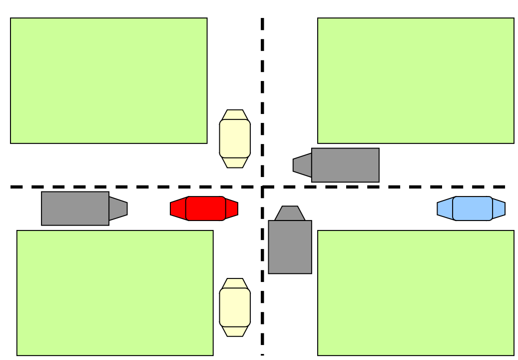
Deadlock

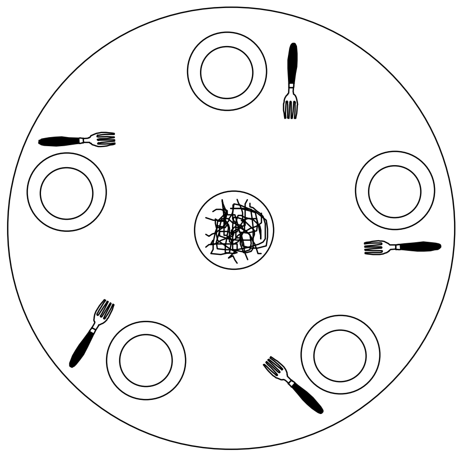
Deadlock

- If a group of processes are blocked on their resource requests that can never be satisfied, then it is a deadlock



Dining Philosophers Problem

- 5 philosophers, each needs to eat with 2 forks



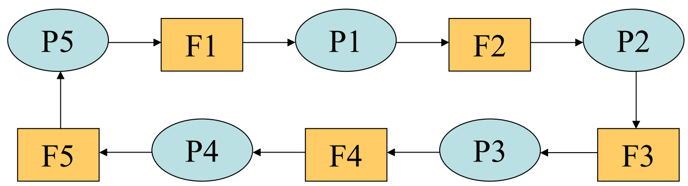
Philosopher *i*:
repeat
  think;
  $f1 = i$;
  $f2 = (i + 1) \bmod 5$;
  wait (*fork*[$f1$]);
  wait (*fork*[$f2$]);
  eat;
  signal (*fork*[$f1$]);
  signal (*fork*[$f2$]);
until *false*;

Necessary Conditions for Deadlock

- Mutual Exclusion
  - Shared resources are acquired and used mutually exclusively
- Hold and Wait
  - Each process continues to hold resources already allocated to it while waiting to acquire new resources
- No Preemption
  - Resources granted to a process can only be released back to the system after the process finished using it
- Circular Waiting
  - Processes forms a chain in the wait for graph

Wait-For Graph

- Specify the relationship among processes and resources
  - A directed edge from a process to a resource represents that the process is requesting for the resource
  - A directed edge from a resource to a process represents that the resource is held by the process



Deadlock Solutions

- Deadlock Prevention
  - Disallow at least one of the four necessary conditions
- Deadlock Avoidance
  - Grant only those requests for available resources which cannot possibly result in deadlock
- Deadlock Detection
  - Grant available resources to requesting processes freely and periodically check for deadlocks
  - After deadlock is detected, proceed to **deadlock recovery**

Deadlock Prevention

- Disallow at least one of the four necessary conditions
- Avoid Mutual Exclusion
  - Nothing can be done if the resource cannot be shared
- Allow Preemption
  - Not all resources can be preempted (e.g., printer)
- Avoid Circular Wait
  - **Linear ordering** on resources
  - E.g., resources A, B, C, order(A) = 1, order(B) = 2, order(C) = 3
  - request (C and A) $\Rightarrow$ request (A); request (C);
  - Problems:
    - Require resource requirement information in advance
    - Low resource utilization
    - Degradation of concurrency (resources are allocated before it is needed)

Deadlock Prevention

- Dinning philosopher problem

Philosopher i:
repeat
  think;
  $f1 = i$; $f2 = (i+1) \bmod 5$;
  if $(f1 > f2)$ $switch(f1, f2)$;
  wait (*fork*[$f1$]);
  wait (*fork*[$f2$]);
  eat;
  signal (*fork*[$f1$]);
  signal (*fork*[$f2$]);
until *false*;

