MEMORY SYNCHRONIZATION

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Upcoming deadline
  - Feb. 8th: project proposal
    - Apply my suggestions to your pre-proposal and resubmit
  - 5 pre-proposal received; one group is missing

- This lecture
  - What cache coherence is unable to do
    - Shared memory synchronizations
    - Locks
    - Barriers
    - Transactional memory
Recall: Cache Coherence

- Coherency protocols (must) guarantee
  - write propagation
  - write serialization

- Coherency protocols do not guarantee
  - only one thread accesses shared data
  - threads start executing a section of code together

How to synchronize threads?
Example

```c
int mem[]; // large array
...
main() {
    ...
    for(i=0; i<N; ++i) {
        sum += mem[i];
    }
    avg = sum / N;
    ...
}
```
Shared Memory Synchronization

- Critical section problem
  - How to order thread access to shared data?

- Memory barriers
  - Force threads to start executing a section together

```
P1 ... Pn
...
X ← X+1;
...

P1 ... Pn
X ← X+1;
...
...
...
Y ← X+Y;
```
Synchronization Components

- **Acquire method**
  - obtain the lock or proceed past the barrier

- **Waiting algorithm**
  - *spin (busy wait)*
    - Repeatedly test a condition; additional traffic
  - *block (suspend)*
    - Let OS suspend the process; large resume overheads

- **Release method**
  - allow other processes to proceed
Critical Section Problem

- **Definition**
  - N threads compete to use some shared data
  - Each process has a code segment, called critical section, in which the shared data is accessed

- **Need to provide**
  - *Mutual exclusion*: no two threads are allowed in the critical section
  - *Forward progress*: no one outside the critical section may block other processes
  - *Fairness*: bounded waiting times for entering the critical section
Basic Hardware for Synchronization

- Test-and-set — atomic exchange
- Fetch-and-op (e.g., increment)
  - returns value and atomically performs op (e.g., increments it)
- Compare-and-swap
  - compares the contents of two locations and swaps if identical
- Load-linked/store conditional
  - pair of instructions — deduce atomicity if second instruction returns correct value
Lock Example

- Test-and-set spin lock (TSL)

entry_section:
- TSL R1, LOCK  | copy lock to R1 and set lock to 1
- CMP R1, #0    | was lock zero?
- JNE entry_section | if it wasn’t zero, lock was set, so loop
- RET          | return; critical section entered

exit_section:
- MOV LOCK, #0  | store 0 into lock
- RET          | return; out of critical section

Problem: many memory reads and writes due to busy waiting
Question: what if a process is switched out of CPU during CS?
Lock Example

- Test-and-Test-and-set spin lock (TTSL)
  - Spinning on read only data (local copy)

```assembly
entry_section:
    MOV R1, LOCK | copy lock to R1
    CMP R1, #0   | if it was zero
    JNE entry_section | if it wasn’t zero, loop
    TSL R1, LOCK  | copy lock to R1 and set lock to 1
    CMP R1, #0   | was lock zero?
    JNE entry_section | if it wasn’t zero, lock was set, so loop
    RET          | return; critical section entered
```

- Excessive memory traffic due to multiple cores spinning on a lock
- TTSL is unfair
Lock Example

- Ticket lock using fetch-and-op (increment)

```c
lock:
myticket = fetch & increment (&(L->next_ticket));
while(myticket!=L->now_serving) {
    delay(time * (myticket-L->now_serving));
}
unlock:
L->now_serving = L->now_serving+1;
```

- Advantage : Fair (FIFO)

- Disadvantage : Contention (Memory/Network)
Lock Example

- **MCS linked-list based queue locks**
  - Processors waiting on the lock are stored in a linked list
  - Every processor using the lock allocates a queue node `(I)` with two fields
    - `must_wait (bool)` and `next_node (pointer)`
- **Lock variable is a pointer to the tail of the queue**

```c
acquire(lock):
  I->next = null;
  predecessor = Swap(lock, I)
  if predecessor != NULL
      I->must_wait = true
      predecessor->next = I
  repeat while I->must_wait
```

How to release MCS lock?
Lock Example

- Release MCS lock

```c
release(lock):
    if (I->next == null)
        if CAS(lock, I, null)
            return
    I->next->must_wait = false
```
Centralized Barrier

- A globally-shared piece of state keeps track of thread arrivals
  - e.g., a counter
- Each of the threads
  - updates shared state to indicate its arrival
  - polls that state and waits until all threads have arrived
- Then, it can leave the barrier
- Since barrier has to be used repeatedly:
  - state must end as it started
Key idea: decouple spinning from the counter

```c
// global variables
int count = P;
bool sense = true;

// local variable
bool local_sense = true;

// barrier
local_sense = !local_sense;
if(fetch_and_dec(&count) == 1) {
    count = P;
    sense = local_sense;
}
else {
    while(sense != local_sense);
}
```
Lock Freedom

- Priority inversion: a low-priority process is preempted while holding a lock needed by a high-priority process

- Convoying: a process holding a lock is de-scheduled (e.g. page fault, no more quantum), no forward progress for other processes capable of running

- Deadlock (or Livelock): processes attempt to lock the same set of objects in different orders (could be bugs by programmers)

- Error-prone
A sequence of instructions that is guaranteed to execute and complete only as an atomic unit

Begin Transaction

Inst #1
Inst #2
Inst #3
...

End Transaction

Satisfy the following properties

- Serializability: Transactions appear to execute serially.
- Atomicity (or Failure-Atomicity): A transaction either
  - commits changes when complete, visible to all; or
  - aborts, discarding changes (will retry again)