Deadlock Detection and Avoidance

44-550: Operating Systems

Requirements for Deadlock

1. Hold and Wait
   - Process/thread holds resource while waiting for more resources
2. Mutual Exclusion
   - Resource can't be shared
3. Circular Wait
   - Process wants something another processor has
4. No preemption
   - Can't interrupt a process' hold on a resource

Resource Allocation Graph

- Circles represent processes
- Rectangles represent resources
- Dots within the resource represent number of resources of that type
- Arrows from resources to processes indicate resource ownership
- Arrows from process to resource indicate need/request

Using a Resource Allocation Graph

- We can show deadlock exists by showing the presence of all four requirements
  - The set of 4 requirements is both necessary and sufficient
- We can show deadlock will not exist in the problem by proving one of the requirements does not exist.

Dealing with Deadlock

- Ignore it!
- Detect and recover from it
- Avoid it (make deadlock impossible)
- Prevent it (short circuit conditions and wait, or circular wait)

We need mutual exclusion and prefer to NOT preempt necessary resources

Dining Philosophers: Hold and Wait

- Process must get all required resources before continuing
  - Good for batch systems
- Process must release all resource before requesting new resources
  - Good for interactive systems

Only allow philosophers to get forks if both are available. If there are n philosophers dining, only $\lfloor \frac{n}{2} \rfloor$ may eat at one time;

Pseudocode

```
while true:
    acquire semaphore
    if both chopsticks are available:
        get left chopstick
        get right chopstick
        eat
        release both forks
    release semaphor
    think
```

Dining Philosophers: Circular Wait

- Disallow circular wait
- Can do by forcing an ordering on the resources
- Only get forks based on lowest index first (Dijkstra's solution, ish)

Pseudocode

```
while true:
    if index_left < index_right:
        try to get left chopstick
        try to get right chopstick
    else
        try to get right chopstick
        try to get left chopstick
    eat
    release both chopsticks
    think
```

Think about how to code these in C...

Avoiding Deadlock

- Only allow system to change state if it is *safe* to
- Analyze system state to ensure it is safe
- When requests occur, examine current state/allocation and maximum claim
- Maximum claim is the most resources a process will ever need

Banker's Algorithm

- Check if an allocation state is a safe state
- Define the resource allocation state as:
  - Total number of resources in the system
  - Number of allocated resources
  - Maximum resource claim of a process

The Banker's Algorithm Data Structure

- Vector C
  - `C[r]`: Number of resources of type `r`
- Matrix A
  - `A[p][r]`: Number of resources of type `r` allocated to process `p`
- Matrix M
  - `M[p][r]`: Maximum number of resources of type `r` needed by process `p`

Preparing for an Example

- Assume a system with five processes with four resource types
  - P1, P2, P3, P4, P5
  - R1, R2, R3, R4
- Assume following resource limitations
  - 5 units of R1
  - 6 units of R2
  - 8 units of R3
  - 4 units of R4
- C=$[5, 6, 8, 4]$

Example Data

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(2,1,1,0)	(1,1,1,2)
P2	(2,1,1,2)	(0,1,1,0)	(2,0,0,2)
P3	(1,1,3,1)	(1,1,1,0)	(0,0,2,1)
P4	(3,4,2,2)	(1,1,2,1)	(2,3,0,1)
P5	(2,4,1,2)	(1,2,1,1)	(1,2,0,3)

Total resources(C): (5,6,8,4)
Total allocated: (5,6,6,2)
Total available: (0,0,2,2)
Assume processes will run to completion provided their resource needs are met. Once a process completes, it releases the resources.
We can see that only P3 is capable of running to completion

Safe Sequence of States

P3 gets (0,0,2,1)

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(2,1,1,0)	(1,1,1,2)
P2	(2,1,1,2)	(0,1,1,0)	(2,0,0,2)
P3	(1,1,3,1)	(1,1,3,1)	(0,0,0,0)
P4	(3,4,2,2)	(1,1,2,1)	(2,3,0,1)
P5	(2,4,1,2)	(1,2,1,1)	(1,2,0,3)

