semaphores

Vaibhav Bajpai

OS 2012

synchronization patterns

- signaling
- rendezvous
- mutex
- multiplex
- barrier
- reusable barrier
- queue

synchronization patterns

- signaling
  - problem:

Thread A

```
statement a1
```

a1 < b1

Thread B

```
statement b1
```

  - solution:

```
a1_done = 0
```

Thread A

```
statement a1
a1_done.signal()
```

Thread B

```
a1_done.wait()
statement b1
```

synchronization patterns

- rendezvous
  - problem:

Thread A

```
statement a1
statement a2
```

a1 < b2
b1 < a2

Thread B

```
statement b1
statement b2
```

  - solution:

```
a_arrived = 0
b_arrived = 0
```

Thread A

```
statement a1
a_arrived.signal()
b_arrived.wait()
statement a2
```

Thread B

```
statement b1
b_arrived.signal()
a_arrived.wait()
statement b2
```

no of context switches?

synchronization patterns

- rendezvous
  - problem:

Thread A

```
statement a1
statement a2
```

a1 < b2
b1 < a2

Thread B

```
statement b1
statement b2
```

  - solution:

```
a_arrived = 0
b_arrived = 0
```

Thread A

```
statement a1
b_arrived.wait()
a_arrived.signal()
statement a2
```

Thread B

```
statement b1
b_arrived.signal()
a_arrived.wait()
statement b2
```

no of context switches?

synchronization patterns

- rendezvous
  - problem:

Thread A

```
statement a1
statement a2
```

a1 < b2
b1 < a2

Thread B

```
statement b1
statement b2
```

  - how about?

```
a_arrived = 0
b_arrived = 0
```

Thread A

```
statement a1
b_arrived.wait()
a_arrived.signal()
statement a2
```

Thread B

```
statement b1
a_arrived.wait()
b_arrived.signal()
statement b2
```

deadlock!

synchronization patterns

- mutex
  - problem:

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

  - solution:

```
mutex = 1
```

Thread A

```
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
```

synchronization patterns

- multiplex
  - problem:

Thread 1

```
count = count + 1
```

Thread 2

```
count = count + 1
```

...

Thread N

```
count = count + 1
```

  - solution:

```
mutex = N
```

Thread 1

```
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
```

Thread 2

```
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
```

...

Thread N

```
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
```

synchronization patterns

- barrier (generalized rendezvous)
  - problem:

Thread 1 ... N

```
rendezvous
critical point
```

  - hint:

```
N = no. of threads
count = 0
mutex = semaphore(1)
barrier = semaphore(0)
```

synchronization patterns

- barrier (generalized rendezvous)
  - problem:

Thread 1 … N

```
rendezvous
critical point
```

- how about?

```
N = no. of threads
count = 0
mutex = semaphore(1)
barrier = semaphore(0)
```

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == N: barrier.signal()

barrier.wait()

critical point
```

deadlock!

synchronization patterns

- barrier (generalized rendezvous)
  - problem:

Thread 1 … N

```
rendezvous
critical point
```

  - solution

```
N = no. of threads
count = 0
mutex = semaphore(1)
barrier = semaphore(0)
```

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == N: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

turnstile

synchronization patterns

- barrier (generalized rendezvous)
  - problem:

Thread 1 … N

```
rendezvous
critical point
```

  - solution

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == N: turnstile.signal() # unlock turnstile

turnstile.wait()
turnstile.signal()

critical point
```

synchronization patterns

- reusable barrier
  - problem:

Thread 1 … N

```
for(...):
  rendezvous
  critical point
```

- hint:

  re-lock the turnstile

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == N: turnstile.signal() # unlock turnstile

turnstile.wait()
turnstile.signal()

critical point

... ?
```

synchronization patterns

- reusable barrier
  - problem:

Thread 1 … N

```
for(...):
  rendezvous
  critical point
```

- how about?

all threads can lock and unlock the turnstile!

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == N: turnstile.signal() # unlock turnstile

turnstile.wait()
turnstile.signal()

critical point

mutex.wait()
  count = count - 1
mutex.signal()

if count == 0: turnstile.wait() # lock turnstile
```

synchronization patterns

- reusable barrier
  - problem:

Thread 1 … N

```
for(...):
  rendezvous
  critical point
```

- how about?

threads can loop around!

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
  if count == N: turnstile.signal() # unlock turnstile
mutex.signal()

turnstile.wait()
turnstile.signal()

critical point

mutex.wait()
  count = count - 1
  if count == 0: turnstile.wait() # lock turnstile
mutex.signal()
```

synchronization patterns

- reusable barrier
  - problem:

Thread 1 … N

```
for(...):
  rendezvous
  critical point
```

- hint:

```
turnstile_1 = semaphore(0)  # locked
turnstile_2 = semaphore(1)  # unlocked
mutex = semaphore(1)
```

synchronization patterns

- reusable barrier
  - problem:

Thread 1 … N

```
for(...):
  rendezvous
  critical point
```

  - solution

Thread 1 … N

```
rendezvous
mutex.wait()
  count = count + 1
  if count == N:
     turnstile_2.wait()    # lock turnstile #2
     turnstile_1.signal()  # unlock turnstile #1

mutex.signal()

turnstile_1.wait()         # turnstile #1
turnstile_1.signal()

critical point

mutex.wait()
  count = count - 1
  if count == 0:
     turnstile_1.wait()    # lock turnstile #1
     turnstile_2.signal()  # unlock turnstile #2
mutex.signal()

turnstile_2.wait()         # turnstile #2
turnstile_2.signal()
```

synchronization patterns

- queue
  - problem:
    - threads represent ballroom dancers.
    - there are 2 kinds of dancers: leaders and followers; waiting in 2 queues
    - when a leader arrives, it checks if a follower is waiting and vice versa
    - leader and follower form a pair and proceed to dance
  - hint:

```
leader_q = 0
follower_q = 0
```

leaders

```
...
dance()
```

followers

```
...
dance()
```

synchronization patterns

- queue
  - problem:
    - threads represent ballroom dancers.
    - there are 2 kinds of dancers: leaders and followers; waiting in 2 queues
    - when a leader arrives, it checks if a follower is waiting and vice versa
    - leader and follower form a pair and proceed to dance

- solution:

```
leader_q = 0
follower_q = 0
```

leaders

```
follower_q.signal()
leader_q.wait()
dance()
```

followers

```
leader_q.signal()
follower_q.wait()
dance()
```

synchronization patterns

- exclusive queue
  - problem:
    - threads represent ballroom dancers.
    - there are 2 kinds of dancers: leaders and followers; waiting in 2 queues
    - when a leader arrives, it checks if a follower is waiting and vice versa
    - leader and follower form a pair and proceed to dance
    - each leader can dance concurrently with only 1 follower and vice versa
  - hint:

```
leader_q = semaphore(0)
follower_q = semaphore(0)
rendezvous = semaphore(0)

leaders = followers = 0
mutex = semaphore(1)
```

synchronization patterns

- exclusive queue

```
leader_q = semaphore(0)
follower_q = semaphore(0)
rendezvous = semaphore(0)

leaders = followers = 0
mutex = semaphore(1)
```

- solution:

leaders

```
mutex.wait()
if followers > 0:
  followers = followers - 1
  follower_q.signal()
else:
  leaders = leaders + 1
  mutex.signal()
  leader_q.wait()

dance()
rendezvous.wait()
mutex.signal()
```

followers

```
mutex.wait()
if leaders > 0:
  leaders = leaders - 1
  leader_q.signal()
else:
  followers = followers + 1
  mutex.signal()
  follower_q.wait()

dance()
rendezvous.signal()
```