Liveness:
Deadlock & Livelock

https://youtu.be/fX3YE5mSdZk

Liveness

If you'll remember there are two aspects to correctness of a concurrent system:

Safety Property – This means that nothing "bad" will happen such as corrupted data or incorrect output. The ATM problem losing transaction would be a violation of the safety property.

Liveness Property – This means that something "good" will eventually happen. If a program locks-up and can never produce a result it is a violation of the liveness property.

This is what we are going to focus on today.

Liveness

Deadlock: A set of processes S is deadlocked is every process in S is blocked on an event that can only be caused by another process in S.

Livelock: A set of processes S repeatedly enter the same sequence of states without producing useful work and will repeatedly continue to do so.

Note that these are closely related, but not quite exactly the same phenomena.

Liveness

Let’s start with deadlock:

For a deadlock to exist, four conditions are required:

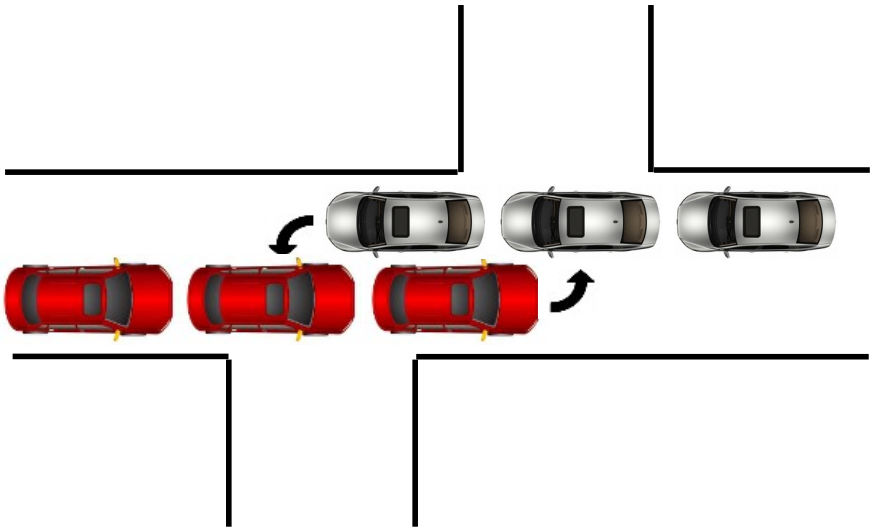
1. Processes claim exclusive control of some resource. (Lock())
2. Processes hold resources while waiting.
3. Resources cannot be taken from a process. (non-preemptive)
4. A circular chain of waiting exists.

Note that in this case, each process is holding onto something that another needs but that process is also waiting on another...

Liveness

Deadlock Examples:

Here's a real-world example: cars trying to turn.

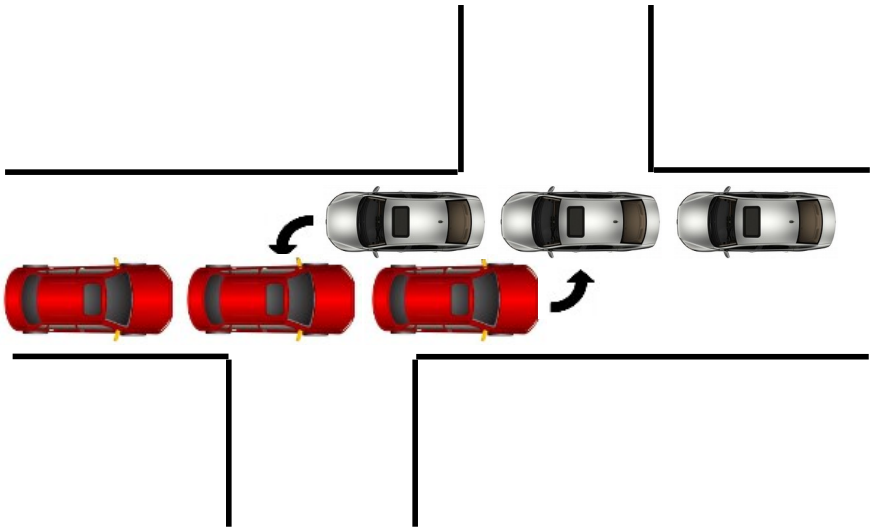


This has happened right down the street!

Liveness

Deadlock Examples:

Here's a real-world example: cars trying to turn.



This has happened right down the street!