Deadlock Prevention

- Disallow at least one of the four necessary conditions
- Avoid Hold-and-Wait
  - Solution 1: A process has to get all of its resources before it begins execution, but the resources may be released at any time, any order
  - Problems for solution 1
    - Require resource requirement information in advance
    - Low resource utilization
    - Degradation of concurrency (worse than linear ordering)
  - Solution 2: A process that has to wait for some resources with: order of one of the held resources > order of the waiting resource
    - Will release all the resources allocated to it
    - (similar to linear ordering without resource information in advance)
  - Problems for solution 2
    - Not all resources can be preempted

Deadlock Avoidance

- System grants allocations of resources only if it is guaranteed to be deadlock free
  - For a new request requesting for resources
    - System "pretends" to grant the resources
    - Check whether the "new state" is safe
    - Grant the requested resources only if it is safe
  - Safe
    - In the future, the system can allocate resources in at least one order that guarantees free of deadlock
  - Unsafe
    - System may (still may not) end up in deadlock situation
  - Problem
    - Need to know max resource needed for all processes in advance

Banker's Algorithm

- System Model
  - N: Number of processes
  - M: Number of different types of resources
  - Input: *Request*[i,j]
    - $1 \le i \le N$, $1 \le j \le M$
    - Number of type j resources being requested by process i
  - Output: grant or deny *Request*[i,j]

Banker's Algorithm

- Information to be maintained
  - *Total*[M]
    - *Total*[j]: Total number of available resources of type j
  - *MaxReq*[N, M]
    - *MaxReq*[i,j]: Maximum number of type j resources that may be requested by process i
  - *Allocated*[N, M]
    - *Allocated*[i,j]: Number of type j resources that are currently allocated to process i

Banker's Algorithm

- Information to be derived
  - *Need*[*N*, *M*]
    - *Need*[*i*,*j*]: Number of type *j* resources that may still be needed by process *i*
  - *Available*[*M*]
    - *Available*[*j*]: Number of resources of type *j* that is still available

Banker's Algorithm Example

- Current system state

	Total	Max			Alloc			Need			Avail
		P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3	
R_1	7	2	7	3	0	2	1				
R_2	6	3	3	4	1	1	0				
R_3	6	5	1	1	0	1	0				

- Request

	P_1	P_2	P_3
R_1	0	1	0
R_2	1	0	0
R_3	0	0	0

Banker's Algorithm Example

- Current system state

	Total	Max			Alloc			Need			Avail
		P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3	
R_1	7	2	7	3	0	2	1	2	5	2	4
R_2	6	3	3	4	1	1	0	2	2	4	4
R_3	6	5	1	1	0	1	0	5	0	1	5

- Request

	P_1	P_2	P_3
R_1	0	1	0
R_2	1	0	0
R_3	0	0	0

Banker's Algorithm

- Preliminary checks
  - Check to see if the request is invalid

    if *Request*[*i*,*j*] > *Need*[*i*,*j*], for any *i*, *j*, **then**

    error;
  - Check to see whether the system has enough resources for the request

    if Σ_i *Request*[*i*,*j*] > *Available*[*j*], for any *j*, **then**

    deny the request;

Banker's Algorithm

- Pretend to allocate

  for all j: $Available[j] := Available[j] - \Sigma_i\ Request[i,j]$;

  for all i, j: $Allocated[i,j] := Allocated[i,j] + Request[i,j]$;

  for all i, j: $Need[i,j] := Need[i,j] - Request[i,j]$;

- Check safety

  if current system is safe **then**

  grant the allocation to the request;

  else deny the request

- Restore original state if request is denied

  for all j: $Available[j] := Available[j] + \Sigma_i\ Request[i,j]$;

  for all i, j: $Allocated[i,j] := Allocated[i,j] - Request[i,j]$;

  for all i, j: $Need[i,j] := Need[i,j] + Request[i,j]$;

