COPADS

#5 - Advanced Threading

2023 - IFEOLUWATAYO IGE

OUTLINE

- Deadlocks
- Conditions for Deadlock
- Available solutions to Deadlock
- Starvation
- Scheduling
- Dining Philosopher problem

DEADLOCKS: INTRODUCTION

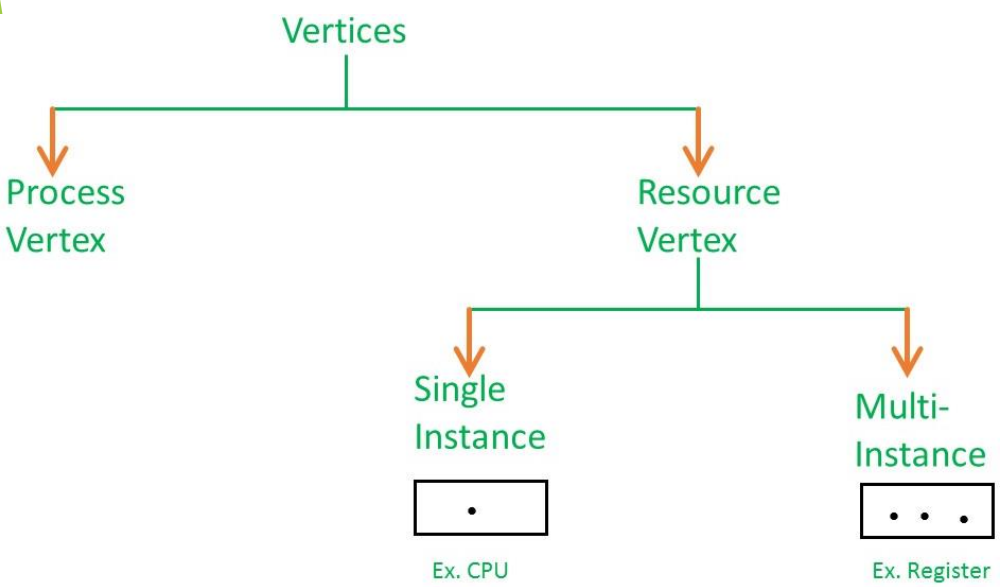
- For many applications, a process needs exclusive access to not one resource, but several.
- Suppose, two processes each want to print a scanned document on paper.
- Process *A* requests permission to use the scanner and it is granted.
- Process *B* is programmed differently, requests the printer first, and is granted it.
- Now *A* asks for the printer, but the request is suspended until *B* releases it. Unfortunately, instead of releasing the printer, *B* asks for the scanner. At this point, both processes are blocked and will remain so forever.
- This situation is called a **deadlock**.
- **A deadlock happens when two or more threads each wait for a resource held by the other, so neither can proceed**

RESOURCES: Categories

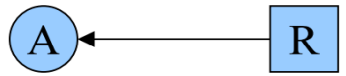
- A non-preemptable resource **cannot be taken from one process and given to another without side effects.** One obvious example is a printer: certainly, we would not want to take the printer away from one process and give it to another in the middle of a print job
- A resource is preemptable **if it can be taken away from the process that is holding it.** Memory is an example of a preemptable resource

Resource Allocation Graph (RAG)

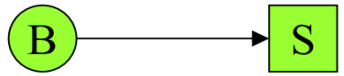
- This showcases how many resources are available, how many are allocated, and what is the request of each process (used to represent the state of a system in the form of a picture). RAG consists of the following:
  - **Process vertex** - Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
  - **Resource vertex** - Every resource will be represented as a resource vertex. It is also two type -
    - **Single instance type resource** - It represents a box, inside the box, there will be one dot
    - **Multi-resource instance type resource** - It also represents a box, inside the box, there will be many dots present.
  - **Assign Edge** - indicates a resource has been assigned to a process
  - **Request Edge** - It means in the future the process might want some resource to complete the execution



- Process A is holding resource R

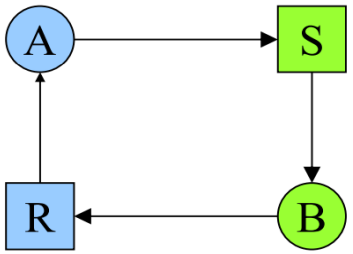


- Process B requests resource S



Example:

