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 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 += 1
            self.hungrykid.notify()

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

Monitors in “4410 Python”:

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

from rvr import MP, MPthread

class BurgerKingMonitor(MP):
    def __init__(self):
        MP.__init__(self, None)
        self.lock = Lock("monitor lock")
        self.hungrykid = self.lock.Condition("hungry kid")
        self.nBurgers = self.Shared("num burgers", 0)

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

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

Monitors in “4410 Python”: kid_eat

```python
def kid_eat(self):
    with self.lock:
        while self.nBurgers == 0:
            self.hungrykid.wait()
        self.nBurgers += 1
        self.hungrykid.notify()
```

Monitors in “4410 Python”: kid_eat

```python
def kid_eat(self):
    with self.lock:
        while (self.nBurgers.read() == 0):
            self.hungryKid.wait()
        self.nBurgers.decr()
```

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

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();
}

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();

void BeginRead()
    with monitor.lock:
        ++waitingReaders
        while (activeWriters >0 or waitingWriters >0)
            canRead.wait();
        --waitingReaders
        activeReaders = 1;
Readers/Writers

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();
}

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

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

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

```c
semaphore:
  file_mutex = 1
  printer_mutex = 1

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

{ 
  P(printer_mutex)
  P(file_mutex)
  /* use resources */
  V(file_mutex)
  V(printer_mutex)
}
```
Dining Philosophers

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

```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])
```

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
Deadlock possible only if all four hold

1. **Bounded resources**
   - A finite number of threads can use a resource; resources are finite
2. **No preemption**
   - The resource is mine, MINE! (until I release it)
3. **Hold & Wait**
   - Holds one resource while waiting for another
4. **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

**Necessary Conditions for Deadlock**

1. **Bounded resources**
   - A finite number of threads can use a resource; resources are finite
2. **No preemption**
   - The resource is mine, MINE! (until I release it)
3. **Hold & Wait**
   - Holds one resource while waiting for another
4. **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

**Necessary Conditions for Deadlock**

1. **Bounded resources**
   - A finite number of threads can use a resource; resources are finite
2. **No preemption**
   - The resource is mine, MINE! (until I release it)
3. **Hold & Wait**
   - Holds one resource while waiting for another
4. **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

**Necessary Conditions for Deadlock**

1. **Bounded resources**
   - A finite number of threads can use a resource; resources are finite
2. **No preemption**
   - The resource is mine, MINE! (until I release it)
3. **Hold & Wait**
   - Holds one resource while waiting for another
4. **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
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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each 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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each resource
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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each 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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each resource

---

**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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each resource

---

**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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each resource

---

**Not sufficient in general**

- **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**
  - Ti waits for Ti+1 and holds a resource requested by Ti-1
  - sufficient only if one instance of each resource
RAG Reduction

Deadlock?

NO! (no cycles)

Step 1: Satisfy P₃ requests

NO! (no cycles)

Step 1: Satisfy P₃ requests

NO! (no cycles)
RAG Reduction

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
NO! (no cycles)

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
NO! (no cycles)
RAG Reduction

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edges!
Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edge!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edge!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edge!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃, P₂, P₁] completely eliminates edge!

Deadlock?
Cannot satisfy any of P₁, P₂, P₃ requests!

Deadlock?
RAG has a cycle

Deadlock?
Cannot satisfy any of P₁, P₂, P₃ requests!

Deadlock?
RAG has a cycle

Deadlock?
RAG has a cycle

Deadlock?
RAG has a cycle
RAG Reduction

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₁'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₃'s requests
Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
Yes!
RAG has a cycle
Cannot satisfy any of P₁, P₂, P₃ requests!

Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₁'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₃'s requests
Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
Yes!
RAG has a cycle
Cannot satisfy any of P₁, P₂, P₃ requests!

Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₁'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₃'s requests
Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
Yes!
RAG has a cycle
Cannot satisfy any of P₁, P₂, P₃ requests!

Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
NO! (no cycles)
Step 1: Satisfy P₁'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₃'s requests
Schedule (P₁, P₂, P₃) completely eliminates edges!

Deadlock?
Yes!
RAG has a cycle
Cannot satisfy any of P₁, P₂, P₃ requests!

Schedule (P₁, P₂, P₃) completely eliminates edges!
**RAG Reduction**

1. **Step 1:** Satisfy P₃'s requests
2. **Step 2:** Satisfy P₂'s requests
3. **Step 3:** Satisfy P₁'s requests

Schedule [P₃, P₂, P₁] completely eliminates edges!

**Deadlock?**

- NO! (no cycles)
- YES!

**RAG has a cycle**

Cannot satisfy any of P₁, P₂, P₃ requests!

**RAG has a cycle**

Schedule [P₂, P₁, P₃] completely eliminates edges!
More Musings on Deadlock

Does the order of RAG reduction matter?

- No. If $P_i$ and $P_j$ can both be reduced, reducing $P_i$ does not affect the reducibility of $P_j$

Does a deadlock disappear on its own?

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

Can we prevent one of these conditions?
Ideas?

