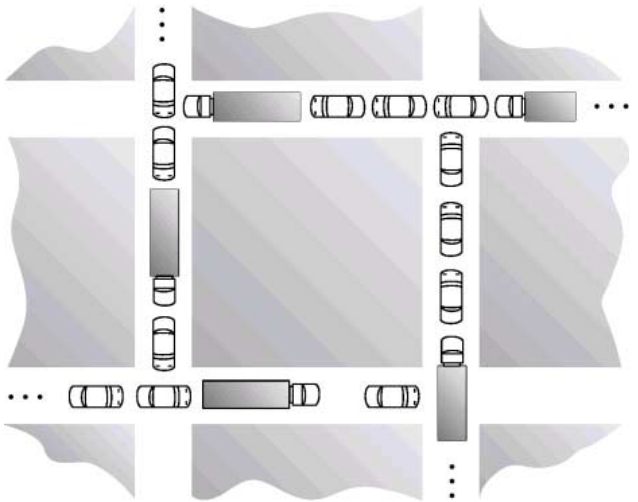
What will we learn today?

Week 10: Resource Allocation Graphs and Safe States

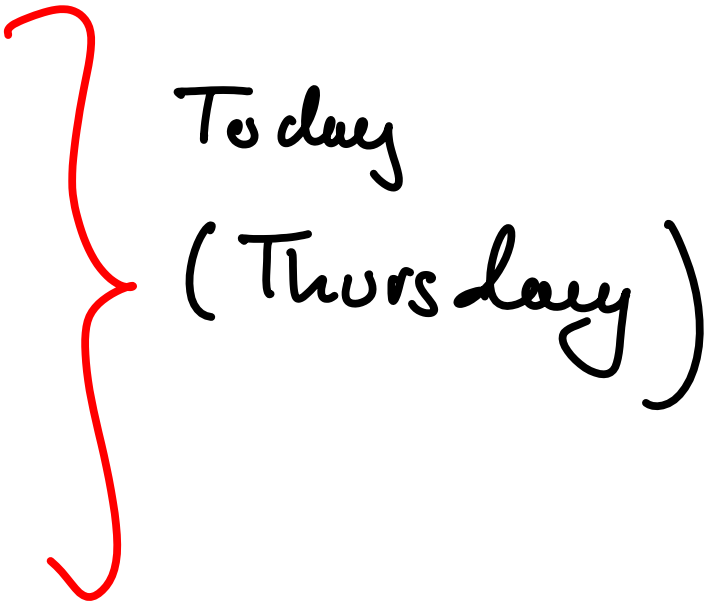
CS211: Programming and Operating Systems

Wednesday, 18 March
and Thursday 19 March, 2020



Today, in CS211, . . .

1. Plans for the rest of the semester
2. Recall: Concurrency
3. Recall: Deadlock
4. Recall: The Dining Philosophers Problem
5. Deadlock handling
   - Solutions
6. Deadlock Detection
   - Recall: Resource Allocation Graph
7. Deadlock Avoidance
   - Safe states and sequences
8. Exercise



Plans for the rest of the semester

Even with the NUIG campus closed, we are going to try to progress with **CS211** as much "normality" as possible. In particular,

1. In an effort to keep in touch, emails from me will be more frequent than before (that is, I won't limit myself to one message per week).
2. Lectures will take place at the usual times, but online, starting Wednesday (18 March) at 3pm. You can join it by logging on to Blackboard and selecting "Virtual Classroom".
   Same plan for Thurday's 1pm lecture.
3. All classes will be recorded, so if you can't join at the time, you can still review them later. Access them through blackboard: "Virtual Classroom"... "Recordings"

Plans for the rest of the semester

On Friday. we will try to do this with our lab. However, we *may* be a little more complicated, since one-to-one help is a little bit trickier. But, we will try it!

. . . . . .