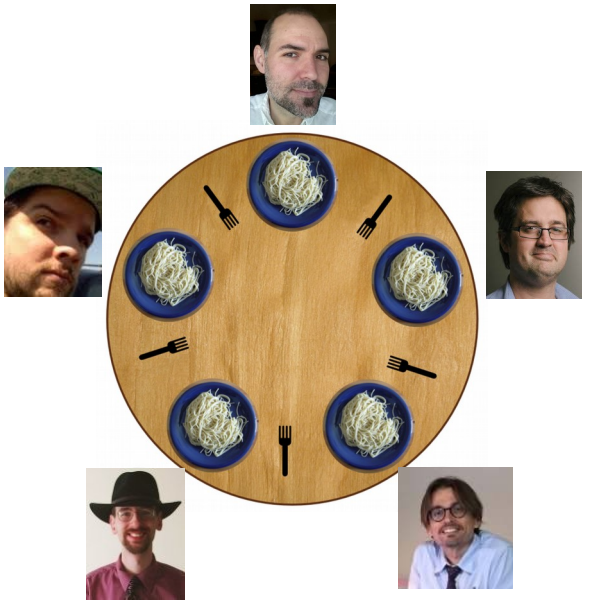
How does it meet the 4 conditions of deadlock?

1. Exclusive resource control?
2. Hold while waiting?
3. Resource can't be taken?
4. Circular wait?

Liveness

Deadlock Examples:

A fun example: The “Dining Philosophers”



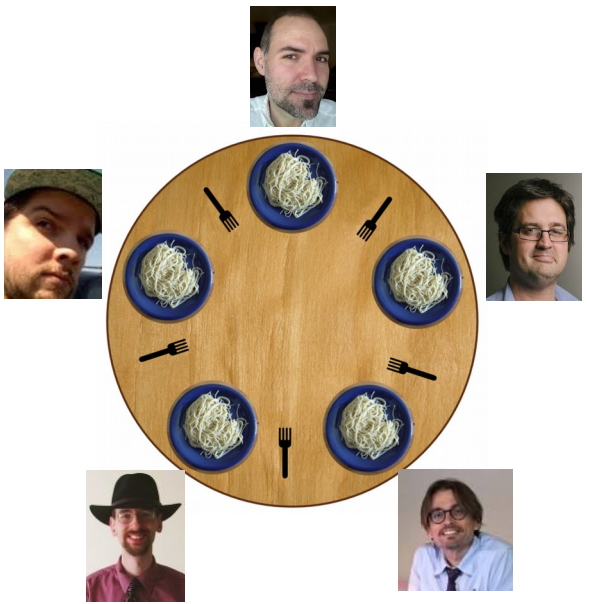
Each philosopher:

- Thinks for a bit.
- Picks up 1 fork.
- Picks up the other fork.
- Eats for a bit.
- Puts down both forks.
- Repeat until done.

This has not happened right down the street. :(

Liveness

What can go wrong here?



Remember, each philosopher does this:

- Thinks for a bit.
- Picks up 1 fork when available.
- Picks up the other fork when available.
- Eats for a bit.
- Puts down both forks.
- Repeat until done.

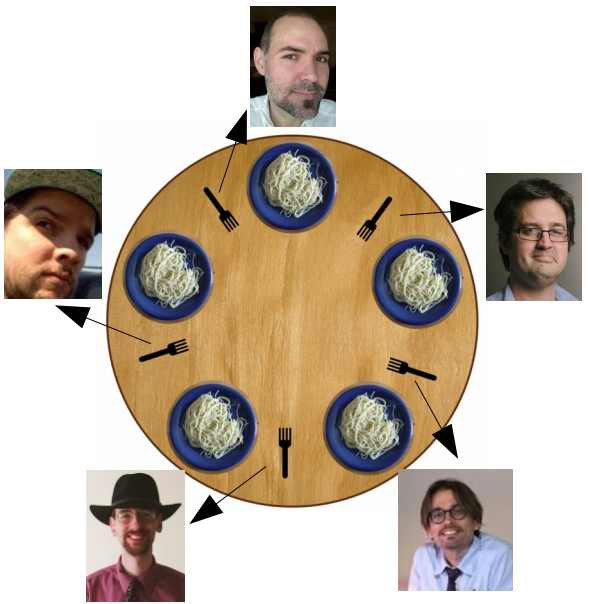
Looking at this, it seems like it makes sense:

When ready to eat, get two forks and eat, then put them down.

But this can deadlock! How?

Liveness

What can go wrong here?



Each could be ready to dine at the same time.

