COMP124

Computer Systems

23 | Dining Philosophers Problem | Deadlock

Dr Stuart Thomason

Dining Philosophers Problem

- In the Producer-Consumer problem…
  - Threads have distinct roles (eg. manager and secretary)
  - The synchronisation environment involves a shared resource (eg. letter tray)

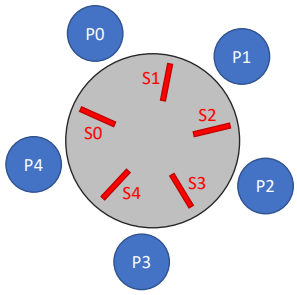
- In the Dining Philosophers problem…
  - Threads have identical roles
  - The synchronisation environment involves a shared resource

- This problem helps to visualise two common issues with concurrent systems
  - Deadlock
  - Starvation

- Operating system processes and resources are also affected by these issues

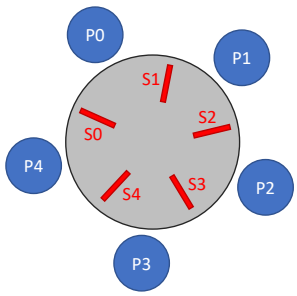
Dining Philosophers Problem

- There are N philosophers seated around a table
  - Each philosopher flips between eating and thinking (eat – think – eat – think – etc.)
  - Philosophers don't need any resources to think
  - But they need two chopsticks to eat
- There are only N chopsticks (one between each philosopher)
- If a philosopher wants to eat, they must wait for both neighbours to stop eating



Solution Approach

- Each philosopher has a unique index value in the set {0, 1, 2, 3, 4}
- And each chopstick has an index from the set {0, 1, 2, 3, 4}
- The philosopher with index K must wait for both chopsticks K and (K + 1) % 5
- When a philosopher eats, they block two other philosophers (their neighbours)
  - When P0 is eating, they are using sticks S0 and S1
  - Which means P1 and P4 cannot eat until P0 has stopped eating



Java Semaphore Solution

- We treat each chopstick as a shared resource with its own critical region

```
class Chopstick {
    private volatile boolean using = false;

    public synchronized void pickUp() {
        while(using) {
            try {
                wait();
            } catch(InterruptedException e) {}
        }
        using = true;
    }
    public synchronized void putDown() {
        using = false;
        notify();
    }
}
```

Java Philosopher Thread

- Each philosopher is a thread with a run() method (note that this is incomplete code)

```
public void run() {
    while(alive) {
        think();
        stick[i].pickUp();
        stick[(i + 1) % 5].pickUp();
        eat();
        stick[i].putDown();
        stick[(i + 1) % 5].putDown();
    }
}
```

non-atomic statements

- Each chopstick has a semaphore
  - Two philosophers cannot pick it up simultaneously
  - If a philosopher needs the stick but it's already in use, philosopher moves to wait set
  - When chopstick is put down, waiting threads are notified and it can be picked up

Starvation

- We cannot guarantee…
  - The order in which threads will be executed
  - When threads will be interrupted during execution

- Some possible scenarios…
  - If P0 and P2 pick up first, the others (P1, P3, P4) must wait and cannot eat
  - If P0 and P2 put down and immediately pick up again, the others cannot eat
  - If P0 and P2 take turns to eat, P1 still cannot eat

- Without proper controls on when threads run, some philosophers could starve

- In general, starvation occurs when one or more of the threads (or processes) in a concurrent system is denied access to the resources it needs

Deadlock

- Suppose each philosopher picks up the chopstick to their right simultaneously
  - Nobody can pick up the chopstick to their left
  - All philosophers move into the waiting state
  - Nobody ever releases their chopsticks, so all philosophers wait forever

- This is known as deadlock (or deadly embrace)

- In general, deadlock occurs in a concurrent system when each thread (or process) is waiting for one of the others to do something (or release something)

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"

Illogical law passed by the state of Kansas

Deadlock Solutions

- We could add some rules to the code to prevent deadlock
  - Allow only N – 1 philosophers to dine at the same time
  - Introduce asymmetry (so even indices pick up right chopstick first; and vice versa)
  - Insist that both chopsticks are picked up in one atomic operation (uninterruptable)

- All these solutions add extra complication to the code

- The dining philosophers problem helps us to understand the issues that can occur at the operating system level with processes and resources
  - Processes are the philosophers
  - Resources are the chopsticks

Resource Allocation & Deadlock

- The operating system must allocate and share resources sensibly
  - CPUs
  - Devices
  - Memory
  - Files

- In its simplest form, deadlock will occur in the following situation…
  - Process A is granted resource X and then requests resource Y
  - Process B is granted resource Y and then requests resource X
  - Both resources are non-shareable
  - Both resources are non-preemptible (ie. cannot be taken away from the process)