- A requests S while holding R

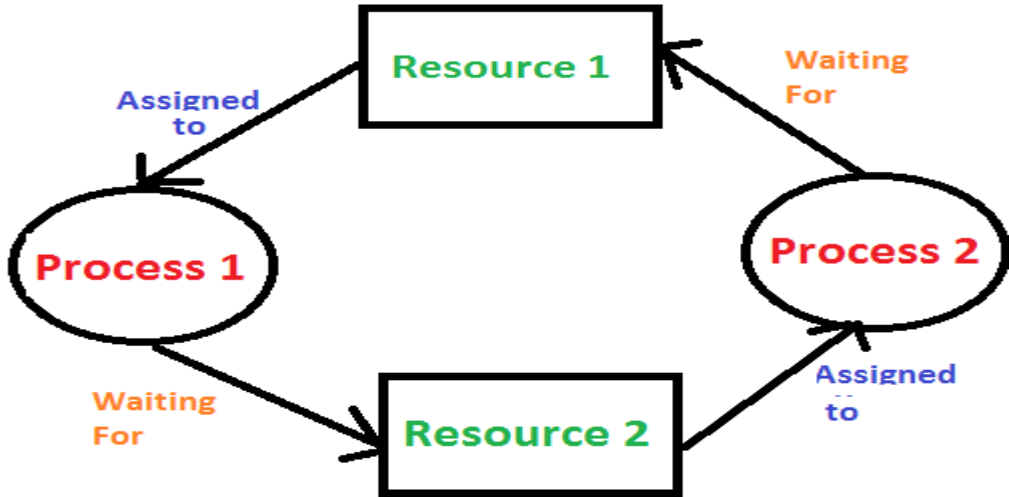


- B requests R while holding S
- A cycle in resource allocation graph ⇒ deadlock

Conditions for Deadlocks

The following conditions must **ALL** be present for a resource deadlock to occur.

1. **Mutual exclusion condition**. Resources are non-shareable (Only one process can use at a time)
2. **Hold-and-wait condition**. Processes currently holding resources that were granted earlier request new resources.
3. **No-preemption condition**. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. **Circular wait condition**. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain



The solution to Deadlock: 1. Detection And Recovery

- Allow deadlock to occur. Then recover through the following:

Recovery 1: Resource Preemption

- It may be possible to temporarily take a resource away from a process using the resource and give it to another process.
- In many cases, manual intervention may be required
- Negative Implication?

Recovery 2: Rollback

- If the programmers know that deadlocks are likely, processes can be **checkpointed** periodically.
- **Checkpointing (restoring)** a process means that its state is written to a file so that it can be restarted later.
- The checkpoint contains the 'memory page' and which resources are currently assigned to the process at those periods.
- A process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting at one of its earlier checkpoints.
- If the rolled-back process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery 3: Processes Termination

- Abort all the Deadlocked Processes. **Cons?**
- Abort one process at a time until deadlock is eliminated
  - The process to be killed is carefully chosen because it is holding resources that some processes in the cycle need.

Example:

- A process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked.
- A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.
- **This is better than just killing any of the two processes.**

The solution to Deadlock:2. Prevention

1. **No Mutual Exclusion**: If no resource was ever assigned exclusively to a single process, there will never be deadlocks.
   - **May cause race condition**
2. **No hold and Wait**: If we can prevent a process that holds a resource from waiting for more resources, there will never be deadlocks.
   - Require all processes to request all their resources before starting execution.
   - If everything is available, the process will be allocated whatever it needs and can run to completion.
   - If one or more resources are busy, nothing will be allocated and the process will just wait.
   - **Many processes do not know how many resources they will need until they have started running.**
   - **Resources will not be used optimally**

3. **Allow Preemption:** Resources previously granted can be forcibly taken away from a process

   Not acceptable for processes in the Kernel (OS).

4. **Avoid Circular wait:**
   - number all resources.
   - Require that processes request resources only in strictly increasing (or decreasing) order.
   - assign a priority number to each resource. A resource with a lower priority value cannot be requested by the process. It ensures that no process can demand a resource that is already in use by another. As a result, no cycle will form.

The solution to Deadlock: 3. Avoidance

- Dynamically grants a resource to a process if the resulting state is safe.
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.
- A system is in a **safe state** if there exists a sequence

  $<P_1, P_2, \ldots, P_n>$

- of ALL the processes in the system, such that for each P_i, the resources that P_i may still request can be satisfied by currently available resources
- If P_i's resource needs are not immediately available, then P_i can wait until all P_j*s* $(j<i)$ have completed execution
- When all P_j*s* run to completion, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_i +1 can obtain its needed resources, and so on.

- Negative implications?

Cons of Deadlock Avoidance

- Only works with a fixed number of resources and processes.
- Guarantees service time - **not** reasonable response time
  - the response time is the sum of the service time and wait time
- Needs advanced knowledge of maximum needs
- Unnecessary delays in avoiding unsafe states which may not lead to deadlock

A major difference between Deadlock Prevention and Avoidance?

- Summarily, deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions for deadlock cannot hold.
- WHILE
- Deadlock avoidance aims to prevent deadlocks by dynamically analyzing the resource allocation state of the system and deciding whether a resource request should be granted or not.

What is Starvation?