Each could pick up the fork to their right at the same time.

Each could then wait for the fork to the left to be available.

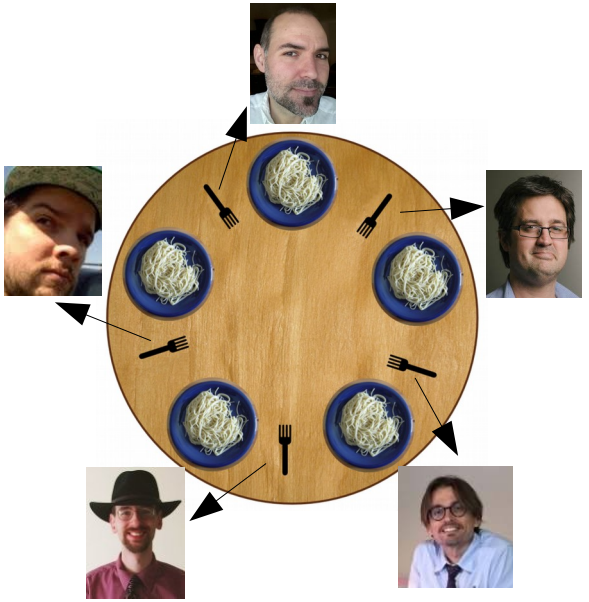
They will be waiting wait forever.

Nothing in the sequence allows them to do anything other than wait at this point.

How does this fit the 4 conditions?

Liveness

What can go wrong here?



How does it meet the 4 conditions of deadlock?

1. Exclusive resource control?
2. Hold while waiting?
3. Resource can't be taken?
4. Circular wait?

Liveness

Dealing with Deadlock:

There are three possible ways to deal with deadlock.

Deadlock Prevention – design the system in a way that eliminates one of the 4 conditions.

Deadlock Detection and Recovery – allow deadlocks to occur, detect when they do, break the deadlock and recover from it.

Deadlock Avoidance – allow all 4 conditions to occur, but allocate resources carefully so they all don't happen together.

Let's look at each of these...

Liveness

Deadlock Prevention:

Design the system in a way that eliminates one (or more) of the 4 conditions.

Some design options:

- Don’t allow exclusive use of resources
- All resources are requested and granted all at once.
- Release held resources while waiting on others.
- Create a linear hierarchical ordering of resource acquisition.

Which of the 4 conditions does each eliminate?

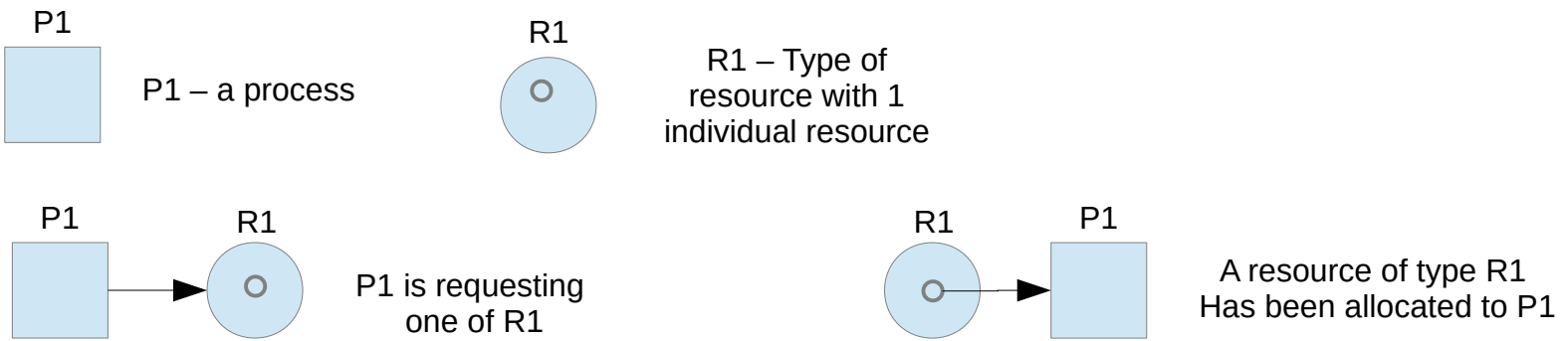
Note that some of these are impractical or can be inefficient.

