12 Commandments of Synchronization

I. Thou shalt name your synchronization variables properly
II. Thou shalt not violate abstraction boundaries nor try to change the semantics of synchronization primitives
III. Thou shalt use monitors and condition variables instead of semaphores whenever possible
IV. Thou shalt not mix semaphores and condition variables
V. Thou shalt not busy wait
VI. Thou shalt protect all shared state
VII. Thou shalt grab the monitor lock upon entry to, and release it upon exit from, a procedure
VIII. Honor thy shared data with an invariant, which your code may assume holds when a lock is acquired successfully and your code must make true before the lock is released
IX. Thou shalt cover thy naked waits.
X. Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement
X1. Thou shalt not split predicates.
XII. Thou shalt help make the world a better place for the creator’s mighty synchronization vision.

Monitors in “4410 Python”:

```python
class BK:
    def __init__(self):
        self.lock = Lock()
        self.hungrykid = Condition(self.lock)
        self.nBurgers= 0

    def make_burger(self):
        with self.lock:
            self.nBurgers = self.nBurgers + 1
            self.hungrykid.notify()

    def kid_eat(self):
        with self.lock:
            while self.nBurgers == 0:
                self.hungrykid.wait()
            self.nBurgers = self.nBurgers - 1

def make_burger(self):
    with self.lock:
        self.nBurgers = self.nBurgers + 1
        self.hungrykid.notify()
```

wait
• releases lock when called
• re-acquires lock when it returns

Monitors in Python

```python
class BK:
    def __init__(self):
        self.lock = Lock()
        self.hungrykid = Condition(self.lock)
        self.nBurgers= 0

def kid_eat(self):
    with self.lock:
        while self.nBurgers == 0:
            self.hungrykid.wait()
        self.nBurgers = self.nBurgers - 1

def make_burger(self):
    with self.lock:
        self.nBurgers = self.nBurgers + 1
        self.hungrykid.notify()
```

Monitors in “4410 Python”:

`wait` releases lock when called and re-acquires lock when it returns.

We do this for helpful feedback:
• from auto-grader
• from debugger

Look in the A2/doc directory for details and example code.
Readers/Writers

Safety

\((\#r \geq 0) \land (0 \leq \#w \leq 1) \land (\#r > 0) \Rightarrow (\#w = 0))\)

What about fairness?

- The last thread to leave the critical section will give priority to writers.
- To implement this policy, one needs to keep track of waitingWriters, waitingReaders, activeWriters, and activeReaders.

```java
Monitor ReadersNWriters {
    int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
    Condition canRead, canWrite;

    void BeginWrite()
    with monitor.lock:
        ++waitingWriters
        while (activeWriters > 0 or activeReaders > 0)
            canWrite.wait();
        --waitingWriters
        activeWriters = 1;

    void EndWrite()
    with monitor.lock:
        activeWriters = 0
        if waitingWriters > 0
            canWrite.signal();
        else if waitingReaders > 0
            canRead.broadcast();
}
```

Readers/Writers

```java
Monitor ReadersNWriters {
    int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
    Condition canRead, canWrite;

    void BeginWrite()
    with monitor.lock:
        ++waitingWriters
        while (activeWriters > 0 or waitingWriters > 0)
            canWrite.wait();
        --waitingWriters
        activeWriters = 1;

    void BeginRead()
    with monitor.lock:
        ++waitingReaders
        while (activeWriters > 0 or waitingWriters > 0)
            canRead.wait();
        --waitingReaders
        ++activeReaders

    void EndWrite()
    with monitor.lock:
        activeWriters = 0
        if waitingWriters > 0
            canWrite.signal();
        else if waitingReaders > 0
            canRead.broadcast();
}
```
Readers/Writers

Monitor ReadersN Writers
int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
Condition canRead, canWrite;

void BeginWrite()
with monitor.lock:
++waitingWriters
while (activeWriters >0 or activeReaders >0)
canWrite.wait();
--waitingWriters
activeWriters = 1;

void EndWrite()
with monitor.lock:
activeWriters = 0
if waitingWriters > 0
canWrite.signal();
else if waitingReaders > 0
activeReaders = 0

void BeginRead()
with monitor.lock:
++waitingReaders
while (activeWriters>0 or waitingWriters>0)
canRead.wait();
--waitingReaders
++activeReaders

void EndRead()
with monitor.lock:
--activeReaders;
if (activeReaders==0 and waitingWriters>0)
canWrite.signal();