By tomorrow (Thursday) I will have figured out a plan for the assessment for the rest of the Semester. At the moment, my idea is

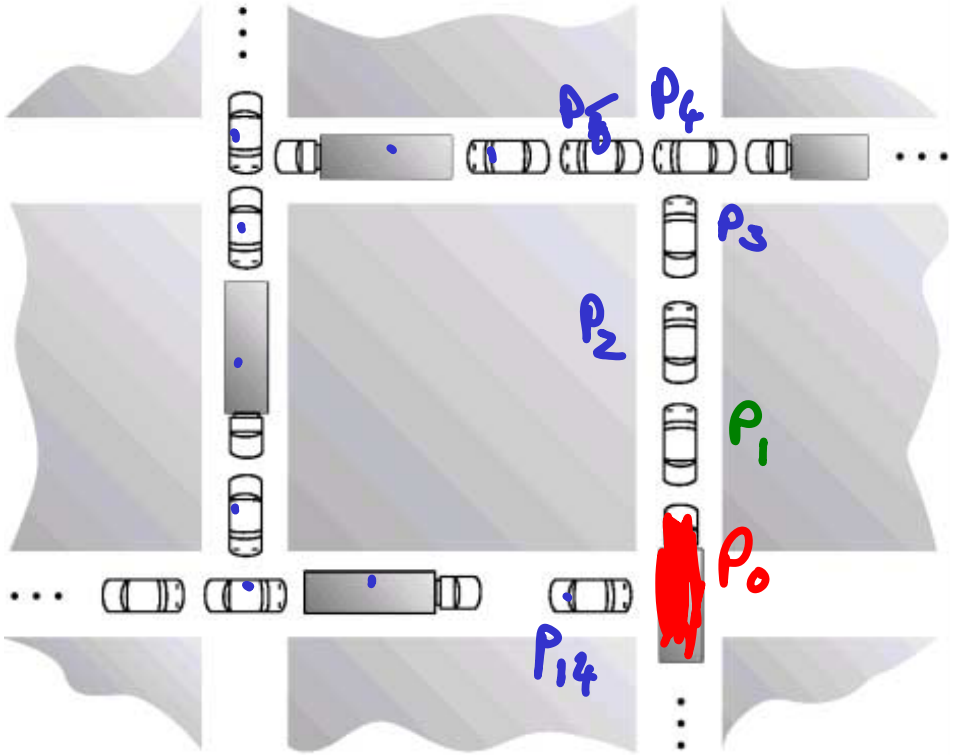
- Complete one more lab assignment (so one fewer than planned);
- Complete a “take-home” assignment (date TBD);
- Complete a on-line exam (date TBD)

Discuss....

Recall: Concurrency → several thing happening at once.

- ***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: Deadlock



Deadlock is when two or more procs are waiting indefinitely for an event that can only be caused by one of the waiting processes.

"Processes" are cars/trucks.

"Resources": road space.

"Circular wait".

Recall: Deadlock

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.

Recall: The Dining Philosophers Problem

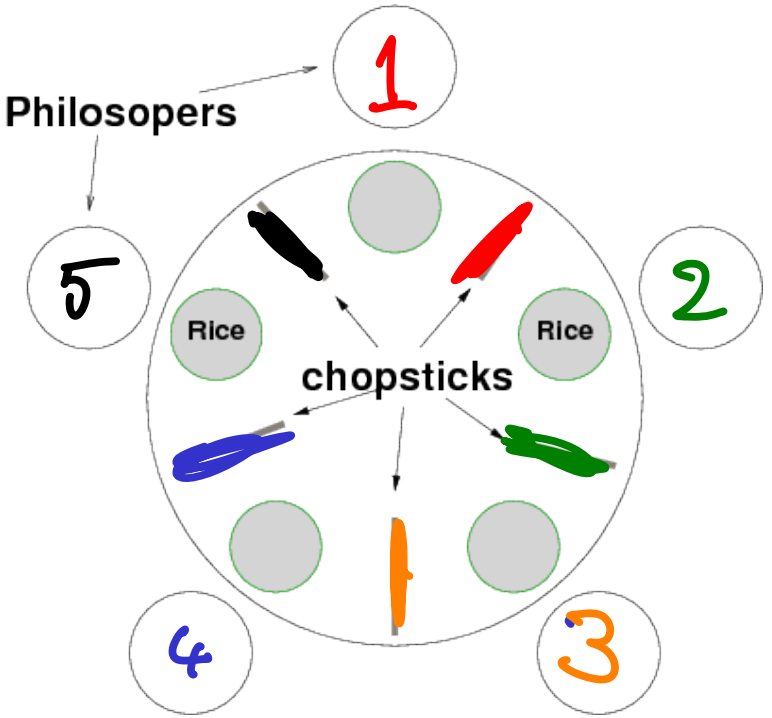
(See also Section 31.6 of the textbook)

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.
- There are **five chopsticks** on the table, each between a pair of philosophers.
- They spend their day alternating between eating and thinking.
- But to eat, they need both the chopsticks to their left and right...