Liveness

Deadlock Detection:

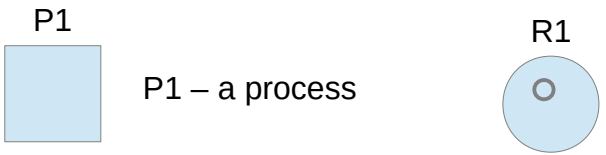
To detect deadlock, the general idea is to look for the circular wait.
We can detect this by looking for a cycle in the resource allocation graph.

Let’s look at how a resource allocation graph would work.



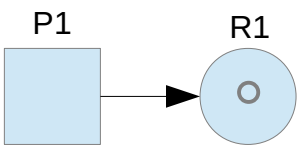
Liveness

Deadlock Detection:

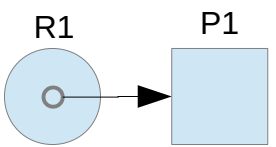


P1 – a process

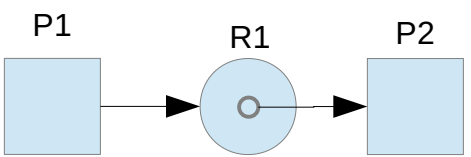
R1 – Type of resource with 1 individual resource



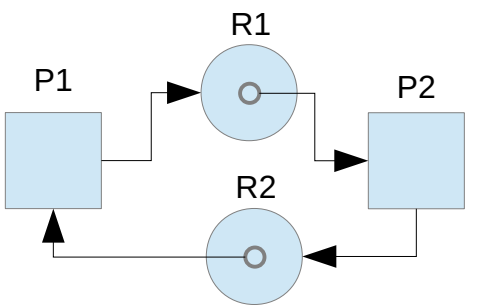
P1 is requesting one of R1



A resource of type R1 Has been allocated to P1



P1 wants R1
P2 holds R1



P1 wants R1
P2 holds R1
P2 wants R1
P1 holds R2

Liveness

Deadlock Detection:

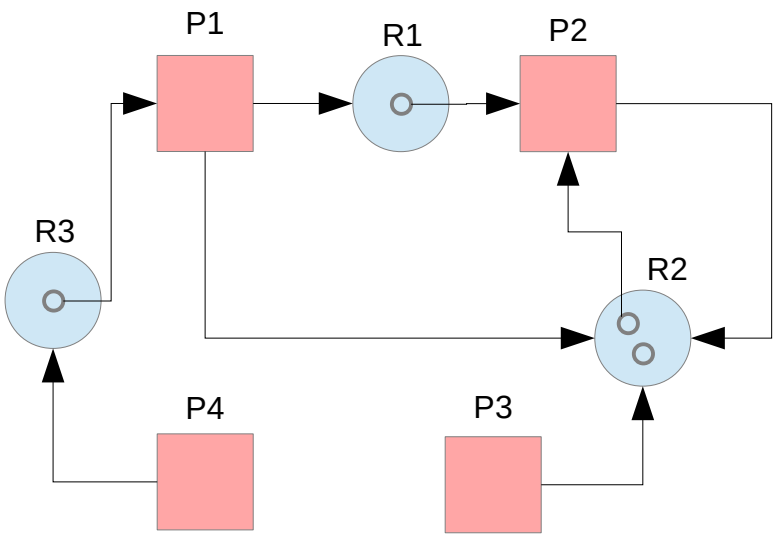
The idea is to perform a reduction of the resource allocation graph as follows:

- If all of the resource requests of a process *could* be granted then we say that the graph can be reduced by that process.
- Reducing by a process assumes that the process finishes and returns all held resources.
- If a graph can be completely reduced then there is no deadlock.
- If a graph cannot be completely reduced then there is a deadlock and the set of irreducible processes are those in the deadlock.

Note that we will have to complete this reduction every time resources requests change.

Liveness

Deadlock Detection Example 1:



Arrows out of a process are requests.

Arrows into a process are held resources.

Right now none of them have all needed resources.

