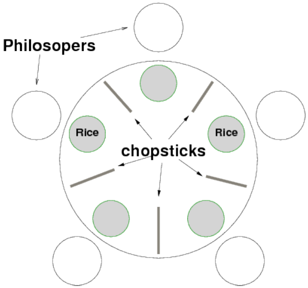
(Slide 12 corrected after class)

Week 9-2: Deadlock

CS211: Programming and Operating Systems

Thursday, 12 March 2020



Today, in CS211, . . .

Recall: Concurrency

- ***Cooperating*** processes are one that can affect each others execution on a system.
  Threads are an important example of this: the share program code and data.
- A ***Race Condition*** (or "data race") holds if the order in which threads execute determines their output.
- A **critical section** is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be (safely) concurrently executed by more than one thread.
- A set of operations is **"atomic"** if, once started, they cannot be interrupted.
- A **Semaphore** S is an integer variable that can only be accessed via one of two operations: **Test/*sem_wait*** $P(S)$, and **Increment/*sem_post*** $V(S)$.

Recall: Concurrency

A more detailed explanation of semaphores

A semaphore indicates is a particular resourse is available; or not. It has

two operations

(1) TEST which

(a) checks if the resourse is available.

(b) If it is, sets a flag to show it has taken the resourse

(c) if not, wait until it is.

(2) INCREMENT : Release the resourse.

Recall: Concurrency

(see Lab 6).

Implementing semaphores with `pipe()` in C.

A pipe is a FIFO Queue, represented as an integer array with 2 entries:

(0) The address of the "read" end

(1) The address of the "write" end.

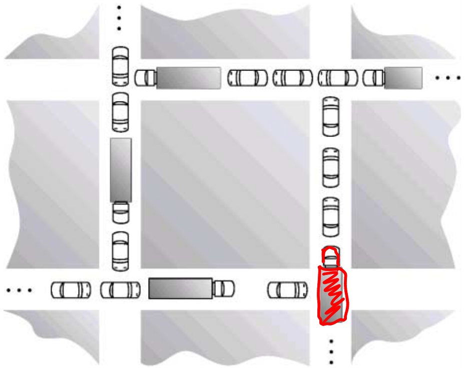
If one process writes to the pipe, another can read from it.

Important: if the pipe is empty a process that tries to read will wait until something is written. So we can use this as a semaphore:

TEST = "Read from pipo"

INCREMEW = "write to pipe".

Deadlock and Starvation



Deadlock is when two or more procs are waiting indefinitely for an event that can only be caused by one of the waiting processes. E.g., all are stuck in the `wait()` loop of the $P()$ function.

Deadlock and Starvation

Deadlock can arise if four conditions hold simultaneously

1. ***Mutual exclusion***: only one process can have access to a particular resource at any given time.
2. ***Hold and wait***: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. ***No preemption***: a resource can only be released voluntarily by the process holding it, after that process has completed its task.
4. ***Circular wait***: there exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ of waiting processes such that
   - P_0 is waiting for a resource that is held by P_1,
   - P_1 is waiting for a resource that is held by P_2, ...,
   - P_{n-1}, is waiting for a resource that is held by P_n which in turn is waiting for P_0.

Resource Allocation Graph

Deadlocks may be described in using a directed graph called a ***resource allocation graph***.

This graph has two sets of **vertices:**

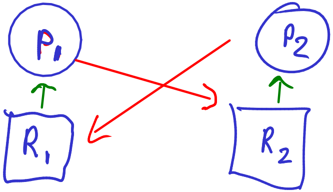
- **Processes:** $P_0, P_1, \ldots, P_n$ and
- **Resources:** $R_0, R_1, \ldots, R_m$

and ***Edges***

- from P_j to R_k is process j as requested resource k but not yet been allocated it,
- from R_k to P_k is process j as been allocated resource k and not yet released it.

Example:

Deadlocked (cycle present)



P_1 has R_1 & is waiting for R_2

P_2 has R_2 & is waiting for R_1.

Resource Allocation Graph

Deadlocks may be described in using a directed graph called a ***resource allocation graph***.

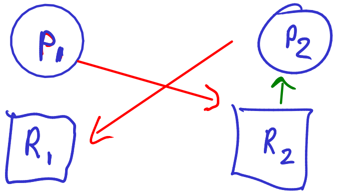
This graph has two sets of **vertices:**

- **Processes:** $P_0, P_1, \dots, P_n$ and
- **Resources:** $R_0, R_1, \dots, R_m$

and ***Edges***

- from P_j to R_k is process j as requested resource k but not yet been allocated it,
- from R_k to P_k is process j as been allocated resource k and not yet released it.

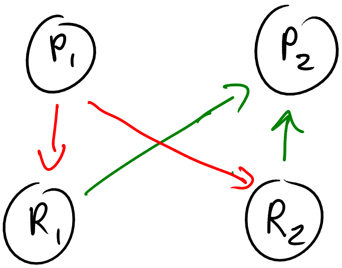
Example:



Here P_2 holds R_2 and wants R_1 which is available: no deadlock.