- Operating system can either prevent, detect, avoid or ignore deadlock

Deadlock Prevention

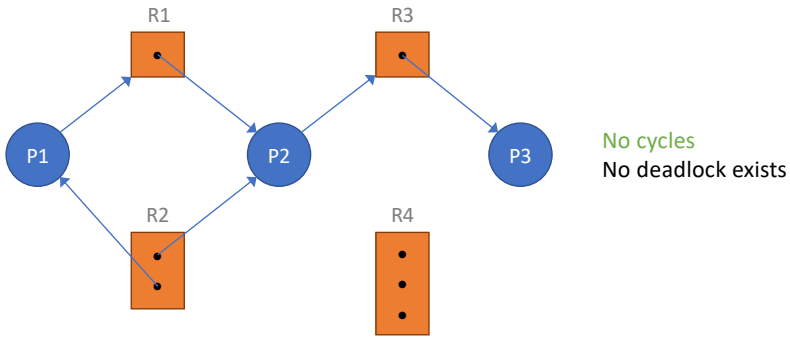
- We could force processes to claim all resources in one atomic operation
  - When process starts, immediately claim all resources it will need
  - Some resources will be under-utilised if not needed right away
  - Process might not know what future resources it will need as it executes

- We could number each resource and force processes to claim them in a specific order
  - Always request resource X before claiming resource Y
  - This avoids processes waiting for each other

- We could force processes to use only one resource at a time
  - Must release resource X before trying to claim resource Y
  - Prevents any process from needlessly hogging a resource
  - But this is very unrealistic in any modern operating system

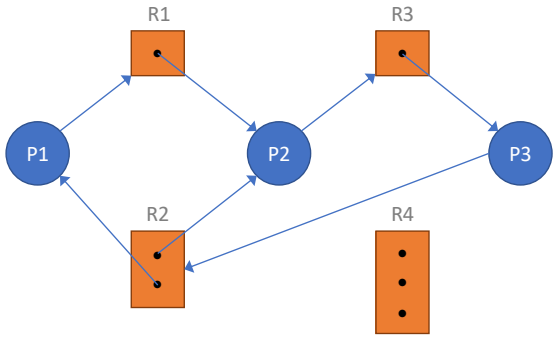
Resource Allocation Graphs

- We can draw a graph that models the processes and resources in a system
  - Processes represented as circles
  - Resource types represented as rectangles with resource instances as dots
  - Directed edges to show resource requests and assignments
    (arrow changes direction when a request is satisfied)



Resource Allocation Graphs

- Suppose P3 requests access to R2
  - This introduces a cycle (in fact, two cycles)
  - Indicates that deadlock is possible
  - But does not necessarily mean there is or will be deadlock
    (it will only happen if the request is granted at the 'wrong' time)



Cycle is present
There could be deadlock

Deadlock Detection and Recovery

- Resource allocation graph is monitored by the operating system
  - Very costly to do for every request
  - Might only check the graph every so often, such as when CPU usage deteriorates

- If a cycle is detected, the operating system could…
  - Take an existing resource away from a process to remove the cycle (preemption)
  - Kill one or more processes (a crude solution but it solves the problem)
  - Roll back to a previous state (requires OS to store checkpoints of process states)

- Each solution involves losing some processing time and even losing some data
  - Rollbacks will involve running the previous instructions again in a different order
  - Lots of memory will be used to store checkpoints
  - Calculations (including memory and disk writes) would need to be undone

Deadlock Avoidance

- Deadlock avoidance is very similar to deadlock prevention
  - Never grant access to a resource if it would lead to deadlock
  - Processes must declare up front what resources they will be using (might not know)
  - Processes will be forced to wait unnecessarily (because deadlock is rare)

- The banker's algorithm is a theoretical approach to deadlock avoidance
  - Devised by Dijkstra
  - Needs to know in advance what resources a process will request
  - Determines whether a static collection of processes is in a safe or unsafe state
  - Guarantees that processes will terminate but doesn't care how long it takes

- In practice, this algorithm is unsuitable because new processes arrive all the time in a dynamic system, and we cannot force processes to wait for an indefinite time

Ignoring Deadlock

- It is practically impossible to implement efficient deadlock detection or prevention

- The ostrich algorithm is a common approach in many modern systems
  - (The bird that sticks its head in the sand to pretend there are no problems)
  - More cost-effective to allow the problem to occur than try to prevent or avoid it
  - Detecting and preventing deadlock is expensive and adds complexity to code
  - In all practical situations, deadlock is relatively rare
  - The easiest way to recover from deadlock is to reboot the system when it happens
  - Rebooting every so often is cheaper than trying to prevent deadlock
  - This trades correctness for convenience

- Windows and Linux ignore deadlock and don't do anything to detect or prevent it
- The JVM also ignores deadlock when scheduling individual Java threads