



CS 3530 Operating Systems

Semaphores & Deadlock

CS 3530 Operating Systems. Dr. Ken Hoganson, Copyright © 2008

- A construct that utilizes atomic operations, to provide a general solution to the critical section problem, usable for many purposes
- Consists of a semaphore flag or lock and
- A list of processes waiting to enter a CS
- Uses two atomic operations that operate on the flag and list
 - **Wait:** adds the process to the waiting list, adjust the flag to indicate another process is waiting
 - **Signal:** adjusts flag to indicate process has completed, and “wakes-up” the next process in the list.

- Note: This list of waiting processes avoids the use of spin loops.
- A waiting process can go into a wait state
- Spin loops utilize processor cycles to execute instructions
- And use bus bandwidth to fetch instructions and data (could be from a cache)
- Spin loops are inefficient, and should be avoided if possible.

Wait(flag) {

Decrement flag

If flag < 0 {

 Add self to list

 Go into wait state }

}

(Else proceed on flag == 0)

Signal(flag) {

Increment flag

If flag <= 0 {

 Remove next process from list,
 and wake it up}

}

! definition of wait in pseudocode is

```
wait(S) {  
    while (S ≤ 0)  
        ; // no-op  
    S--;  
}
```

of signal in pseudocode is

```
signal(S) {  
    S++;  
}
```

	l	1	2	3	4	5	6	7	8	9	10	11	12	13
Flag S1	1													
List S1														
Using S1														
Flag S2	1													
List S2														
Using S2														

1. $P1 \leftarrow R1$
2. $P2 \leftarrow R2$
3. $P3 \leftarrow R1$
4. $P1 \rightarrow R1$
5. $P4 \leftarrow R1$
6. $P2 \rightarrow R1$
7. $P1 \leftarrow R2$

8. $P3 \leftarrow R2$
 9. $P1 \leftarrow R1$
 10. $P3 \rightarrow R1$
 11. $P3 \rightarrow R2$
 12. $P1 \rightarrow R2$
 13. $P1 \rightarrow R1$
- \leftarrow request \rightarrow release

```

Wait(flag) {
    Decrement flag
    If flag < 0 {
        Add self to list
        Go into wait state }
    }

```

```

Signal (flag) {
    Increment flag
    If flag <= 0 {
        Remove next process from
        list, and wake it up}
    }

```

- Necessary conditions for deadlock
 1. Mutual Exclusion (non-shared)
 2. Hold and Wait – process holds a resource while waiting for another to become available
 3. No preemption – cannot force a process to release a resource being held
 4. Circular wait – processes are holding the resources needed by other processes, which prevent processes from proceeding and releasing held resources.

- Preventing Deadlock?
 1. Mutual Exclusion (non-shared): allow sharing when appropriate – clearly not always possible
 2. Hold and Wait – force stalled process to release and re acquire, rather than hold and wait, can be inefficient and produce thrashing
 3. No preemption – allow preemption – what happens to a pre-empted process – start over?
 4. Circular wait – Prevent circular waits from occurring, using an algorithm/strategy –

- If system allows deadlock (all 4 conditions are present), then;
- Need to be able to detect deadlock
- Need to be able to break deadlock

Symbols:

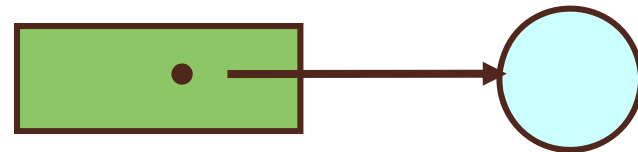
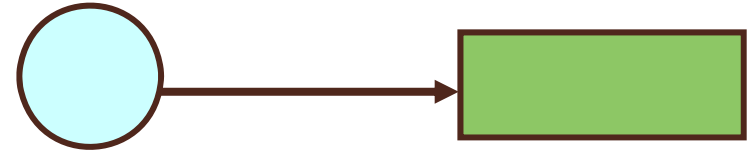
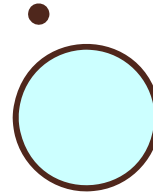
• Resource