Recall: 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 chopstick to their left. No chopsticks 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.

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. (More about this in a minute).

The Dining Philosophers Problem can be represented as follows.

forever

From section 31.6 of the textbook

```
1 while (1) {
    think();
3   get_chopskicks();
    eat();
5   put_chopsticks();
  }
```

what a philosopher is doing.

Here the key is two write versions of `get_chopsticks()` and `put_chopsticks()` which result in no deadlock, none starves (also, ideally, concurrency is optimised).
Within these functions we will have others:

```
2 int left(int p) { return p; }
  int right(int p) { return (p + 1) % 5; }
```

Eg right(4) = (4+1)%5 = 0.

Next we will use semaphores to control access to the chopsticks. Let us suppose there are five:

will explain next week.

```
sem_t chopsticks[5].
```

When we can write the get/put functions as

```
1  void get_chopsticks() {
     sem_wait(chopsticks[left(p)]);
3    sem_wait(chopsticks[right(p)]);
   }

   void putchopsticks() {
7    sem_post(chopsticks[left(p)]);
     sem_post(chopsticks[right(p)]);
9  }
```

Here "wait" means "wait until".

"post" = "put down"

Implemented like this, we get the expected deadlock. We'll now try to get a solution.

The most famous solution, which was proposed by the problem's inventor, Edsger Dijkstra, is simply to have one philosopher attempt to pick up the chopstick to their right first, while everyone else tries to get the one on their left first.

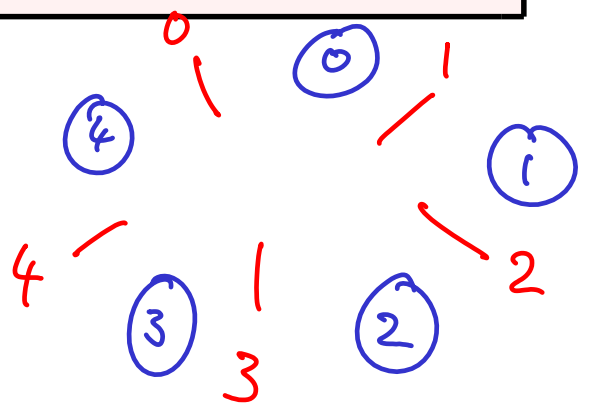
```
1   void getchopsticks() {
      if (p == 4) {
3       sem_wait(chopsticks[right(p)]);
        sem_wait(chopsticks[left(p)]);
5     } else {
        sem_wait(chopsticks[left(p)]);
7       sem_wait(chopsticks[right(p)]);
      }
9   }
```

p is the number of the Phisopher

p = 0, 1, 2, 3, 4.

Can you convince yourself this works?

(Thought) Exercise.



Finished here Wednesday

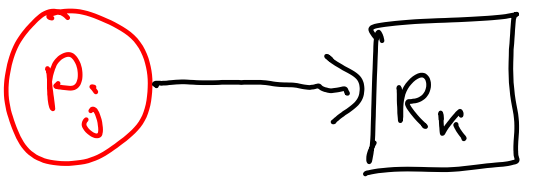
Recall: Resource Allocation Graph

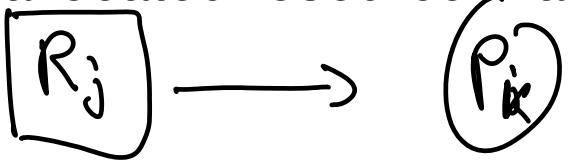
Competition for resouces can be described using a directed graph called a ***resource allocation graph***.
This graph has two sets of **vertices:**

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

and ***Edges***

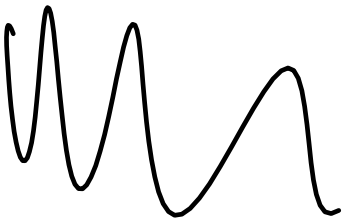
- from P_j to R_k is process j as requested resource k but not yet been allocated it,
- from R_k to P_j is process j as been allocated resource k and not yet released it.





