buf_not_empty);
        thread_mutex_unlock(buflock);
    }
}

tracker()
{
    image_type track_image;
    int head = 0;
    loop |
        thread_mutex_lock(buflock);
        if (bufavail == MAX)
            thread_cond_wait(buf_not_empty, buflock);
        thread_mutex_unlock(buflock);
        track_image = frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_cond_signal(buf_not_full);
        thread_mutex_unlock(buflock);
        analyze(track_image);
    }
}

This solution is correct so long as there is exactly one producer and one consumer

Gotchas in programming with cond vars

acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex);
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy);
    thread_mutex_unlock(cs_mutex);
}
State of waiting queues

(a) Waiting queues before T1 signals

<table>
<thead>
<tr>
<th>cs_mutex</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>res_not_busy</td>
<td>T2</td>
</tr>
</tbody>
</table>

(a) Waiting queues after T1 signals

<table>
<thead>
<tr>
<th>cs_mutex</th>
<th>T3</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>res_not_busy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Gotchas in programming with cond vars

```c
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex);
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}
```

T2 is here (get the lock)

```c
release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy);
    thread_mutex_unlock(cs_mutex);
}
```

T3 is here (release the lock)

T1 is here

Both T2 and T3 access the shared resource (which was supposed to be accessed exclusively)
Defensive programming – retest predicate

acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex); ← T3 is here
    while (res_state == BUSY)
        thread_cond_wait (res_not_busy, cs_mutex); ← T2 is here
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_buys);
    thread_mutex_unlock(cs_mutex);
} ← T1 is here

T3 is here (release the lock)
Transactional Memory

- Borrow the ‘transaction’ idea from database systems

- Atomic region
  - All commit or none commit
  - Read set / write set conflict detection
  - Rollback when conflict happens
  - Software implementation using locks
  - Hardware implementation with specialize cache designs

---

Lock-based code

digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    thread_mutex_lock(buflock);
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail mod MAX] = dig_image;
      tail = tail + 1;
      bufavail = bufavail - 1;
    }
    thread_mutex_unlock(buflock);
  }
}

tracker()
{
  image_type track_image;
  int head = 0;
  loop {
    thread_mutex_lock(buflock);
    if (bufavail < MAX) {
      track_image = frame_buf[head mod MAX];
      head = head + 1;
      bufavail = bufavail + 1;
      analyze(track_image);
    }
    thread_mutex_unlock(buflock);
  }
}

Problem?
Transactionals memory programs

digitizer()
{
    image_type dig_image;
    int tail = 0;
    loop {
        atomic {
            if (bufavail > 0) {
                grab(dig_image);
                frame_buf[tail mod MAX] = dig_image;
                tail = tail + 1;
                bufavail = bufavail - 1;
            }
        } // end of atomic region
    }
}

tracker()
{
    image_type track_image;
    int head = 0;
    loop {
        atomic {
            if (bufavail < MAX) {
                track_image = frame_buf[head mod MAX];
                head = head + 1;
                bufavail = bufavail + 1;
                analyze(track_image);
            }
        } // end of atomic region
    }
}
Challenges of TM

- I/O in atomic regions
- Nested atomic regions
- Atomic regions conflicting with code in non-atomic regions
- Etc.

- Promising research area with significant challenges