Barrier Synchronization

n threads divide work, run rounds of computation separated by barriers

Common paradigm in HPC
  Create n threads and barrier
  Each thread does round1()
  barrier.checkin()
  Each thread does round2()
  barrier.checkin()

Checkin with one condition variable

self.allCheckedIn = Condition(self.lock)
def checkin():
with self.lock:
nArrived++
if nArrived < nThreads:
  while nArrived < nThreads:
    allCheckedIn.wait()
else:
  allCheckedIn.broadcast()
nArrived = 0
Deadlocks:
Prevention, Avoidance, Detection, Recovery

System Model

- Exclusive (one-at-a-time) computer resources
  - CPU, printers, memory, locks, etc.

- Processes
  - Acquire resource
    - if resource is available, access is granted
    - if not, process is blocked
  - Use resource
  - Release resource

Dining Philosophers

- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!
- Need exclusive access to two chopsticks

Deadlock

- A cycle of waiting among a set of threads
- A violation of liveness
  - \( T_1 \) acquire resource 1, waits for resource 2
  - \( T_2 \) acquires resource 2, waits for resource 1

Semaphore:
- file_mutex = 1
- printer_mutex = 1

```python
class Philosopher:
    chopsticks[N] = [Semaphore(1),…]
    def __init__(mynum):
        self.id = mynum
    def eat():
        right = self.id
        left = (self.id+1) % N
        while True:
            P(chopsticks[left])
            P(chopsticks[right])
            # om nom nom nom
            V(chopsticks[right])
            V(chopsticks[left])
```

```python
T1
            { P(file_mutex)
              P(printer_mutex)
              /* use resources */
              V(printer_mutex)
              V(file_mutex)
            }

T2
            { P(printer_mutex)
              P(file_mutex)
              /* use resources */
              V(file_mutex)
              V(printer_mutex)
            }
```
Musings on Deadlock

Deadlock vs Starvation
- Starvation: some thread's access to a resource is indefinitely postponed
- Deadlock: circular waiting for resources
- Deadlock implies Starvation, but not vice versa
- “Subject to deadlock” does not imply “Will deadlock”
  - Testing is not the solution
  - System must be deadlock-free by design

A Graph Theoretic Model of Deadlock

Computer system modeled as a RAG, a directed graph \( G(V, E) \)
- \( V = \{P_1,\ldots,P_n\} \cup \{R_1,\ldots,R_n\} \)
- \( E = \{\text{edges from a resource to a process}\} \cup \{\text{edges from a process to a resource}\} \)

Necessary Conditions for Deadlock

Deadlock possible only if all four hold
- **Bounded resources**
  - A finite number of threads can use a resource; resources are finite
- **No preemption**
  - the resource is mine, MINE! (until I release it)
- **Hold & Wait**
  - holds one resource while waiting for another
- **Circular waiting**
  - \( T_i \) waits for \( T_{i+1} \) and holds a resource requested by \( T_{i-1} \)
  - sufficient only if one instance of each resource

RAG Reduction

**Not sufficient in general**

Deadlock?

NO! (no cycles)

Step 1: Satisfy \( P_3 \) requests
Step 2: Satisfy \( P_2 \) requests
Step 3: Satisfy \( P_1 \) requests
Schedule \( [P_1, P_2, P_3] \) completely 

Eliminates edges!
RAG Reduction

Deadlock?

Step 1: Satisfy P₃ requests
Step 2: Satisfy P₂ requests
Step 3: Satisfy P₁ requests
Schedule [P₃ P₂ P₁] completely eliminates edges!

More Musings on Deadlock

Does the order of RAG reduction matter?

- No. If Pᵢ and Pⱼ can both be reduced, reducing Pᵢ does not affect the reducibility of Pⱼ

Does a deadlock disappear on its own?

- No. Unless a process is killed or forced to release a resource, we are stuck!

If a system is not deadlock at time T, is it guaranteed to be deadlock-free at T+1?

- No. Just by requesting a resource (never mind being granted one) a process can create a circular wait!