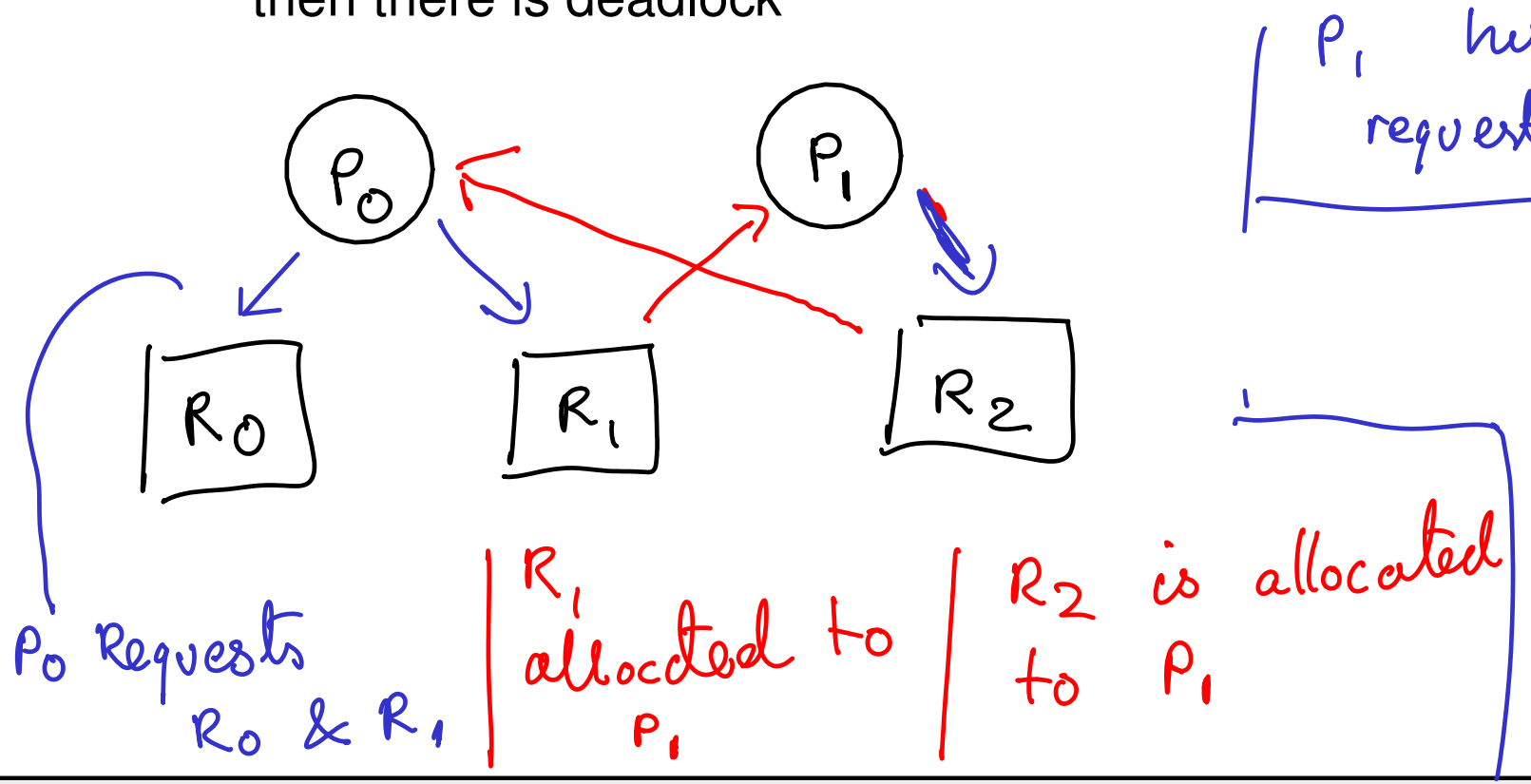
means R_j has been allocated to P_K.

Example:



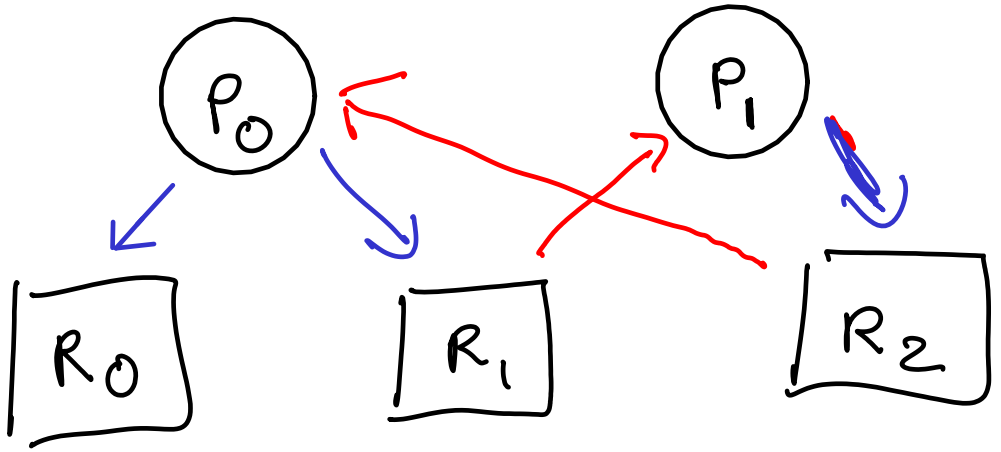
Deadlock Detection Recall: Resource Allocation Graph

