Synchronization, Critical section

Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
- **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task
- **Critical Section**: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Locks: using interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    Place on ready queue;
  } else {
    value = FREE;
  }
  enable interrupts;
}
```
Better locks: using test&set

- `test&set (address) { /* most architectures */
  result = M[address];
  M[address] = 1;
  return result;
}

int guard = 0;
int value = FREE;

Acquire() {
  // Short busy-wait time
  while (test&set(guard));
  if (value == BUSY) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = BUSY;
    guard = 0;
  }
}

Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue;
    Place on ready queue;
  } else {
    value = FREE;
  }
  guard = 0;
}

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - `P()`: an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the `wait()` operation
  - `V()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the `signal()` operation
  - Note that `P()` stands for “proberen” (to test) and `V()` stands for “verhogen” (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except to set it initially
  - Operations must be atomic
    - Two P’s together can’t decrement value below zero
    - Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

Condition Variables

- Condition Variable: a queue of threads waiting for something **inside** a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section

- Operations:
  - `Wait(lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

- Rule: Must hold lock when doing condition variable ops!
Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue
  Lock lock;
  Condition dataready;
  Queue queue;
  AddToQueue(item) {
      lock.Acquire(); // Get Lock
      queue.enqueue(item); // Add item
      dataready.signal(); // Signal any waiters
      lock.Release(); // Release Lock
  }
  RemoveFromQueue() {
      lock.Acquire(); // Get Lock
      while (queue.isEmpty()) {
          dataready.wait(&lock); // If nothing, sleep
      }
      item = queue.dequeue(); // Get next item
      lock.Release(); // Release Lock
      return(item);
  }

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:
  while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
- Why didn’t we do this?
  if (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
- Answer: depends on the type of scheduling
  - Hoare-style (most textbooks):
    » Signaler gives lock, CPU to waiter; waiter runs immediately
    » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - Mesa-style (most real operating systems):
    » Signaler keeps lock and processor
    » Waiter placed on ready queue with no special priority
    » Practically, need to check condition again after wait

Read/Writer Revisited

Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
    // read-only access
    AccessDatabase(ReadOnly);
    // check out of system
    lock.Acquire();
    if (AR > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}

Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
    // read/write access
    AccessDatabase(ReadWrite);
    // check out of system
    lock.Acquire();
    if (WW > 0) {
        okToWrite.broadcast();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
Read/Writer Revisited

Reader() {
  // check into system
  lock.Acquire();
  while ((AW + WW) > 0) {
    WR++;  
    okContinue.wait(&lock);  
    WR--;  
  }  
  AR++;  
  lock.release();  
  // read-only access  
  AccessDbase(ReadOnly);  
  // check out of system  
  lock.Acquire();  
  AR--;  
  if (AR == 0 && WW > 0) {  
    okContinue.signal();  
    lock.Release();  
  }
}

Writer() {
  // check into system
  lock.Acquire();
  while ((AW + AR) > 0) {
    WW++;  
    okContinue.wait(&lock);  
    WW--;  
  }  
  AW++;  
  lock.release();  
  // read/write access  
  AccessDbase(ReadWrite);  
  // check out of system  
  lock.Acquire();  
  AW--;  
  if (WW > 0) {  
    okToWrite.signal();  
  } else if (WR > 0) {  
    okContinue.broadcast();  
    lock.Release();  
  }
}

Need to change to broadcast!
Why?

What if we turn okToWrite and okToRead into okContinue?

• R1 arrives  
• W1, R2 arrive while R1 reads  
• R1 signals R2  

Deadlock
**Four requirements for Deadlock**

- **Mutual exclusion**
  - Only one thread at a time can use a resource.

- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads.

- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it.

- **Circular wait**
  - There exists a set \( \{ T_1, \ldots, T_n \} \) of waiting threads.
    - \( T_1 \) is waiting for a resource that is held by \( T_2 \).
    - \( T_2 \) is waiting for a resource that is held by \( T_3 \).
    - \( \ldots \)
    - \( T_n \) is waiting for a resource that is held by \( T_1 \).

**Resource Allocation Graph Examples**

- **Simple Resource Allocation Graph**
  - \( T_1 \)
  - \( T_2 \)
  - \( T_3 \)
  - \( R_1 \)
  - \( R_2 \)
  - \( R_3 \)
  - \( R_4 \)

- **Allocation Graph With Deadlock**
  - \( T_1 \)
  - \( T_2 \)
  - \( T_3 \)
  - \( T_4 \)
  - \( R_1 \)
  - \( R_2 \)
  - \( R_3 \)
  - \( R_4 \)

- **Allocation Graph With Cycle, but No Deadlock**
  - \( T_1 \)
  - \( T_2 \)
  - \( T_3 \)
  - \( T_4 \)

**Deadlock Detection Algorithm**

- Only one of each type of resource \( \Rightarrow \) look for loops.

