deadlock

IN PROCESS (need sof)
- mutex
- hold & wait
- no preemption
- circular wait

Solutions
- Lock ordering
  - grab locks in predefined order
  - breaks circular wait
  - difficult to manage on global scale
  - client could grab lock earlier than it would like
- Conservative 2-phase lock
  - breaks hold & wait
  - if locks can't be acquired release all locks
  - provide serializability
- Monitors
  - encapsulate data
  - 1 lock ONLY
  - breaks hold & wait

IN OS (necessary)
- bounded resources *

- hold & wait
- no preemption
- circular wait

Solutions

- Prevention = break 1 condition
  - Resource ordering (see lock ordering)
- Avoidance = check resource request
  - 3 states
    - safe ✓
    - unsafe !
    - deadlock X
  - banker's algorithm
    - has N units but loans out more
    - OK as long as N+1 is not needed
    - denies request @ unsafe st

```
if(maxReq[i] <= availible){
        availible += allocated;
        p[i] += availible
    }
else
    exit(UNSAFE);
```

Availible | Max Poss Req

R_1	R_2	R_3	R_4
1	5	2	0

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	2	4

P_1: 1 5 3 2
P_2: ~~X~~
P_3: ~~X~~
P_4: 1 11 6 4
P_5 ~~X~~

P_2: 2 11 6 4
P_3: 3 14 11 7
P_5: 3 14 15 11

SAFE

Allocated

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	1	4	4

Available

R_1	R_2	R_3	R_4
0	0	0	0

Max Poss Req

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3

P_4	0	0	2	0
P_5	0	6	2	4

Allocated

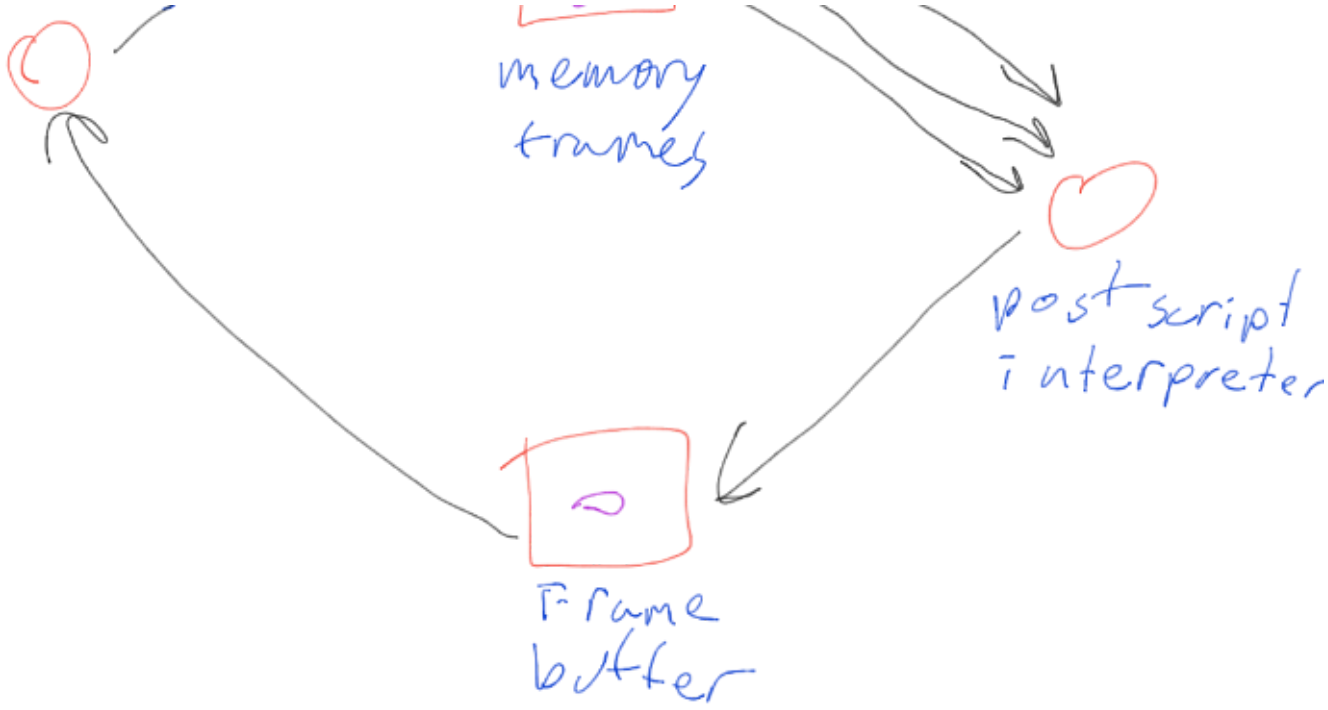
	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	1	4	4

DEADLOCK

- processes may enter system
- must know resource request upfront
- resources are assumed to be working

- detection = let system go into deadlock and recover

RAG (resource allocation graph)



- might have deadlock
  - preempt resources 1/time
  - kill threads in cycle
  - kill threads 1/time

- We don't know when to execute algorithm

- OSTRICH ALGORITHM
  - Ignore the problem & pretend it doesn't exist