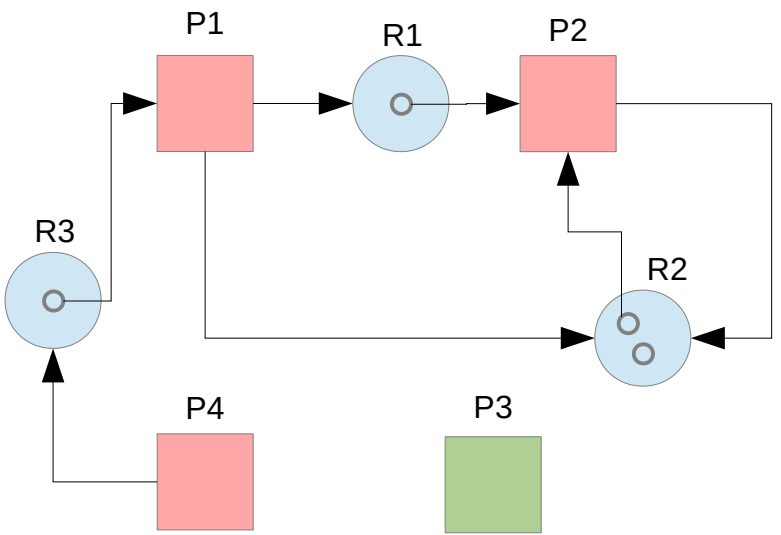
But, could any of them get all needed resources?

Yes, there's one of R2 that could be granted to either P1, P2, or P3!

Let's give it to P3 and reduce by P3.

Liveness

Deadlock Detection Example 1:

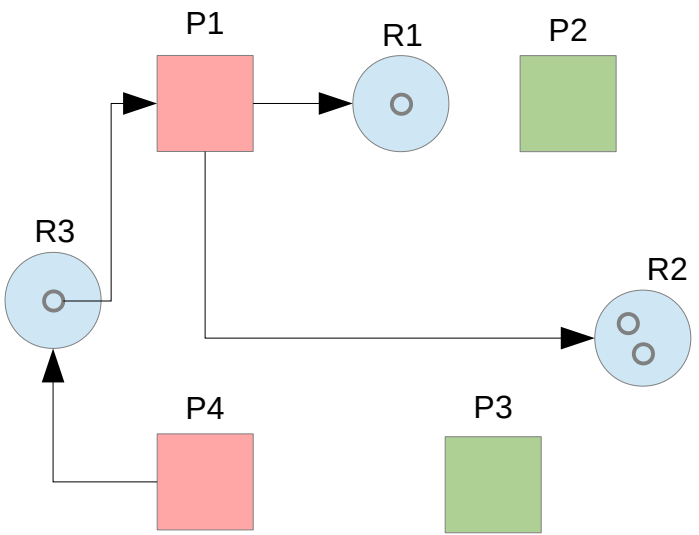


We reduced by P3.

Now let’s give it to P2 and reduce by P2.

Liveness

Deadlock Detection Example 1:



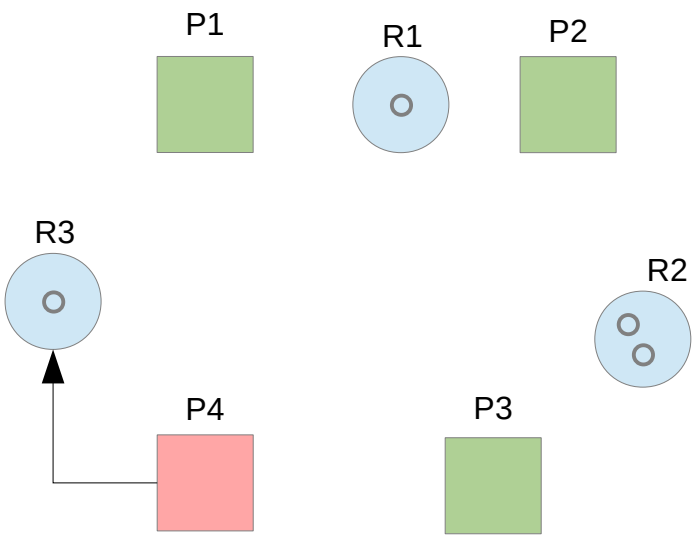
We reduced by P3.

We reduced by P2.

Now P1 can get R1 and R2 and complete.

Liveness

Deadlock Detection Example 1:



We reduced by P3.

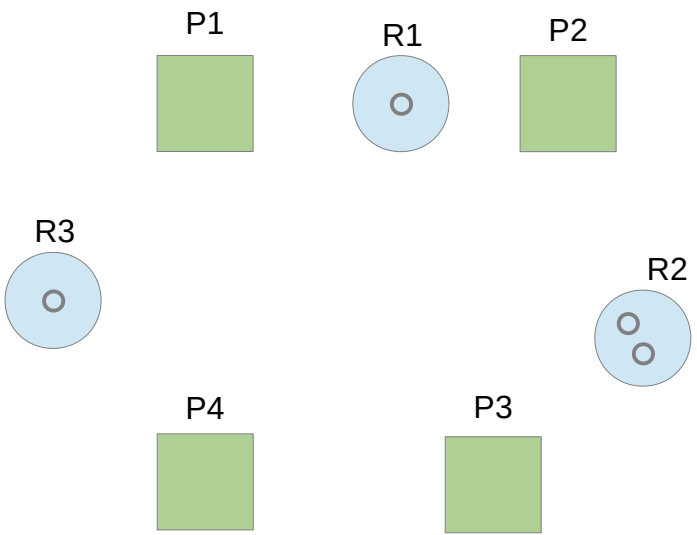
We reduced by P2.