• Tokens

• Process

• Request

• Grant



Simple Cycle

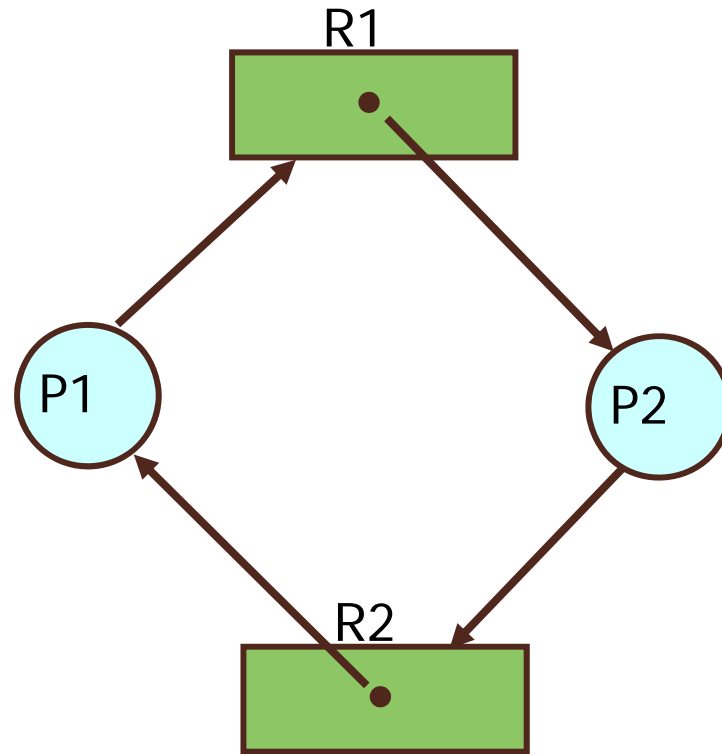
P1 request R2, granted

P2 request R1, granted

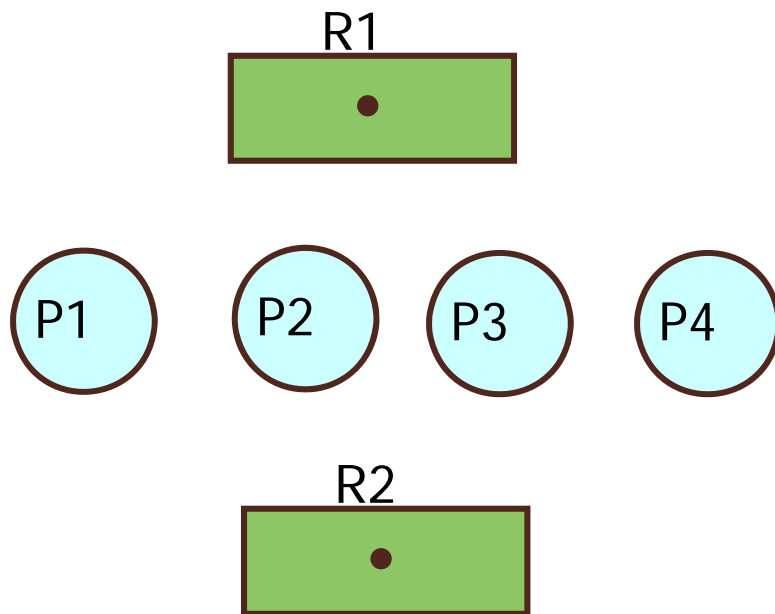
P1 request P1, wait

P2 request R2, wait

Cycle – deadlock when all other conditions are present



	I	1	2	3	4	5	6	7	8	9	10	11	12	13
Flag S1	1													
List S1														
Using S1														
Flag S2	1													
List S2														
Using S2														



1. P1 ← R1
2. P2 ← R2
3. P3 ← R1
4. P1 → R1
5. P4 ← R1
6. P2 → R1
7. P1 ← R2
8. P3 ← R2
9. P1 ← R1
10. ~~P3 → R1~~
11. P3 → R2
12. P1 → R2
13. P1 → R1

Deadlock?

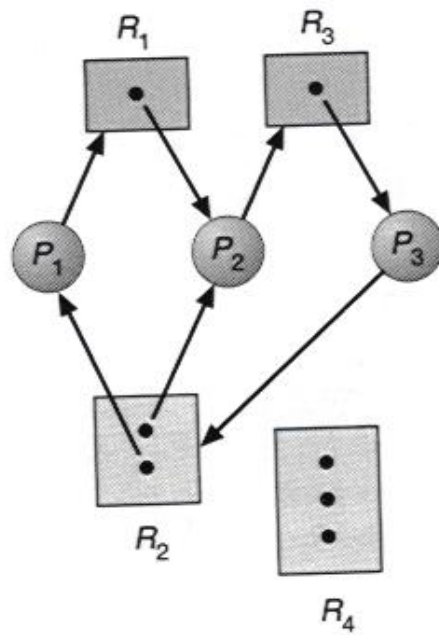


Figure 8.2 Resource-allocation graph with a cycle and a deadlock.