Banker's Algorithm

- Check safety

```
for all j: Temp[j] := Available[j];  -- not to change Available
Pset := the set of all processes;
repeat
   if there exists an i such that Need[i, j] < = temp[j] then
   { -- process i can complete its execution
     remove process i from Pset;
     for all j: Temp[j] := Temp[j] + Allocated[i,j];
     -- pretend process i is done, return all resources
   }
   else return (unsafe);
until Pset = Empty;
return (safe);
```

Banker's Algorithm Example 1

❖ Current system state

	Total	Max			Alloc			Need			Avl
		P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3	
R_1	7	2	7	3	0	2	1	2	5	2	4
R_2	6	3	3	4	1	1	0	2	2	4	4
R_3	6	5	1	1	0	1	0	5	0	1	5

Request

	P_1	P_2	P_3
R_1	0	1	0
R_2	1	0	0
R_3	0	0	0

$\Rightarrow$ **no problem**

❖ Preliminary checks

- **if** *Request*[*i*,*j*] > *Need*[*i*,*j*], for any *i*, *j* **then error**;

Request
0 1 0
1 0 0 $\Rightarrow$ **error**
0 0 2

- **if** Σ_i *Request*[*i*,*j*] > *Available*[*j*], for any *j* **then deny**;

Request
0 5 0
1 0 0 $\Rightarrow$ **deny**
0 0 0

Banker's Algorithm Example 1

❖ Current system state

Total	Max	Alloc	Need	Avl
7	2 7 3	0 2 1	2 5 2	4
6	3 3 4	1 1 0	2 2 4	4
6	5 1 1	0 1 0	5 0 1	5

Request
0 1 0
1 0 0
0 0 0

❖ After "pretend to allocate"

Total	Max	Alloc	Need	Avl
7	2 7 3	0 3 1	2 4 2	3
6	3 3 4	2 1 0	1 2 4	3
6	5 1 1	0 1 0	5 0 1	5

No change

Banker's Algo Example 1 (Check Safety)

- Check safety

Total	Max	Alloc	Need	Avl
7	2 7 3	0 3 1	2 4 2	3
6	3 3 4	2 1 0	1 2 4	3
6	5 1 1	0 1 0	5 0 1	5
	7 3	3 1	4 2	3
	3 4	1 0	2 4	5
	1 1	1 0	0 1	5
	7	3	4	4
	3	1	2	5
	1	1	0	5

Which process can continue?
Need[*i*, *j*] <= *Avail*[*j*]
P_1 can!

Pretend:
- Allocate resources to P_1
- P_1 finishes
- P_1 returns all its resources

P_3 can continue!
P_2 can continue!

All finishes ⇒ Safe!
⇒ Grant request!

Banker's Algorithm Example 2

- Current system state

Total	Max	Alloc	Need	Avl
7	2 7 3	0 2 1	2 5 2	4
6	3 3 4	1 1 0	2 2 4	4
6	5 1 1	0 1 0	5 0 1	5

Request
0 1 0
1 0 0
0 0 1

- After "pretend to allocate"

Total	Max	Alloc	Need	Avl
7	2 7 3	0 3 1	2 4 2	3
6	3 3 4	2 1 0	1 2 4	3
6	5 1 1	0 1 1	5 0 0	4

Which process can continue?
Need[*i*, *j*] < = *Avail*[*j*]

None of them can!!!!
⇒ Deny request

Banker's Algorithm Example 3

- Current system state

Total	Max	Alloc	Need	Avl
5	1 4 3	0 1 0	1 3 3	4
4	4 3 1	0 2 0	4 1 1	2
3	2 1 1	2 0 0	0 1 1	1

Request
0 1 0
0 1 1
0 0 0

- After "pretend to allocate"

Total	Max	Alloc	Need	Avl
5	1 4 3	0 2 0	1 2 3	3
4	4 3 1	0 3 1	4 0 0	0
3	2 1 1	2 0 0	0 1 1	1

Banker's Algorithm Example 3

- Check safety

Total	Max	Alloc	Need	Avl
5	1 4 3	0 2 0	1 2 3	3
4	4 3 1	0 3 1	4 0 0	0
3	2 1 1	2 0 0	0 1 1	1

Which process can continue?
$Need[i, j] <= Avail[j]$
P_2 and P_3 both can!
Choose P_2

-- Pretend:
- Allocate resources to P_2
- P_2 finishes
- P_2 returns all its resources

Total	Alloc		Need		Avl
5	0	0	1	3	5
4	0	1	4	0	3
3	2	0	0	1	1

-- P_3 can continue!