Resource Allocation Graph

- If there are no cycles, there is no deadlock,
- If there is deadlock, there must be a cycle
- If there is a cycle, there ***may*** be deadlock
- If each resource has only one instance, and there is a cycle, then there is deadlock



no deadlock

The Dining Philosophers Problem

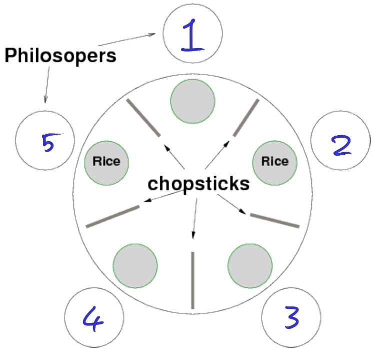
(See also Section 31.6 of the textbook)

Starvation occurs when a particular process is always waiting for a particular semaphore to become available.

The ideas of **Deadlock** and **Starvation** are exhibited in the classic synchronization problem: ***The Dining Philosophers Problem***.

- There are five philosophers seated at a round table.
- Each has a bowl of rice in front of them and a chopstick to their left and right.
- They spend their day alternating between eating and thinking.
- However there are only five chopsticks...

The Dining Philosophers Problem



If a philosopher is hungry, she will try to pick up the chopstick to the left and then the chopstick to the right. If she manages to do this they will eat for a while before putting down both and thinking for a while. However, if she it picks up one, she will not let go of it until she can picks up the second and eat.

Suppose each of the picks up the fork to their left. No forks remain on the table so we reach a state of deadlock.
The challenge is to find a solution so that

- **Deadlock** does not occur.
- neither does **starvation** where one philosopher never gets to eat.

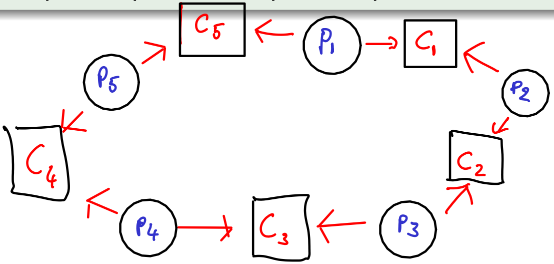
The Dining Philosophers Problem [Corrected after class]

Example (Taken from 1718-CS211 Exam)

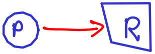
In the dining philosophers problem, each philosopher wants to pick up the 2 forks beside him/her so that they can eat. Suppose we have 5 such philosophers and

- at time $t = 0$: nobody has picked up any fork
- at time $t = 1$: Philosophers 1, 2 and 3 have picked up the forks to their right, philosopher 4 has not picked up any fork and philosopher 5 has picked up two forks.

$t = 0$



Recall



P has requested (if waiting for) R

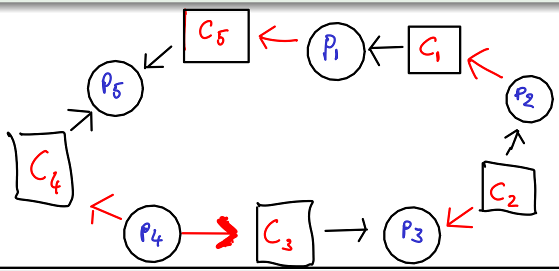
The Dining Philosophers Problem [Corrected after class]

Example (Taken from 1718-CS211 Exam)

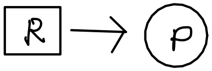
In the dining philosophers problem, each philosopher wants to pick up the 2 forks beside him/her so that they can eat. Suppose we have 5 such philosophers and

- at time $t = 0$: nobody has picked up any fork
- at time $t = 1$: Philosophers 1, 2 and 3 have picked up the forks to their ~~right~~ left, philosopher 4 has not picked up any fork and philosopher 5 has picked up two forks.

$t = 1$



Recall



R has been allocated to p

The Dining Philosophers Problem Deadlock handling

In general operating systems take one of three approaches to deal with deadlock:

1. Ensure that the system will never enter a deadlock state.
2. Allow the system to enter a deadlock state and then recover.

In Case 1, there are two possibilities:

- ***Prevention:*** we ensure at least one of the four necessary conditions never hold.
- ***Deadlock avoidance:*** where the OS uses ***a priori*** information about the procs the devise an algorithm to circumvent deadlock.

In Case 2, the OS must have mechanisms for first detecting deadlock and then dealing with it.

Exercises

Exercise (10.1)

Draw the resource allocation graph for the scenario where all philosophers pick up the fork/chopstick to their left.

Exercise (10.2)

A system has $m = 4$ identical resources, and $n = 3$ processes, P_1, P_2 and P_3, which make a request for 1, 2, and 3 resources, respectively. Draw the resource allocation graph for the scenario. Can the system reach deadlock?