Operating Systems

Semaphores, Condition Variables, and Monitors

Lecture 6
Michael O’Boyle
Semaphore

• More sophisticated Synchronization mechanism
• Semaphore $S$ – integer variable
• Can only be accessed via two indivisible (atomic) operations
  – $\text{wait}()$ and $\text{signal}()$
    • Originally called $\text{P}()$ and $\text{V}()$
• Definition
  
  $\text{wait}(S)$ {
    while ($S <= 0$) 
      ; // busy wait 
    $S--;$
  }

• Definition

  $\text{signal}(S)$ {
    $S++;$
  }

Do these operations \textit{atomically}
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a lock
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$
  - Create a semaphore “synch” initialized to 0

\[\begin{align*}
P1: \\
S_1; \\
signal(synch); \\
P2: \\
wait(synch); \\
S_2;
\end{align*}\]

- Can implement a counting semaphore $S$ as a binary semaphore
Implementation with no Busy waiting

Each semaphore has an associated queue of threads

wait(semaphore *S) { 
    S->value--; 
    if (S->value < 0) { 
        add this thread to S->list; 
        block(); 
    } 
} 

signal(semaphore *S) { 
    S->value++; 
    if (S->value <= 0) { 
        remove a thread T from S->list; 
        wakeup(T); 
    } 
}
Binary semaphore usage

• From the programmer’s perspective, P and V on a binary semaphore are just like Acquire and Release on a lock

```plaintext
P(sem)
  ...
  do whatever stuff requires mutual exclusion; could conceivably be a lot of code
  ...
V(sem)
```

  – same lack of programming language support for correct usage

• Important differences in the underlying implementation, however
• No busy waiting
Example: Bounded buffer problem

- **AKA “producer/consumer” problem**
  - there is a circular buffer in memory with N entries (slots)
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)
- **Threads are concurrent**
  - so, we must use synchronization constructs to control access to shared variables describing buffer state
Bounded buffer using semaphores
(both binary and counting)

```
var mutex: semaphore = 1 ; mutual exclusion to shared data
empty: semaphore = n  ; count of empty slots (all empty to start)
full: semaphore = 0   ; count of full slots (none full to start)
```

**producer:**
- `P(empty)` ; block if no slots available
- `P(mutex)` ; get access to pointers
  - `<add item to slot, adjust pointers>`
- `V(mutex)` ; done with pointers
- `V(full)` ; note one more full slot

**consumer:**
- `P(full)` ; wait until there’s a full slot
- `P(mutex)` ; get access to pointers
  - `<remove item from slot, adjust pointers>`
- `V(mutex)` ; done with pointers
- `V(empty)` ; note there’s an empty slot
  - `<use the item>`
Example: Readers/Writers

• Description:
  – A single object is shared among several threads/processes
  – Sometimes a thread just reads the object
  – Sometimes a thread updates (writes) the object
  – **We can allow multiple readers at a time**
    • Do not change state – no race condition
  – **We can only allow one writer at a time**
    • Change state- race condition
Readers/Writers using semaphores

\[
\begin{align*}
\text{var} & \quad \text{mutex: semaphore = 1} \quad ; \text{controls access to readcount} \\
& \quad \text{wrt: semaphore = 1} \quad ; \text{control entry for a writer or first reader} \\
& \quad \text{readcount: integer = 0} \quad ; \text{number of active readers} \\
\end{align*}
\]

\[
\begin{align*}
\text{writer:} & \quad \text{P(wrt)} \quad ; \text{any writers or readers?} \\
& \quad <\text{perform write operation}> \\
& \quad \text{V(wrt)} \quad ; \text{allow others} \\
\end{align*}
\]

\[
\begin{align*}
\text{reader:} & \quad \text{P(mutex)} \quad ; \text{ensure exclusion} \\
& \quad \text{readcount++} \quad ; \text{one more reader} \\
& \quad \text{if readcount == 1 then P(wrt)} \quad ; \text{if we’re the first, synch with writers} \\
& \quad \text{V(mutex)} \quad <\text{perform read operation}> \\
& \quad \text{P(mutex)} \quad ; \text{ensure exclusion} \\
& \quad \text{readcount--} \quad ; \text{one fewer reader} \\
& \quad \text{if readcount == 0 then V(wrt)} \quad ; \text{no more readers, allow a writer} \\
& \quad \text{V(mutex)} \\
\end{align*}
\]
• Notes:
  – the first reader blocks on $P(\text{wrt})$ if there is a writer
    • any other readers will then block on $P(\text{mutex})$
  – if a waiting writer exists, the last reader to exit signals the waiting writer
    • A new reader cannot get in while a writer is waiting
  – When writer exits, if there is both a reader and writer waiting, which one goes next?
Semaphores vs. Spinlocks

• Threads that are blocked at the level of program logic (that is, by the semaphore P operation) are placed on queues, rather than busy-waiting.