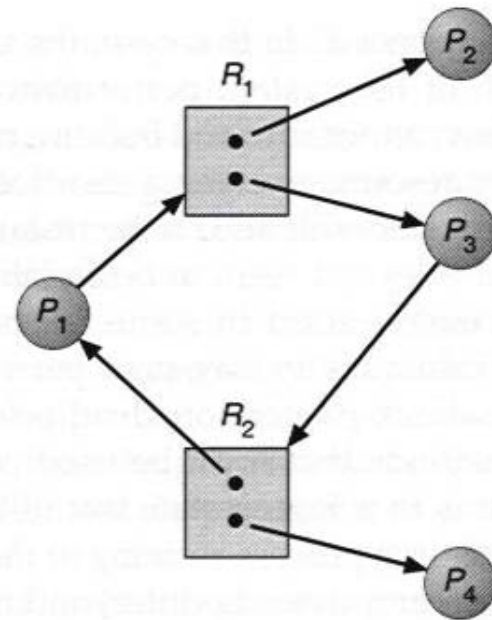
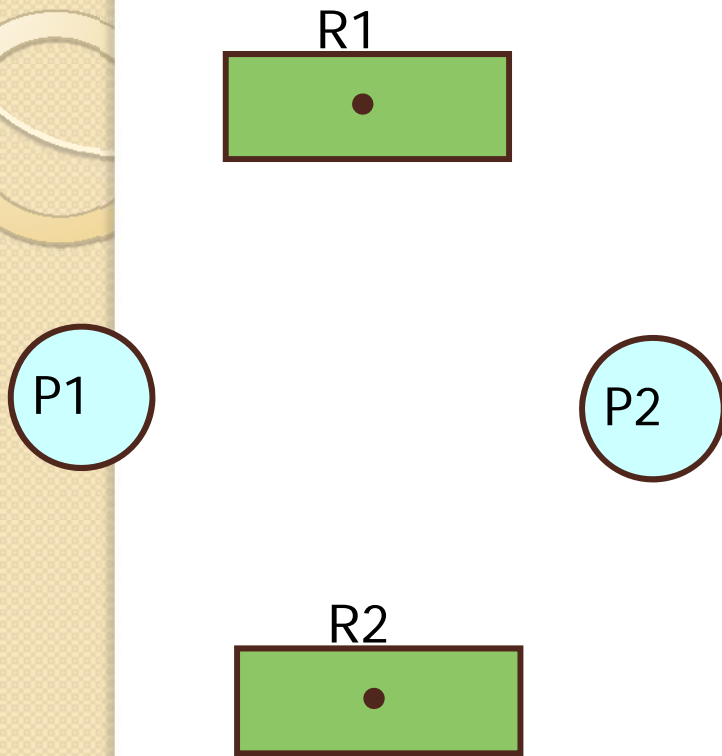


Figure 8.3 Resource-allocation graph with a cycle but no deadlock.

- Resource Ranking
- Concept: Rank each resource in numerical order, no duplicates. Processes must request resources in order.
- Prevents deadlock by preventing cycles – some process will have a higher numbered resource, and wants a lower numbered resource – must release high number prior to requesting lower number



Resource rank order is R1 then R2.

- P1 request R2, granted
- P2 request R1, granted
- P1 request P1, cannot do, must release R2, then request and wait on R1
- P2 request R2, granted, progress, release

Progress – both processes complete

- Is there a “best” resource ranking scheme, for a given system, with a specific set of resources?
- Number of each resource is known
- Frequency of access is known

- For each resource request (prior to granting it), determine if the system will be in a “safe” state if request is granted.
- What is a “safe state”? (not deadlock) There is some sequence of processing and resource release and request, that will allow all processes to complete. Guarantee Progress
- Overhead work – running the algorithm may require significant processing – imagine 1000s of processes on a machine with dozens or 100s of resources.

Safe State

- Single resource example
- Safe State? Yes – P1 can get enough resources to complete, then P0, then P2.
- What if P0 now requests 2 resources?
- What if P2 now requests 1 resource?

Total = 12	Max need	Currently allocated	Need
P0	10	5	5
P1	4	2	2
P2	9	2	7
Available		3	

- Multi-resource solution to deadlock prevention
- Overhead processing
- Name comes from Banks allocating cash to ensure it can meet the demand from all customers.
- Extends Safe State Concept for multiple resources
- Formalizes the evaluation of whether a granting a request leaves system in a safe state:
Safety Algorithm

	Allocated			Max Need			Need Alloc-Max =	10 5 7 Available	
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3			
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

- **Safe State?** Sequence of process completions that let all complete? (more than one?)

	Allocated			Max Need			Need Alloc-Max =	10 5 7 Available	
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3			
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

- P1 requests (1,0,2). OK to grant?

**End
Of
Today's
Lecture.**



This slide intentionally left blank