- 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



Deadlock Detection Recall: 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



This is deadlocked, because there is a cycle:

$P_0 \rightarrow R_1 \rightarrow P_1 \rightarrow R_2$

Deadlock Detection Recall: Resource Allocation Graph

Detection of deadlock is a important, but not very easy problem. Using ideas like the Resource Allocation Graph, the system's needs can be represented mathematically, and then a deadlock state checked for.

Idea

Suppose that a system has n processes, and a total of m resources that it can allocate. Resources can only be requested or released one at a time.

Then the system is *Deadlock free* if the following conditions hold:

(i) each process requires at least 1 resource and at most m.

(ii) the sum of all their requirements is less than $m + n$.

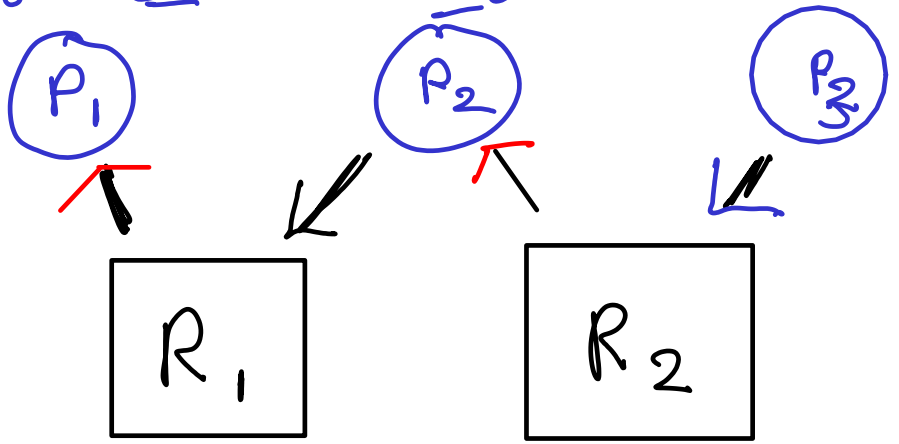
Example

Use a resource allocation graph to show that

- if there are $n = 3$ processes, and
- $m = 2$ resources
- Resources can only be requested or released one at a time.
- (i) each process requires at least 1 resource and at most 2.
  (ii) the sum of all their requirements is at most **4** (i.e., less than $m + n = 5$).

Then there is no deadlock.

P_1 has R_1
P_2 requests R_1
P_2 has R_2
P_3 requests R_2.



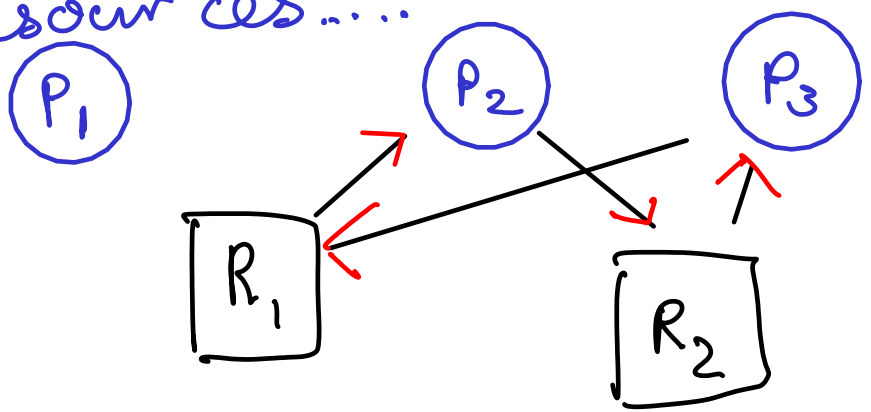
This is not Deadlocked. why? P_1 can proceed. Then it will release R_1. P_2 can then proceed.

