Shared Memory Synchronization

Goal

- Coordinate sharing among all threads
  - mutually exclusive access to shared data
- Coordinate pairwise sharing
  - Producer / Consumer 같은 Problem!

Synchronization

- locks : only one thread can enter critical section
- barriers : no thread can execute code until all iterations have finished

Shared Memory

- limitation
  : Multiple writers of variable ~ unpredictable values
- solution : atomic operation

Synchronization 없음!

Shared Memory Synchronization

lock 구현하는 방법

Test and Set

- test_and_set (word &M)
  - writes 1 to M
  - return M's Previous value

```
type Lock = (unlocked, locked)

procedure acquire_lock(Lock *L)
  loop
    // NOTE: test and set returns old value
    if test_and_set(L) == unlocked
       return

procedure release_lock(Lock *L)
  *L = unlocked
```

acquire test 할때마다 lock에 write 발생함.

- Shared Memory system에서는 local cache에 write 함에따라 Bus transaction이 발생하므로 communication overhead ↑↑↑

Compare and Swap

- Compare_and_Swap (word &M, word oldV, word newV)
  - if (M == oldV) M ← newV
  - returns true if store was performed

```
type Lock = (unlocked, locked)

procedure acquire_lock(Lock *L)
  loop
    // NOTE: test and set returns old value
    if compare_and_swap(L,unlocked,locked) == TRUE
       return

procedure release_lock(Lock *L)
  *L = unlocked
```

Why better?

레지스터의 경우에 (M,S)는 read request는 transaction X 그냥 성공.

Bus transaction 횟수 줄일 수 있음.

Atomic Primitives

- test_and_set(Word &M)
  - —writes a 1 into M
  - —returns M's previous value
- swap(Word &M, Word V)
  - —replaces the contents of M with V
  - —returns M's previous value
- compare_and_swap(Word &M, Word oldV, Word newV)
  - —if (M == oldV) M ← newV
  - —returns TRUE if store was performed
- fetch_and_Φ(Word &M, Word V)
  - —Φ can be ADD, OR, XOR, ...
  - —replaces the value of M with Φ(old value, V)
  - —returns M's previous value

Shared Memory Synchronization

barrier 한번쉬어가기

Design a Simple Barrier

- each Processor indicates its arrival at the barrier
  - updates shared state
- busy-wait on shared state to determine when all have arrived
- Once all have arrived, each processor is allowed to continue

Sense-switching centralized barrier

```
integer count = P
bool sense = true

thread_local bool local_sense = true

void central_barrier() {
  // each processor toggles its own sense
  local_sense = not local_sense
  if (fetch_and_add(&count,-1) == 1)
    count = P
    sense = local_sense  // last processor toggles global sense
  else
    repeat until sense == local_sense
}
```

this thread is last thread in execution

계속 overwrite 되는 code에서 compiler가 다른짓을 할 수도 있다.

count를 레지스터로 update하기 때문에

count가 invalidate 될수도 있다

General Consideration

- Adjusting state of synchronization
  - reinitialize 코드 ~ sense switching 사용해서
- Interconnect traffic & Contention
  - spin-waiting을 사용할때 주의해야함.

Avoid Spin Waiting over Interconnect

Avoid Spin Waiting over the Interconnect

- **How?**
  - —don't have multiple threads spin wait on a shared variable that will change multiple times per synchronization operation
- **For instance**
  - —avoid spin waiting on
    - a barrier count that others are adjusting with atomic_add
      - use a barrier flag (e.g., sense) instead
    - a lock variable that others will toggle with test and set
      - use a compare and swap (no excessive broadcasting)
      - use a link-list-based lock (local spinning)
        - e.g. MCS lock