Proactive Responses to Deadlock: Prevention

Negate one of deadlock’s four necessary conditions

- Remove “Mutual exclusion/Bounded Resources”
  - Make resources sharable without locks
    - Wait-Free synchronization
  - Make more resources available (duh!)
- Remove “No preemption”
  - Allow OS to preempt resources of waiting processes
  - Allow OS to preempt resources of requesting process if not all available
Proactive Responses to Deadlock: Prevention

Negate one of deadlock's four necessary conditions

- Remove "Hold & Wait"
  - Request all resources before execution begins
  - Processes may not know what they will need
  - Starvation (if waiting for many popular resources)
  - Low utilization (if resource needed only for a bit)
  - Release all resources before asking anything new
  - Still has the last two problems...

- Remove "Circular waiting"
  - Single lock for entire system?
  - Impose total/partial order on resources
    - A cycle needs edges to go from low to high, and then back to low (or to cycle on the same node)
    - Only a convention...

Preventing Philosopher’s Deadlock

class Philosopher:
    chopsticks[N] = [Semaphore(1),...]

    def __init__(mynum):
        self.id = mynum

    def eat():
        right = self.id % N
        left = (self.id + 1) % N
        while True:
            P(left)
            P(right)
            # om nom nom nom
            V(right)
            V(left)

Living dangerously: Safe, Unsafe, Deadlocked States

Safe state:
- It is possible to avoid deadlock and eventually grant all resource by careful scheduling (a safe schedule)
- Transitioning among safe states may delay a resource request even when resources are available

Unsafe state:
- Unlucky sequence of requests can force deadlock

Deadlocked state:
- System has at least one deadlock
Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
  - If lenders reduced or delayed their requests, all would be well!
  - spoiler alert: this is exactly what happens…
- Still begs the question:
  - Can resources be allocated so that the system always transitions among safe states?

Proactive Responses to Deadlock: Avoidance

The Banker’s Algorithm

- Processes declare worst-case needs (big assumption!), but then ask for what they “really” need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of \([P_1, P_2, \ldots, P_n]\) such that, for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus the resources currently held by all \(P_j\) for \(P_j\) preceding \(P_i\) in the permutation

<table>
<thead>
<tr>
<th>Available</th>
<th>Process</th>
<th>Hold</th>
<th>Needs</th>
<th>MaxClaim</th>
<th>HasNow</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_2)</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If so, request is granted; otherwise, requester must wait.
An Example

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0 1 2</td>
<td>0 1 2</td>
</tr>
<tr>
<td>P2</td>
<td>1 7 5</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 3 5</td>
<td>1 3 5</td>
</tr>
<tr>
<td>P4</td>
<td>0 5 2</td>
<td>0 6 3</td>
</tr>
<tr>
<td>P5</td>
<td>0 6 5</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

- 5 processes, 4 resources

- Is this a safe state?

An Example

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0 1 2</td>
<td>1 5 2</td>
</tr>
<tr>
<td>P2</td>
<td>0 0 0</td>
<td>0 7 5</td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>1 3 5</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 0</td>
<td>0 6 3</td>
</tr>
<tr>
<td>P5</td>
<td>0 0 0</td>
<td>0 6 4</td>
</tr>
</tbody>
</table>

- 5 processes, 4 resources

- Is this a safe state?

An Example

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0 1 2</td>
<td>0 1 2</td>
</tr>
<tr>
<td>P2</td>
<td>1 7 5</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 3 5</td>
<td>1 3 5</td>
</tr>
<tr>
<td>P4</td>
<td>0 6 5</td>
<td>0 0 1</td>
</tr>
<tr>
<td>P5</td>
<td>0 6 5</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

- P2 want to change its holdings to 0 4 2 0

- Safe?
Reactive Responses to Deadlock

- **Deadlock Detection**
  - Track resource allocation (who has what)
  - Track pending requests (who’s waiting for what)

- **When should it run?**
  - For each request?
  - After each unsatisfiable request?
  - Every hour?
  - Once CPU utilization drops below a threshold?

Detecting Deadlock

- 5 processes, 3 resources. We no longer know Max.

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Deadlock Recovery

- Blue scree & reboot
- Kill one/all deadlocked processes
  - Pick a victim (how?); Terminate; Repeat as needed
    - Can leave system in inconsistent state
- Proceed without the resource
  - Example: timeout on inventory check at Amazon
- Use transactions
  - Rollback & Restart
  - Need to pick a victim...
Summary

- **Prevent**
  - Negate one of the four necessary conditions

- **Avoid**
  - Schedule processes carefully

- **Detect**
  - Has a deadlock occurred?

- **Recover**
  - Kill or Rollback