We reduced by P1.

Now P4 can complete.

Liveness

Deadlock Detection Example 1:



We reduced by P3.

We reduced by P2.

We reduced by P1.

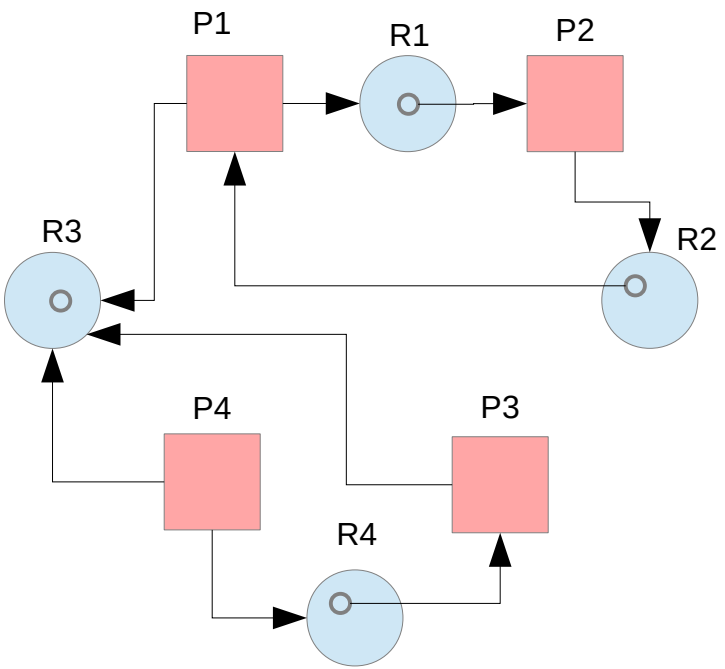
We reduced by P4.

All have been reduced so there is…

No deadlock!

Liveness

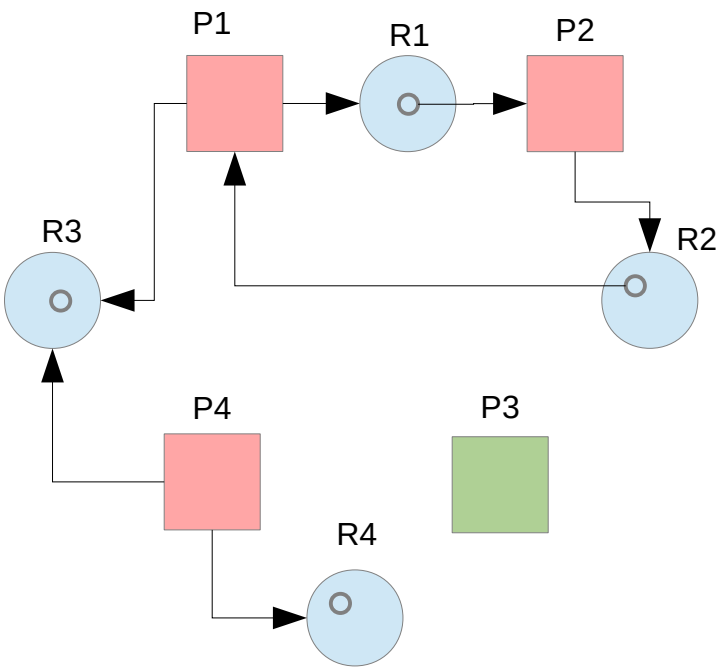
Deadlock Detection Example 2:



P3 could get R3 and complete.