- Suppose that three processes (P1, P2, P3) each require periodic access to resource R. P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource.
- When P1 exits its critical section, P2 or P3 should be allowed access to R. If the OS grants access to P3, P1 again requires access before P3 completes its critical section.
- The OS grants access to P1 after P3 has finished, and subsequently grants access to P1 and P3, then P2 may indefinitely be denied access to the resource.
- A thread is starved when it should be allowed to run but never does due to **priority scheduling.**
- Some priority is assigned to every thread; based on that priority, the CPU is allocated to the highest priority one, which always gets to run first.
- CPU scheduler must deal with **processor starvation** (a process of determining which process will own CPU for execution while another process waits)
- **Resource Starvation (one or more processes being starved of resources)** can be solved by adding resource(s)

Solution to Starvation: **Aging**

- Aging is a scheduling technique that gradually increases the priority of a task and its waiting time in the ready queue.
- Aging is used to ensure that jobs with lower priority will eventually complete their execution.
- For example, Suppose process P has priority 75 at 0 milliseconds.
- Then we can reduce the priority number of the process P by 1 every 5 ms (you can use any time quantum)
- process P will have a priority of 74 after 5 ms.
- After 10 milliseconds, process P's priority will change to 73, and the procedure will proceed.
- Process P will eventually be given the CPU to execute when the priority number approaches 0 after a specific amount of time. At that point, process P will become a high-priority process.

SCHEDULING

- There may be a need to schedule how processes and threads access resources, especially if these processes are in a ready state.
- A part of the operating system called the **scheduler** makes this choice using a **scheduling algorithm** because process switch is expensive (the current state of the process has to be saved), hence the need to make efficient use of the CPU
- **There may be a need for scheduling when:**
  - a process blocks waiting for I/O, on a semaphore, or for some other reason, another process has to be selected to run
  - some other process must be chosen from the set of ready processes when a process releases the CPU
  - If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

Scheduling Algorithms

- Categories:
  - Preemptive algorithms
    1. **Round-Robin**- each process is equal and allotted a time interval (**quantum)**
    2. **Priority scheduling-** each process is assigned a priority, the process with the highest priority runs first
    3. **Shortest Remaining Time Next-** If a new job needs less burst time to finish than the current process on the CPU, the current process is suspended and the new job is started.
  - Non-preemptive algorithms
    1. First come first served (FCFS)
    2. Shortest Job First

Environments where scheduling algorithms are required

- **Batch systems**: Periodic tasks, payroll, inventory.
  - Non-preemptive algorithms, or preemptive algorithms with long time periods
- **Interactive systems**: Servers.
  - A preemptive algorithm is essential to keep one process from hogging the CPU and denying service to the others
- **Real-time systems**: Patient monitoring systems, Autopilot in aircraft.
  - Absolute deadlines that must be met. Hence always require a preemptive algorithm
  - periodic (occurs at regular intervals) or aperiodic (occurs unpredictably)

Approaches to Thread Scheduling on Multiprocessor Systems

- Load Sharing
- Gang Scheduling
- Dedicated Processor Assignment
- Dynamic scheduling

Load Sharing

- Processes are not assigned to any particular processor.
- Load is distributed evenly across the processors
- Only one global -central queue is required

- Cons:
  - Central queue occupies more memory
  - Preemptive threads are unlikely to resume execution on the same processor- which will make caching less efficient

Gang Scheduling

- Simultaneous scheduling of threads that make up a single process
- Less context switching

- **Cons:**
- Useful for parallel applications
- Useful for medium-grained to fine-grained parallel applications whose performance severely degrades when any part of the application is not running while other parts are ready to run

Dedicated Processor Assignment

- A process is permanently assigned to one processor from activation until its completion,
- A dedicated short-term queue is maintained for each processor.
- If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle

- Con:
- Underutilization of resources.

