Starvation and Deadlock

Another EventBarrier Example

EventBarrier channel;

void OutputThread ()
{  
    void OutputThread ()
{  
    while (TRUE) {
        computeDataToSend();
        channel.Wait();
        sendData();
        channel.Complete();
    }
}

void ChannelScheduler ()
{  
    void ChannelScheduler ()
{  
    while (TRUE) {
        waitUntilTimeToOpenChannel();
        channel.Signal(); /* open floodgate for burst of outgoing data */
        /* channel is closed */
    }
}

  Invariants:
  1. Output thread never blocks in Wait() if the
      channel is already open.
  2. Channel never closes while a thread is
      sending data.
  3. Each thread sends at most once each time
      the channel opens.
Highway 110 with EventBarrier

EventBarrier north;
EventBarrier south;
void HeadingNorth() {
    north.Wait();
    go through one-lane tunnel;
    north.Complete();
}
void HeadingSouth() {
    south.Wait();
    go through one-lane tunnel;
    south.Complete();
}
void TunnelScheduler() {
    while (TRUE) {
        north.Signal();
        south.Signal();
        ...
    }
}

Problems?

Reader/Writer with Semaphores

SharedLock::AcquireRead() {
    rmx.P();
    if (first reader)
        wsem.P();
    rmx.V();
}
SharedLock::AcquireWrite() {
    wsem.P();
}
SharedLock::ReleaseRead() {
    rmx.P();
    if (last reader)
        wsem.V();
    rmx.V();
}
SharedLock::ReleaseWrite() {
    wsem.V();
}
Reader/Writer with Semaphores: Take 2

```cpp
SharedLock::AcquireRead() {
  rblock.P();
  rmx.P();
  if (first reader)
    wsem.P();
  rmx.V();
  rblock.V();
}

SharedLock::ReleaseRead() {
  if (last reader)
    wsem.V();
  rmx.V();
}

SharedLock::AcquireWrite() {
  wmx.P();
  if (first writer)
    rblock.P();
  wmx.V();
  wsem.P();
}

SharedLock::ReleaseWrite() {
  wsem.V();
  wmx.P();
  if (last writer)
    rblock.V();
  wmx.V();
}
```

This is a bit difficult to read, so let's simplify it....

---

Reader/Writer with Semaphores: Take 2+

```cpp
SharedLock::AcquireRead() {
  rblock.P();
  if (first reader)
    wsem.P();
  rblock.V();
}

SharedLock::ReleaseRead() {
  if (last reader)
    wsem.V();
}

SharedLock::AcquireWrite() {
  if (first writer)
    rblock.P();
  wsem.P();
}

SharedLock::ReleaseWrite() {
  wsem.V();
  if (last writer)
    rblock.V();
}
```

The rblock prevents readers from entering while writers are waiting.
**Starvation**

The reader/writer lock example illustrates *starvation*: under load, a writer will be stalled forever by a stream of readers.

- **Example:** a *one-lane bridge or tunnel*.
  - Wait for oncoming car to exit the bridge before entering.
  - Repeat as necessary.
- **Problem:** a “writer” may never be able to cross if faced with a continuous stream of oncoming “readers”.
- **Solution:** some reader must politely stop before entering, even though it is not forced to wait by oncoming traffic.
  - Use extra synchronization to control the lock scheduling policy.
  - Complicates the implementation: optimize only if necessary.

**Deadlock**

*Deadlock* is closely related to starvation.

- Processes wait forever for each other to wake up and/or release resources.
- **Example:** *traffic gridlock*.

The difference between deadlock and starvation is subtle.

- With starvation, there always exists a schedule that feeds the starving party.
  - The situation may resolve itself…if you’re lucky.
- Once deadlock occurs, it cannot be resolved by any possible future schedule.
  - …though there may exist schedules that *avoid* deadlock.
Dining Philosophers

- $N$ processes share $N$ resources
- resource requests occur in pairs
- random think times
- hungry philosopher grabs a fork
- ...and doesn’t let go
- ...until the other fork is free
- ...and the linguine is eaten

```
while(true) {
    Think();
    AcquireForks();
    Eat();
    ReleaseForks();
}
```

Four Preconditions for Deadlock

Four conditions must be present for deadlock to occur:

1. **Non-preemptability.** Resource ownership (e.g., by threads) is non-preemptable.
   
   Resources are never taken away from the holder.

2. **Exclusion.** Some thread cannot acquire a resource that is held by another thread.

3. **Hold-and-wait.** Holder blocks awaiting another resource.

4. **Circular waiting.** Threads acquire resources out of order.
Resource Graphs

Given the four preconditions, some schedules may lead to *circular waits*.

- Deadlock is easily seen with a *resource graph* or *wait-for graph*.

  The graph has a vertex for each process and each resource.  
  If process A holds resource R, add an arc from R to A.  
  If process A is waiting for resource R, add an arc from A to R.  

*The system is deadlocked iff the wait-for graph has at least one cycle.*

```
A grabs fork 1 and
waits for fork 2.

B grabs fork 2 and
waits for fork 1.
```

Not All Schedules Lead to Collisions

The scheduler chooses a path of the executions of the threads/processes competing for resources.

  Synchronization constrains the schedule to avoid illegal states.  
  Some paths “just happen” to dodge dangerous states as well.

*What is the probability that philosophers will deadlock?*

- How does the probability change as:
  - think times increase?  
  - number of philosophers increases?
**Resource Trajectory Graphs**

Resource trajectory graphs (RTG) depict the scheduler’s “random walk” through the space of possible system states.

RTG for N processes is N-dimensional.

Process \( i \) advances along axis \( I \).

Each point represents one state in the set of all possible system states.

cross-product of the possible states of all processes in the system

(But not all states in the cross-product are legally reachable.)

---

**RTG for Two Philosophers**

(There are really only 9 states we care about: the important transitions are allocate and release events.)
Two Philosophers Living Dangerously

The Inevitable Result

no legal transitions out of this deadlock state
Dealing with Deadlock

1. Ignore it. “How big can those black boxes be anyway?”

2. Detect it and recover. Traverse the resource graph looking for cycles before blocking any customer.
   - If a cycle is found, preempt: force one party to release and restart.

3. Prevent it statically by breaking one of the preconditions.
   - Assign a fixed partial ordering to resources; acquire in order.
   - Use locks to reduce multiple resources to a single resource.
   - Acquire resources in advance of need; release all to retry.

4. Avoid it dynamically by denying some resource requests.
   - Banker’s algorithm

Extending the Resource Graph Model

Reasoning about deadlock in real systems is more complex than the simple resource graph model allows.

- Resources may have multiple instances (e.g., memory).
  Cycles are necessary but not sufficient for deadlock.
  For deadlock, each resource node with a request arc in the cycle must be fully allocated and unavailable.

- Processes may block to await events as well as resources.
  E.g., A and B each rely on the other to wake them up for class.
  These “logical” producer/consumer resources can be considered to be available as long as the producer is still active.
  Of course, the producer may not produce as expected.
Banker’s Algorithm

The Banker’s Algorithm is the classic approach to deadlock avoidance (choice 4) for resources with multiple units.

1. Assign a credit limit to each customer.
   “maximum claim” must be stated/negotiated in advance

2. Reject any request that leads to a dangerous state.
   A dangerous state is one in which a sudden request by any customer(s) for the full credit limit could lead to deadlock.
   A recursive reduction procedure recognizes dangerous states.

3. In practice, this means the system must keep resource usage well below capacity to maintain a reserve surplus.
   Rarely used in practice due to low resource utilization.