Multi-threading Processes

Deadlock, Livelock, Race Conditions and Mutex

Deadlock

- Processes / Threads needs various resources in order to complete their tasks

- A process must request a resource from the Operating System before using it, and release it when it is finished using it
  - **Request -** If the request cannot be granted immediately immediately (for example the resource is being used by another process), then the requesting process must wait
  - **Use -** the process can operate on the resource
  - **Release cycle** – the process releases the resource

- If a process requests a resource that is already allocated it joins a waiting queue for that resource

Deadlock

- In a multiprogramming environment **several processes** may compete for a **finite number of resources.**

- A process requests resources; and if the resources are **not available at that time the process enters a waiting state.**

- **What is Deadlock?**
- A situation in which two or more processes are unable to proceed because each is waiting for one the others to do something.

Deadlock Example

- Process A intends to write to files R and S
- Process B intends to do likewise
  - Process A opens file R for write (I/O Event)
  - Process B opens file S for write (I/O Event)
  - Process A attempts to open file S for write but is informed that S is already open, so it waits
  - Process B attempts to open file R for write but is informed that R is already open, so it waits
- **Both processes are now waiting for files that each other have**
- **This is called Deadlock!**

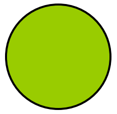
Deadlock, continued

- A good illustration of deadlock can be drawn from a law passed by the Kansas () U.S.A. law early in the 20th century. It said, in part:

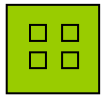
"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"

Resource-Allocation Graph

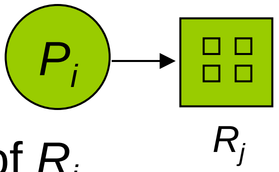
- Process



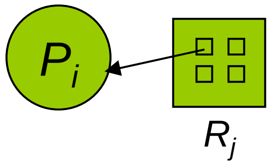
- Resource Type with 4 instances



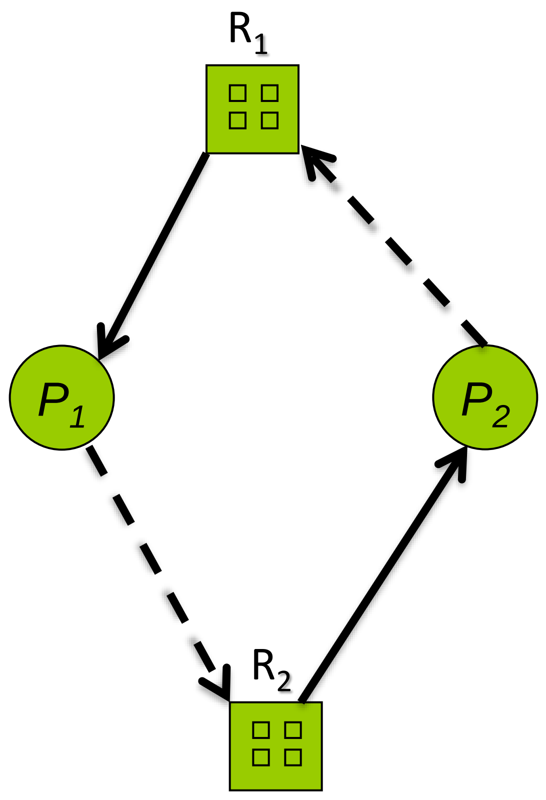
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of Deadlock



- Process P_1 holds resource R_1 and is requesting access to resource R_2
- Process P_2 holds resource R_2 and is requesting access to resource R_1
- **We have deadlock!** Neither process is able to proceed.
- Unless one of the processes detects the situation and is able to withdraw it's request for a resources and release the resource allocated to it, none will ever be able to run

Livelock

- **What is Livelock?**
- A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work

- A real-world example of livelock occurs when two people meet in a narrow corridor
  - Each tries to be polite by moving aside to let the other pass.
  - but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Livelock

- As with deadlock, livelocked threads are **unable to make further progress**.

- However, the **threads are not blocked** — they are simply **too busy responding to each other to resume work**.

- Livelock is a risk with some algorithms that detect and recover from deadlock.
  - If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Race Condition

- **What is a race condition?**
- A race condition occurs when two or more threads can access shared data and they try to change it at the same time.
- Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.
- Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.
- In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time.

Source: http://stackoverflow.com/questions/34510/what-is-a-race-condition

Mutual Exclusion

- **What is a mutual exclusion?**
- In computer science, mutual exclusion is a property of concurrency control, which has the purpose of preventing race conditions;

- **What is a mutual exclusion implemented in practice?**
- There are many different techniques. However the most common technique is to use a locking mechanism, formally known as a mutex.

Mutex

- **What is a Mutex?**
- A Mutex is a mutually exclusive flag.
- The point of a mutex is to synchronize two threads.
  - When you have two threads attempting to access a single resource/update data, the general pattern is to have the first process attempting access to set the mutex before entering the code.
  - When the second process attempts access, it sees that the mutex is set and waits until the first process is complete (and has un-set the mutex).
  - When the mutex is unset, the second process may proceed.