Example

Use a resource allocation graph to show that

- if there are $n = 3$ processes, and
- $m = 2$ resources
- Resources can only be requested or released one at a time.
- (i) each process requires at least 1 resource and at most 2.
  (ii) the sum of all their requirements is at most **4** (i.e., less than $m + n = 5$).

Suppose we change this so one process requests zero resources....



Deadlock!

Deadlock Avoidance

One possible solution to the deadlock problem, but which requires extra information is represented as the **Banker's Algorithm**. IN particular we need to know:

- The number of resources the system has;
- The number currently allocated to each process;
- **The maximum that any process might request**.

With this information, it should be possible to ensure that a circular wait condition does not hold. To understand this, we need to concepts of **safe states** and **safe sequences**.

- A **resources-allocation state** is the number of available and allocated resources, and the maximum demands of processes.
- A state is **safe** if the system can allocated resources to each process, and still avoid deadlock.
- A **safe sequence** is a sequence of processes such that their resource requests can be granted, in order, with no process having to wait indefinitely.

It is, perhaps, easiest explained through an example.

Example (From old exam paper)

Suppose we have a system with

- three resource types, A, B and C. There are in total 10 instances of type A, 5 instances of type B and 7 instances of type C.
- 5 processes, P_0 , $P_1, \ldots, P_4$.

At a given point in time the allocations, maximal demands and availability of each of the resources is given below. Determine if it is a safe state, and give a safe sequence.

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_0	7	5	3	0	1	0
P_1	3	2	2	2	0	0
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

Note we cannot make all allocations at once. Eg, there are a total of 25 requests for A, but just 10 instances of A.

	Total Reqs			Current Alloc			Needs		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2 ← 1st
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1 ← 2nd.
P_4	4	3	3	0	0	2	4	3	1

At the starts we have allocated 7×A, 2×B & 5×C.
Write this as (7, 2, 5). So we have
(10, 5, 7) − (7, 2, 5) = (3, 3, 2) available.
Allocate (1, 2, 2) to P_1, Leaving (3, 3 2) − (1, 2, 2) = (2, 1, 0).
P_1 will finish, and release these, bring us back to (5, 3, 2).
Now let P_3 run. when it is finished, ...

	Total Reqs			Current Alloc		
	A	*B*	*C*	*A*	*B*	*C*
P_0	7	5	3	0	1	0
P_1	3	2	2	2	0	0
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

We did have (5, 3, 2), we allocated (0, 1, 1) to P_3, leaving us with (5, 2, 2). But when P_3 finishes we'll have (7, 4, 4). Now others can run in any order. So

① This is a safe state.

② Safe seq is $P_1 \rightarrow P_3 \rightarrow P_0 \rightarrow P_2 \rightarrow P_4$.

Example (From 2017/2018 CS211 exam)

Suppose we have a system with three resource types:

9 instances of A, 3 instances of B and 6 instances of C.

Consider four processes P_1, P_2 , P_3 , P_4 , with, at a given point in time, current allocations and total requirements given by

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_1	3	2	2	1	0	0
P_2	6	1	3	6	1	2
P_3	3	1	4	2	1	1
P_4	4	2	2	0	0	2

Is this a safe state? If it is, give a safe sequence?

9 instances of A, 3 instances of B and 6 instances of C.

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_1	3	2	2	1	0	0
P_2	6	1	3	6	1	2
P_3	3	1	4	2	1	1
P_4	4	2	2	0	0	2

Finished here... will return to this next week.

Exercise

Exercise (11.1)

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?

Exercise (11.2)

A system has $m = 4$ identical resources, and $n = 3$ processes. The processes make a total request for 6 resources. If one process makes no request, can the system reach deadlock?

Exercise

Exercise (11.3)

Recall an example where we had three resource types, A, B and C, and 5 processes, P_0, ..., P_4. We have

10 instances of type A, 5 instances of type B, and 7 instances of type C.

At a given point in time the allocations, maximal demands and availability of each of the resources is given as follows (take notes). Determine if it is a safe state, and give a safe sequence.

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_0	7	5	3	0	1	0
P_1	4	2	2	**3**	0	**2**
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

Could we make an additional initial allocation of $(0, 2, 0)$ to P_0? That is, would that lead to a safe state?