Synchronization (cont.)

Kai Shen

2/13/2014  CSC 258/458 - Spring 2014  1

Synchronization and Scheduling

- Busy-waiting synchronization
  - Waste CPU on waiting
  - OK if each process/thread has a CPU exclusively
  - What if there are fewer CPUs than processes/threads?

- Blocking (yielding CPU) synchronization
  - Yield the CPU (so other process/thread can make good use) if we must wait
  - Need operating system involvement

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable which can only be accessed via two atomic operations:
  
  wait(S) or P(S):
  - wait until S>0;
  - S--;

  signal(S) or V(S):
  - S++;

  Can you show mutual exclusion?
  Can you show deadlock-free?

Semaphore Implementation

- Define a semaphore as a record
  
  typedef struct {
    int value;
    semaphore mutex=1;
    proc_list *L;
  } semaphore;

- Assume two OS operations:
  - block suspends the process that invokes it.
  - wakeup(P) resumes the execution of blocked process P.

  Semaphore operations now defined as (both are atomic):
  
  wait(S):
  - S.value--;
  - if (S.value < 0) {
    add this process to S.L;
    block;
  }

  signal(S):
  - S.value++;
  - if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
  }

  How to make sure wait(S) and signal(S) are atomic?
  So have we truly removed busy waiting?
Mutex Lock (Binary Semaphore)

- Mutex lock – a semaphore with only two states: locked/unlocked
- Semantics of the two (atomic) operations:
  
  ```
  lock(mutex):
  wait until mutex==unlocked;
  mutex=locked;
  ```
  
  ```
  unlock(mutex):
  mutex=unlocked;
  ```
- Can you implement mutex lock using semaphore?
- How about the opposite?

Implement Semaphore Using Mutex Lock

- Data structures:
  ```
  mutex_lock L1, L2;
  int C;
  ```
- Initialization:
  ```
  L1 = unlocked; L2 = locked;
  C = initial value of semaphore;
  ```
- wait operation:
  ```
  lock(L1);
  C --;
  if (C < 0) {
    unlock(L1); lock(L2);
  }
  unlock(L1);
  ```
- signal operation:
  ```
  lock(L1);
  C ++;
  if (C <= 0)
    unlock(L2);
  else
    unlock(L1);
  ```

Busy-Wait vs. Blocking Synchronization

- Busy-wait synchronization: software/hardware spin locks
- Blocking synchronization: semaphore, mutex lock, condition variable, ...
- When each process/thread has its dedicated CPU
  - Is busy waiting OK?
- When only need to protect a short (bounded size) critical section
  - Is busy waiting OK?
  - Still has the risk of wasting substantial CPU time in waiting, if context switched out in the middle of critical section
- For complex synchronization of unpredictable waiting time
  - Is busy waiting OK?
  - Higher overhead (typically done in the OS, may involve context switch), but no risk of wasting substantial CPU time in waiting

Busy-Wait vs. Blocking Synchronization

- Benefit of blocking:
  - Useful when the waiting time is long
- Cost of blocking:
  - Context switch overhead (cache warmup cost)
- Application does not make the choice, but rather leave it to the OS
  - When a process/thread must wait for synchronization from some other process, should it spin (busy-wait) or block?
- What if you know the waiting time?
  - Spin the waiting time is shorter than the context switch cost; block otherwise
- What if you don’t know the waiting time?
  - Spin for the time equal to the context-switch overhead. If not successful, then block.
Dining-Philosophers Problem

- Philosopher $i \ (1 \leq i \leq 5)$:
  
  ```
  while (1) {
    ... 
    eat;
    ... 
    think;
    ... 
  }
  ```

Eating needs both chopsticks (the left and the right one).

Dining-Philosophers Solution

- Shared data:
  ```
  mutex chopstick[5];
  ```

- Philosopher $i$:
  ```
  while(1) {
    ...
    lock(chopstick[i]);
    lock(chopstick[(i+1) % 5]);
    eat;
    unlock(chopstick[i]);
    unlock(chopstick[(i+1) % 5]);
    ...
    think;
    ...
  }
  ```