Liveness

Deadlock Detection Example 2:

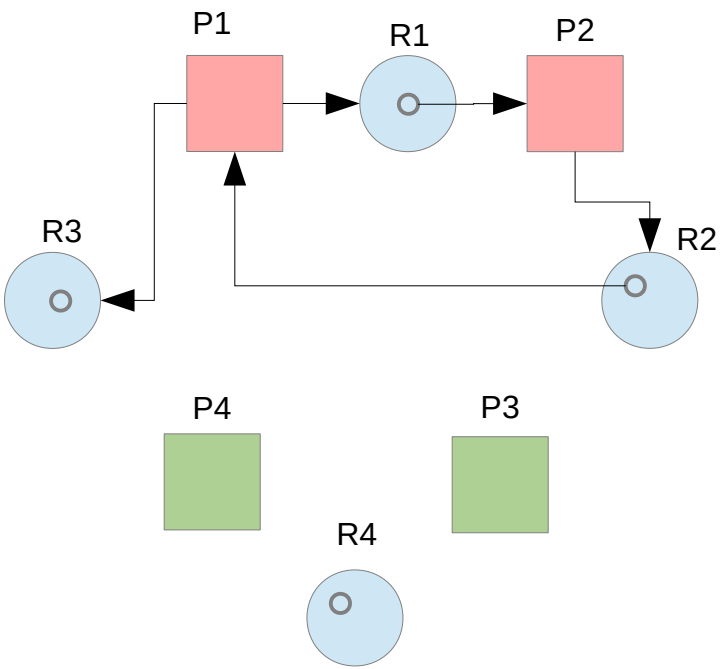


We reduced by P3.

P4 can now get R4 and R3 and complete.

Liveness

Deadlock Detection Example 2:



We reduced by P3.

We reduced by P4.

P1 can get R3 but not R1.
P2 has R1 but can't get R2.

We can't reduce any further.

Deadlock Detected!

P1 and P2 are involved in the deadlock.

Liveness

Deadlock Detection:

So now we have a way to detect deadlock. What do we do about it?

Deadlock Recovery Methods:

Killing processes: killing all the process involved in the deadlock. Killing process one by one. After killing each process check for deadlock again keep repeating the process until system recover from deadlock. Not ideal. Obviously.

Resource Preemption: Resources are preempted from the processes involved in the deadlock, preempted resources are allocated to other processes so that there is a possibility of recovering the system from deadlock. Not always possible.

Liveness

Deadlock Detection:

Note that this resource graph reduction has overhead involved. Especially since it must be done anytime there is a change in resource needs.

An alternative approach is to use timeout-based resource allocation.

If a resource wait or lock waits longer than expected we can stop and report deadlock. Or release all the locks of the process that timed-out and restart it.

Problem: we don't always know how long to wait before timeout is needed.
The system could just be slow and not actually deadlocked.
Some data could be half-updated when the timeout occurs. This could be bad.

Liveness

Deadlock Avoidance:

The idea here is to allow all 4 conditions to occur, but to be careful how allocations are granted.

In other words, never grant a resource request if it could lead to deadlock.

But how?

Suppose each process declares maximum resource requests at startup.

In other words, the processes state all the resources they need before they actually request them.

Liveness

Deadlock Avoidance:

We can define three kinds of system state:

Safe – Not deadlocked and not possibility of deadlock even if all processes had maximum resources allocated.

Unsafe – Not deadlocked but if everyone asked for maximum resources all at once, deadlock could occur.

Deadlocked – A deadlock has occurred.

To track this we will use some data structures...

Liveness

Deadlock Avoidance:

We will track 4 things:

EL: Existence List: How many of each type of resource exists.
ex: 5 file handles, 8 network sockets, 1 keyboard → EL=[5,8,1].

AL: Available List: How many of each type of resource is free.
ex: with 2 files, 4 sockets, and the keyboard allocated → AL=[3,4,0].

CA: Current Allocations: How much of each resource each process is using.

MA: Max Allocations: How much of each resource each process might need.

Liveness

Deadlock Avoidance:

So, Suppose that [Files,Sockets,Keyboard]:

EL=[5,8,1] (Existence List)
AL=[3,4,0] (Available List)

And we have 4 processes: A,B,C,D with CA and MA as

(Current Allocation)

CA	F	S	K
A	1	2	1
B	1	0	0
C	0	2	0
D	0	0	0

(Max Allocation)

MA	F	S	K
A	1	3	1
B	2	0	0
C	0	2	1
D	2	0	0

If all processes suddenly requested max could they all finish?
If Yes=Safe State. If No=Unsafe Sate.

Liveness

Deadlock Avoidance:

Is the previous state safe?
Well, how much could each request? Compute MA-CA.

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	0	1	0	
B	2	0	0	-	B	1	0	0	=	B	1	0	0	AL = [3,4,0]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

Since AL = [3,4,0] then ...