• Busy-waiting may be used for the “real” mutual exclusion required to implement P and V:
  – but these are very short critical sections – totally independent of program logic
  – and they are not implemented by the application programmer.
Abstract implementation

- **P/wait(sem)**
  - acquire “real” mutual exclusion
    - if sem is “available” (>0), decrement sem; release “real” mutual exclusion; let thread continue
    - otherwise, place thread on associated queue; release “real” mutual exclusion; run some other thread

- **V/signal(sem)**
  - acquire “real” mutual exclusion
    - if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    - if no threads are on the queue, sem is incremented
      » the signal is “remembered” for next time P(sem) is called
  - release “real” mutual exclusion
  - the “V-ing” thread continues execution
Problems with semaphores, locks

- They can be used to solve any of the traditional synchronization problems, but it’s easy to make mistakes
  - they are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the synchronization variable and the data being controlled by it
  - No control over their use, no guarantee of proper usage
    - Semaphores: will there ever be a V()?  
    - Locks: did you lock when necessary? Unlock at the right time? At all?
- Thus, they are prone to bugs
  - We can reduce the chance of bugs by “stylizing” the use of synchronization
  - Language help is useful for this
One More Approach: Monitors

- A **programming language construct** supports controlled shared data access
  - synchronization code is added by the compiler

- A class in which every method automatically acquires a lock on entry, and releases it on exit – it combines:
  - shared data structures (object)
  - procedures that operate on the shared data (object methods)
  - synchronization between concurrent threads that invoke those procedures

- Data can only be accessed from within the
  - protects the data from unstructured access
  - Prevents ambiguity about what the synchronization variable protects

- Addresses the key usability issues that arise with semaphores
A monitor

waiting queue of threads trying to enter the monitor

at most one thread in monitor at a time

shared data

operations (methods)

Proc A

Proc B

Proc C
Monitor facilities

• “Automatic” mutual exclusion
  – only one thread can be executing inside at any time
    • thus, synchronization is implicitly associated with the monitor – it “comes for free”
  – if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    • more restrictive than semaphores
    • but easier to use (most of the time)

• But, there’s a problem…
Problem: Bounded Buffer Scenario

- Buffer is empty
- Now what?
Problem: Bounded Buffer Scenario

- Buffer is full
- Now what?
Solution?

- Monitors require condition variables
- Operations on condition variables
  - wait(c)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - signal(c)
    - wake up at most one waiting thread
      - “Hoare” monitor: wakeup immediately, signaller steps outside
    - if no waiting threads, signal is lost
      - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting threads
Bounded buffer using (Hoare) monitors

Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  produce(resource x) {
    if (array “resources” is full, determined maybe by a count)
      wait(not_full);
    insert “x” in array “resources”
    signal(not_empty);
  }

  consume(resource *x) {
    if (array “resources” is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array “resources”
    signal(not_full);
  }
}
Problem: Bounded Buffer Scenario

- Buffer is full
- Now what?
Bounded Buffer Scenario with CV’s

- Buffer is full
- Now what?

Queue of threads waiting for condition “not full” to be signaled
Runtime system calls for (Hoare) monitors

• EnterMonitor(m) \{guarantee mutual exclusion\}
• ExitMonitor(m) \{hit the road, letting someone else run\}
• Wait(c) \{step out until condition satisfied\}
• Signal(c) \{if someone’s waiting, step out and let him run\}

• EnterMonitor and ExitMonitor are inserted automatically by the compiler.
• This guarantees mutual exclusion for code inside of the monitor.
Bounded buffer using (Hoare) monitors

Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource x) {
    if (array “resources” is full, determined maybe by a count)
      wait(not_full);
    insert “x” in array “resources”
    signal(not_empty);
  }

  procedure get_entry(resource *x) {
    if (array “resources” is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array “resources”
    signal(not_full);
  }
}
Monitor Summary

• Language supports monitors
• Compiler understands them
  – Compiler inserts calls to runtime routines for
    • monitor entry
    • monitor exit
  – Programmer inserts calls to runtime routines for
    • signal
    • wait
  – Language/object encapsulation ensures correctness
    • Sometimes! With conditions, you *still* need to think about synchronization
• Runtime system implements these routines
  – moves threads on and off queues
  – *ensures mutual exclusion!*