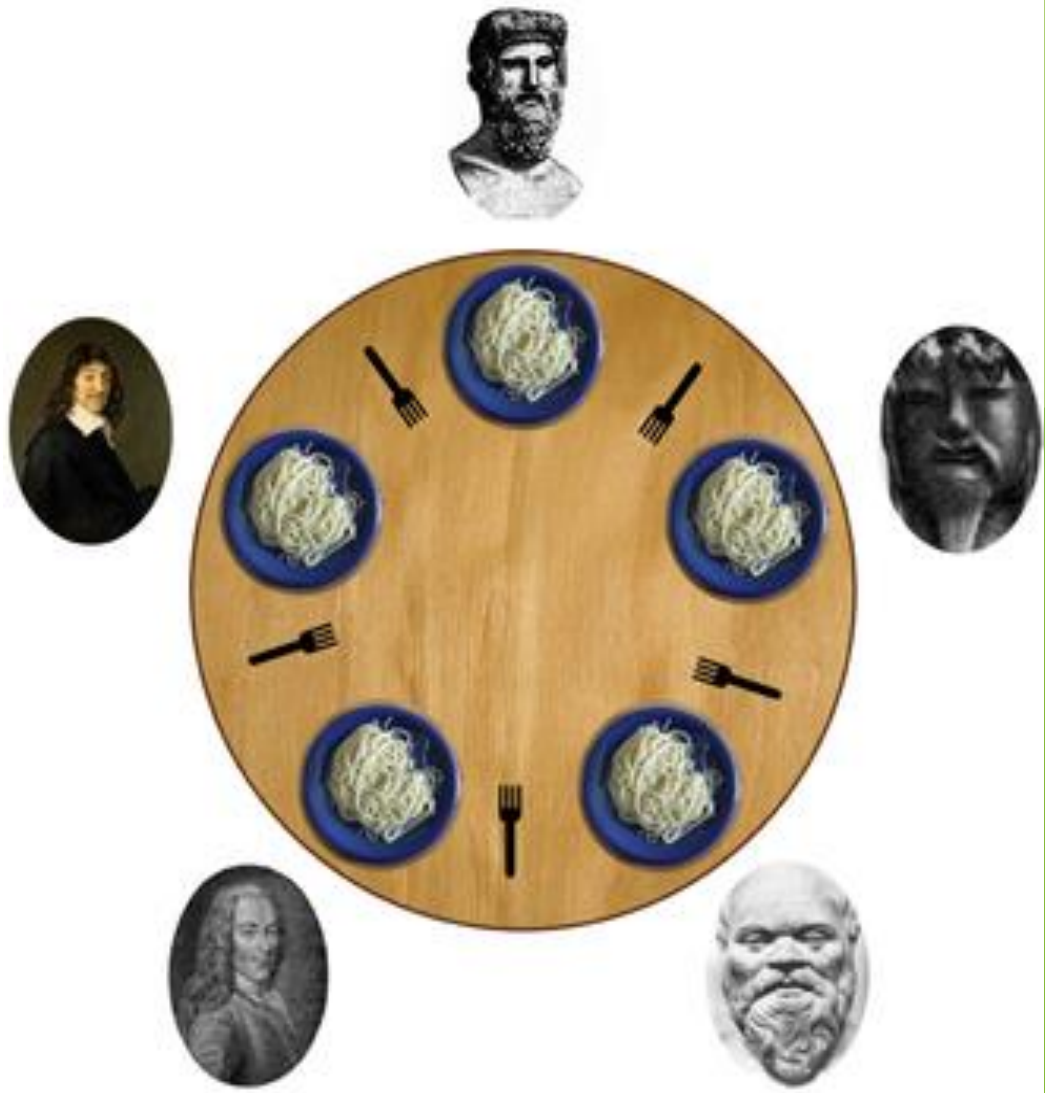
Dynamic Scheduling

- Scan the current queue of unsatisfied requests for processors. Assign a single processor to each job in the queue that currently has no processors
- If there are idle processors, use them to satisfy the request.

- C#- **TaskScheduler** class in System.Threading.Tasks namespace for tasks, local queues and global queues
- https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler?view=net-7.0

Dining Philosophers' Problem

- Created by 'Dijkstra and Hoare' to illustrate problems with multi-threaded programming
- We have bowls of pasta on the table
- Each philosopher must alternately think and eat (not at the same time)
- Each philosopher needs 2 forks to eat (and can only access the one on their immediate left and right)
- The forks are picked up one at a time and the philosophers cannot talk
- Assume that no philosopher can know when others may want to eat or think
- If each philosopher were to pick up their left fork at (roughly) the same time, we end up with a deadlock



Solutions to the Dining Philosopher Problem

- The maximum number of philosophers on the table should not exceed four.
- Only when both forks (left and right) are available at the same moment can a philosopher be permitted to choose their forks.
- Each fork might be viewed as a shared resource secured by semaphore. Before starting to eat, each philosopher locks his left and right fork. If both locks are successfully purchased, the philosopher will now have two locks and access to food.
- If the philosopher is in an even position, he/she should choose the right fork first, followed by the left, and in an odd position, the left fork should be chosen first, followed by the right.

Solutions to the Dining Philosopher Problem

- There are three states of the philosopher: **THINKING, HUNGRY, and EATING.**

- Each philosopher is represented :

```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

```
repeat
      if left fork is not available
      then
          block(Pi);
      lift left fork;
      if right fork is not available
      then
          block(Pi);
      lift right fork;
      {eat}
      put down both forks
      if left neighbor is waiting for his right fork
      then
      activate (left neighbor);
      if right neighbor is waiting for his left fork
          then
      activate(right neighbor);
         {think}
forever
```

An Implementation in C#

- Please see the Dining Philosopher folder for C# implementation.
- https://www.codeproject.com/Articles/1239410/Dining-Philosophers-Problem