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?
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...
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...
- George relied on the bad execution in which "everyone wants everything now" not happening

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
E.W. Dijkstra & N. Habermann

- 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
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 \([P_0, P_1, \ldots, P_n]\) 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>Process</th>
<th>Max Need</th>
<th>Holds</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_0)</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(P_1)</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(P_2)</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Available resources can satisfy \(P_1\)’s needs.
Proactive Responses to Deadlock: Avoidance

The Banker’s Algorithm

E.W. Dijkstra & N. Habermann

- 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 \([P_1, P_2, ..., P_n]\) 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>Need</th>
<th>Max</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>(P_1)</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Available</td>
<td>(P_2)</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Available</td>
<td>(P_3)</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- Available resources can satisfy \(P_1\)’s needs
- Once \(P_1\) finishes, 5 available resources
- Available resources can satisfy \(P_2\)’s needs
- Once \(P_2\) finishes, 5 available resources
- Available resources can satisfy \(P_3\)’s needs
- Once \(P_3\) finishes, 10 available resources

Available = 3
Process: \(P_1\), \(P_2\), \(P_3\)
Need: 2, 3, 4
Max: 2, 2, 2
Hold: 2, 2, 2

Safe?

- 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 \([P_1, P_2, ..., P_n]\) 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>Need</th>
<th>Max</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>(P_1)</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Available</td>
<td>(P_2)</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Available</td>
<td>(P_3)</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- Available resources can satisfy \(P_1\)’s needs
- Once \(P_1\) finishes, 5 available resources
- Available resources can satisfy \(P_2\)’s needs
- Once \(P_2\) finishes, 5 available resources
- Available resources can satisfy \(P_3\)’s needs
- Once \(P_3\) finishes, 10 available resources

Available = 3
Process: \(P_1\), \(P_2\), \(P_3\)
Need: 2, 3, 4
Max: 2, 2, 2
Hold: 2, 2, 2

Safe?
Proactive Responses to Deadlock: Avoidance

The Banker’s Algorithm

E.W. Dijkstra & N. Habermann

- 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.
The Banker's books

- Assume \( n \) processes, \( m \) resources
- \( \text{Max}_{ij} \) = max amount of units of resource \( R_j \) needed by \( P_i \)
  - \( \text{MaxClaim}_i \) = Vector of size \( m \) such that \( \text{MaxClaim}_i[j] = \text{Max}_{ij} \)
- \( \text{Holds}_{ij} \) = current allocation of \( R_j \) held by \( P_i \)
  - \( \text{HasNow}_i \) = Vector of size \( m \) such that \( \text{HasNow}_i[j] = \text{Holds}_{ij} \)
- \( \text{Available} \) = Vector of size \( m \) such that \( \text{Available}[j] \) units of \( R_j \) available

A request by \( P_k \) is safe if, assuming the request is granted, there is a permutation of \( P_1, P_2, \ldots, P_n \) such that, for all \( P_i \) in the permutation

\[
\text{Needs}_i = \text{MaxClaim}_i - \text{HasNow}_i \leq \text{Avail} + \sum_{j=1}^{i-1} \text{HasNow}_j
\]

An Example

- 5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{Holds} )</th>
<th>( \text{Available} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>1</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
</tr>
</tbody>
</table>

- Is this a safe state?

An Example

- 5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{Holds} )</th>
<th>( \text{Available} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>1</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
</tr>
</tbody>
</table>

- Is this a safe state?
An Example

5 processes, 4 resources
Max
P1 0 1 1 2
P2 1 7 5 0
P3 2 3 5 6
P4 0 6 5 2
P5 0 6 5 6

Holds
P1 0 0 1 2
P2 1 0 0 0
P3 1 3 5 3
P4 0 6 3 2
P5 0 0 1 4
Available
R1 R2 R3 R4
1 5 2 0

Needs
P1 0 0 0 0
P2 0 7 5 0
P3 1 0 0 3
P4 0 0 2 0
P5 0 6 4 2

Is this a safe state?

1. While safe permutation does not include all processes:
   - Is there a \( P_i \) such that \( \text{Needs}_{i} \leq \text{Avail} \)?
     - if no, exit with \text{unsafe}
     - if yes, add \( P_i \) to the sequence and set \( \text{Avail} = \text{Avail} + \text{HasNow} \)
2. Exit with \text{safe}

P2 want to change its holdings to 0 4 2 0

An Example

5 processes, 4 resources
Max
P1 0 0 1 2
P2 1 7 5 0
P3 2 3 5 6
P4 0 6 5 2
P5 0 6 5 6

Holds
P1 0 0 1 2
P2 1 0 0 0
P3 1 3 5 3
P4 0 6 3 2
P5 0 0 1 4
Available
R1 R2 R3 R4
1 5 2 0

Needs
P1 0 0 0 0
P2 0 7 5 0
P3 1 0 0 3
P4 0 6 3 2
P5 0 0 1 4

Is this a safe state?

1. While safe permutation does not include all processes:
   - Is there a \( P_i \) such that \( \text{Needs}_{i} \leq \text{Avail} \)?
     - if no, exit with \text{unsafe}
     - if yes, add \( P_i \) to the sequence and set \( \text{Avail} = \text{Avail} + \text{HasNow} \)
2. Exit with \text{safe}

P1, P4, P2, P3, P5

P2 want to change its holdings to 0 4 2 0
An Example

- 5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

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

An Example

- 5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Holds</th>
<th>Available</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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.**

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( R_2 )</td>
<td>( R_3 )</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

### Detecting Deadlock

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

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( R_2 )</td>
<td>( R_3 )</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

### Detecting Deadlock

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

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( R_2 )</td>
<td>( R_3 )</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Can we avoid deadlock by delaying granting requests?

### Detecting Deadlock

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

<table>
<thead>
<tr>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( R_2 )</td>
<td>( R_3 )</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Can we avoid deadlock by delaying granting requests?
  - Deadlock triggered when request formulated, not granted!
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