- More General Deadlock Detection Algorithm

  - Let \( [X] \) represent an \( m \)-ary vector of non-negative integers (quantities of resources of each type):
    - \( [\text{FreeResources}] \): Current free resources each type
    - \( [\text{Request}_X] \): Current requests from thread \( X \)
    - \( [\text{Alloc}_X] \): Current resources held by thread \( X \)

  - See if tasks can eventually terminate on their own.

    \[ [\text{Avail}] = [\text{FreeResources}] \]
    \[ \text{Add all nodes to UNFINISHED} \]
    \[ \text{do} \]
    \[ \text{done = true} \]
    \[ \text{foreach node in UNFINISHED} \]
    \[ \text{if} \ ( [\text{Request}_{\text{node}}] \leq [\text{Avail}] ) \]
    \[ \text{remove node from UNFINISHED} \]
    \[ [\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{node}}] \]
    \[ \text{done = false} \]
    \[ \text{until} ( \text{done} ) \]
    \[ \text{Nodes left in UNFINISHED} \Rightarrow \text{deadlocked} \]

**Banker’s Algorithm for Preventing Deadlock**

- Toward right idea:
  - State maximum resource needs in advance.
  - Allow particular thread to proceed if:

    \[ ( \text{available resources} - \#\text{requested} ) \geq \text{max remaining that might be needed by any thread} \]

- Banker’s algorithm (less conservative):

  - Allocate resources dynamically.
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward.
    - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting \( ([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \) for \( ([\text{Request}_{\text{node}}] \leq [\text{Avail}]) \).
    - Grant request if result is deadlock free (conservative!).
    - Keeps system in a “SAFE” state, i.e. there exists a sequence \( (T_1, T_2, \ldots, T_n) \) with \( T_1 \) requesting all remaining resources, finishing, then \( T_2 \) requesting all remaining resources, etc.,

  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources.
Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - Can be used to avoid overlap
    - Can be used to give uniform view of memory to programs

Why Address Translation?

Addr. Translation: Segmentation vs. Paging
Review: Address Segmentation

Virtual memory view

Physical memory view

Stack Seg # base limit
00 0001 0000 10 0000
01 0101 0000 10 0000
10 0111 0000 11 1000
11 1011 0000 1 0000

Heap

Data

Code

Offset

Virtual memory view

Physical memory view

Stack Seg # base limit
00 0001 0000 10 0000
01 0101 0000 10 0000
10 0111 0000 11 1000
11 1011 0000 1 0000

Heap

Data

Code

Offset

Page Table

Physical memory view

Review: Paging

Virtual memory view

Page Table

Physical memory view

Stack 1110 0000

Heap 1111 0000

Data 1111 0000

Code 1110 0000

No room to grow!! Buffer overflow error

What happens if stack grows to 1110 0000?

Page 7
What happens if stack grows to 1110 0000?

Virtual memory view

Physical memory view

Review: Two-Level Paging

Page Tables (level 1)

Page Tables (level 2)
Address Translation Comparison

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation</td>
<td>• Large size: Table size ~ virtual memory</td>
</tr>
<tr>
<td>Paged segmentation</td>
<td>• No external fragmentation</td>
<td>• Internal fragmentation</td>
</tr>
<tr>
<td>Two-level pages</td>
<td>• Table size ~ memory used by program</td>
<td>• Multiple memory references per page access</td>
</tr>
<tr>
<td>Inverted Table</td>
<td></td>
<td>• Internal fragmentation</td>
</tr>
</tbody>
</table>

Review: Sources of Cache Misses

- **Compulsory** (cold start): first reference to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: When running “billions” of instruction, Compulsory Misses are insignificant
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Conflict** (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- **Two others**:
  - **Coherence** (Invalidation): other process (e.g., I/O) updates memory
  - **Policy**: Due to non-optimal replacement policy

Caches, TLBs
Direct Mapped Cache

- Cache index selects a cache block
- “Byte select” selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same “cache tag” shares the same cache entry
  - Conflict misses

Example:

- Cache Data
  - Byte 0
  - Byte 32
  - Byte 63
- Valid Bit
- Cache Tag
- Example: 0x01
- Cache Index
- Byte Select
- Compare
- Hit

Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operate in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

Example:

- Cache Data
  - Cache Block 0
  - Cache Block 0
- Cache Tag
- Valid
- Cache Index
- Byte Select
- Compare
- OR
- Hit
- Cache Block

Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

Example:

- Cache Tag (27 bits long)

Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache
  - Set Associative: block 12 can go anywhere
  - Block 12 can go anywhere in set 0 (12 mod 4)
  - Fully associative: block 12 can go anywhere

Example:

- Block no.
  - 0 1 2 3 4 5 6 7
  - 1 1 1 1 0 0 0 0

3/7 Ion Stoica CS162 ©UCB Spring 2011
Review: Caching Applied to Address Translation

- Problem: address translation expensive (especially multi-level)
- Solution: cache address translation (TLB)
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some...

TLB organization

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a “TLB Slice”
- When does TLB lookup occur?
  - Before cache lookup?
  - In parallel with cache lookup?

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
Putting Everything Together