We can observe that A can get all resources (Process A needs 1 of S and 4 are available.)
We can now update AL by adding row A back to AL.
This goes as follows: AL[3,4,0] + CA.A[1,2,1] = AL[4,6,1]

Liveness

Deadlock Avoidance:

This would result in the following new request state:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	-	-	-	
B	2	0	0	-	B	1	0	0	=	B	1	0	0	AL = [4,6,1]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

Now **B** can finish and return it's allocated resources:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	-	-	-	
B	2	0	0	-	B	1	0	0	=	**B**	**-**	**-**	**-**	AL = [5,6,1]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

Liveness

Deadlock Avoidance:

This would result in the following new request state:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	-	-	-	
B	2	0	0	-	B	1	0	0	=	B	-	-	-	AL = [5,6,1]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

Now C can finish and return it's allocated resources:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	-	-	-	
B	2	0	0	-	B	1	0	0	=	B	-	-	-	AL = [5,8,1]
C	0	2	1		C	0	2	0		**C**	**-**	**-**	**-**	
D	2	0	0		D	0	0	0		D	2	0	0	

Liveness

Deadlock Avoidance:

This would result in the following new request state:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	-	-	-	
B	2	0	0	-	B	1	0	0	=	B	-	-	-	AL = [5,8,1]
C	0	2	1		C	0	2	0		**C**	**-**	**-**	**-**	
D	2	0	0		D	0	0	0		D	2	0	0	

Now **D** can finish and return it's allocated resources and all are done with AL = EL = [5,8,1]

So this state was **safe.**

Liveness

Deadlock Avoidance:

Let’s look at a different example…

EL=[3,4,1]
AL=[1,0,0]

(Current Allocation)

CA	F	S	K
A	1	2	1
B	1	0	0
C	0	2	0
D	0	0	0

(Max Allocation)

MA	F	S	K
A	1	3	1
B	2	0	0
C	0	2	1
D	2	0	0

Liveness

Deadlock Avoidance:

Let's look at a different example…

EL=[3,4,1]
AL=[1,0,0]

(Current Allocation)

CA	F	S	K
A	1	2	1
B	1	0	0
C	0	2	0
D	0	0	0

(Max Allocation)

MA	F	S	K
A	1	3	1
B	2	0	0
C	0	2	1
D	2	0	0

This is the same CA and MA as before, only EL and AL have changed.

Is this state safe?

Liveness

Deadlock Avoidance:

MA – CA = Req

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	0	1	0	
B	2	0	0	-	B	1	0	0	=	B	1	0	0	AL = [1,0,0]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

B can complete and return all resources.

This results in AL=[2,0,0]

Liveness

Deadlock Avoidance:

Now we have this:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	0	1	0	
B	2	0	0	-	B	1	0	0	=	B	-	-	-	AL = [2,0,0]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	2	0	0	

D can complete and return all resources.

This results in AL=[2,0,0] (AL didn't change because CA.D was [0,0,0])

Now we have this...

Liveness

Deadlock Avoidance:

Now we have this:

(Max Allocation)					(Current Allocation)					(Request Remain)				
MA	F	S	K		CA	F	S	K		Req	F	S	K	
A	1	3	1		A	1	2	1		A	0	1	0	
B	2	0	0	-	B	1	0	0	=	B	-	-	-	AL = [2,0,0]
C	0	2	1		C	0	2	0		C	0	0	1	
D	2	0	0		D	0	0	0		D	-	-	-	

Neither A nor C can go.

So the system could deadlock and is therefore **unsafe.**

Liveness

Deadlock Avoidance:

This leads us to Dijkstra's Banker's Algorithm.

When a process Q makes a request:

- Let AL' = AL (available list)
- Let CA' = CA (current allocation)
- Pretend that the request was granted: update AL' and CU'
- Compute Req'=MA-CA' and act as if all processes are requesting maximum resources.
- Using Req', AL', and CA': see if there is a way to finish. (the resulting state is safe).
- If the state is safe: grant the resource to S and update AL=AL' and CA=CA'
- If the state is unsafe: reject the request and leave AL and CA unchanged.

This is a brilliant solution: Simple and easy to code. Leaves all 4 conditions but is smart about allocation.

Problems: processes might not always know all resources at startup. May deny some requests to preserve safety. Denies requests if there is ANY possibility of deadlock even if not likely.

Liveness

Livelock:

Deadlock’s ugly cousin.

Dining philosophers can livelock too.

Livelock can be considered worse in some cases.

Like zombies using resources while doing nothing.

Event-driven Programming

That’s all for today!

Stay Safe!