Total resources: (5,6,8,4)
Total allocated: (5,6,8,3)
Total available: (0,0,0,1)
P3 finishes and releases its resources (1,1,3,1)
Total allocated: (4,5,5,2)
Total available: (1,1,3,2)

Safe Sequence of States

P1 gets (1,1,1,2)

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(3,2,2,2)	(0,0,0,0)
P2	(2,1,1,2)	(0,1,1,0)	(2,0,0,2)
P3	(1,1,3,1)	(0,0,0,0)	(0,0,0,0)
P4	(3,4,2,2)	(1,1,2,1)	(2,3,0,1)
P5	(2,4,1,2)	(1,2,1,1)	(1,2,0,3)

Total resources: (5,6,8,4)
Total allocated: (5,6,6,4)
Total available: (0,0,2,0)
P1 finishes and releases its resources (3,2,2,2)
Total allocated: (2,4,4,2)
Total available: (3,2,4,2)

Safe Sequence of States

P2 gets (2,0,0,2)

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(0,0,0,0)	(0,0,0,0)
P2	(2,1,1,2)	(2,1,1,2)	(0,0,0,0)
P3	(1,1,3,1)	(0,0,0,0)	(0,0,0,0)
P4	(3,4,2,2)	(1,1,2,1)	(2,3,0,1)
P5	(2,4,1,2)	(1,2,1,1)	(1,2,0,3)

Total resources: (5,6,8,4)
Total allocated: (4,4,4,4)
Total available: (1,2,4,0)
P2 finishes and releases its resources (2,1,1,2)
Total allocated: (2,3,3,2)
Total available: (3,3,5,2)

Safe Sequence of States

P4 gets (2,3,0,1)

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(0,0,0,0)	(0,0,0,0)
P2	(2,1,1,2)	(0,0,0,0)	(0,0,0,0)
P3	(1,1,3,1)	(0,0,0,0)	(0,0,0,0)
P4	(3,4,2,2)	(3,4,2,2)	(0,0,0,0)
P5	(2,4,1,2)	(1,2,1,1)	(1,2,0,3)

Total resources: (5,6,8,4)
Total allocated: (4,6,3,3)
Total available: (1,0,5,1)
P4 finishes and releases its resources (3,4,2,2)
Total allocated: (1,2,1,1)
Total available: (4,4,7,3)

Safe Sequence of States

P5 gets (1,2,1,1)

Process	Max Claim	Allocation	Need
P1	(3,2,2,2)	(0,0,0,0)	(0,0,0,0)
P2	(2,1,1,2)	(0,0,0,0)	(0,0,0,0)
P3	(1,1,3,1)	(0,0,0,0)	(0,0,0,0)
P4	(3,4,2,2)	(0,0,0,0)	(0,0,0,0)
P5	(2,4,1,2)	(2,4,1,2)	(0,0,0,0)

Total resources: (5,6,8,4)
Total allocated: (2,4,1,2)
Total available: (3,2,7,2)
P5 finishes and releases its resources (2,4,1,2)
Total allocated: (0,0,0,0)
Total available: (2,4,1,2)
All resources are free, and all processes completed. Thus, the system is in a safe state.

Detecting Deadlock

- OS allocates resources when sufficient resources are available
  - Can lead to deadlock
  - Need detection algorithm
- Methods
  - Check on every allocation (early detection)
  - Detect cycles in RAG (very hard if multiples of each type of resource, otherwise graph theory)
    - How do you know when to invoke the algorithm?
  - ID deadlocked processes

Invoking Detection Algorithm

- Deadlock can only occur when a process requests a resource that can't immediately be granted
  - ID process that caused deadlock, and those in deadlock
  - Costly: all the overhead
- What else?
  - Invoke algorithm periodically
    - Then have to optimize the schedule
  - Deadlock reduces CPU utilization and system throughput, so invoke when CPU use is low

Recovering From Deadlock

- Easy:
  - KILL IT WITH FIRE! (or kill -9)
- Nah, too easy. Let's see some more complete options:
  - Terminate all processes in deadlock
  - Terminate deadlocked processes one by one, check, for resources again
  - Preempt resources one by one, rerun detection algorithm until no deadlock exists
    - How do we decide which resources/processes get preempted?
  - Roll back to a state pre-deadlock (when possible)