P_3 returns its resources!

Total	Alloc	Need	Avl
5	0	1	5
4	0	4	4
3	2	0	1

All finishes ⇒ Safe!
⇒ Grant request!

Banker's Algorithm Example 4

- Current system state

Total	Max	Alloc	Need	Avl
5	1 4 3	0 1 0	1 3 3	4
4	4 2 4	0 1 1	4 1 3	2
3	2 1 2	1 0 0	1 1 2	2

Request
1 0 0
0 0 0
1 0 0

- After "pretend to allocate"

Total	Max	Alloc	Need	Avl
5	1 4 3	1 1 0	0 3 3	3
4	4 2 4	0 1 1	4 1 3	2
3	2 1 2	2 0 0	0 1 2	1

Which process can continue?
$Need[i, j] <= Avail[j]$

P_2 can ⇒ return all

None of the rest can
⇒ Deny request

Deadlock Detection

- Just let deadlock happen
  - No prevention, no avoidance
- System periodically checks for deadlock
  - When a deadlock is detected, a recovery process is required to break the cycle
  - Use the safety check algorithm for deadlock detection
    - No need to know maximum resource requirements "Max"
    - No need to know still needed resources "Needed"
- deadlock recovery
  - Kill one process at a time till deadlock cycle is broken

Safety Algorithm for Detection

❖ Check safety

	Total	Alloc P_1	Alloc P_2	Alloc P_3	Avl	Req P_1	Req P_2	Req P_3
R_1	3	0	2	0	1	2	0	1
R_2	4	0	1	1	2	0	2	1
R_3	2	1	0	0	1	0	1	2

P_2 can get the resources
$\rightarrow$ P_2 return resources

	Total	Alloc P_1	Alloc P_2	Alloc P_3	Avl	Req P_1	Req P_2	Req P_3
R_1	3	0		0	3	2		1
R_2	4	0		1	3	0		1
R_3	2	1		0	1	0		2

P_1 can get the resources
$\rightarrow$ P_1 return resources

	Total	Alloc P_1	Alloc P_2	Alloc P_3	Avl	Req P_1	Req P_2	Req P_3
R_1	3			0	3			1
R_2	4			1	3			1
R_3	2			0	2			2

P_3 can get the resources

All finishes $\Rightarrow$ Safe!
$\Rightarrow$ **No deadlock!**

Safety Algorithm for Detection

❖ Check safety

	Total	Alloc P_1	Alloc P_2	Alloc P_3	Avl	Req P_1	Req P_2	Req P_3
R_1	2	0	0	1	1	2	0	1
R_2	4	0	1	1	2	0	2	1
R_3	2	1	0	0	1	0	1	2

P_2 can get the resources
$\rightarrow$ P_2 return resources

	Total	Alloc P_1	Alloc P_2	Alloc P_3	Avl	Req P_1	Req P_2	Req P_3
R_1	2	0		1	1	2		1
R_2	4	0		1	3	0		1
R_3	2	1		0	1	0		2

No process can get the resources
$\Rightarrow$ **Deadlock**
P_1 and P_3 involved

Comparisons

- Overhead in deadlock prevention (linear ordering)
  - Take resources in advance ⇒ Lower resources utilization; Reduce degree of concurrency
- Overhead in deadlock avoidance
  - Need to update resource information for every allocation/deallocation
  - Need to run safety algorithm for each request (very expensive)
- Overhead in deadlock detection
  - Detection algorithm can be expensive
    - but only when there is a potential deadlock
  - Need to wait for resources till a deadlock is detected
- No perfect solution -- Consider combinations

Readings

- Section 6.2-6.4, 6.6
- Good but not covered: 6.8