

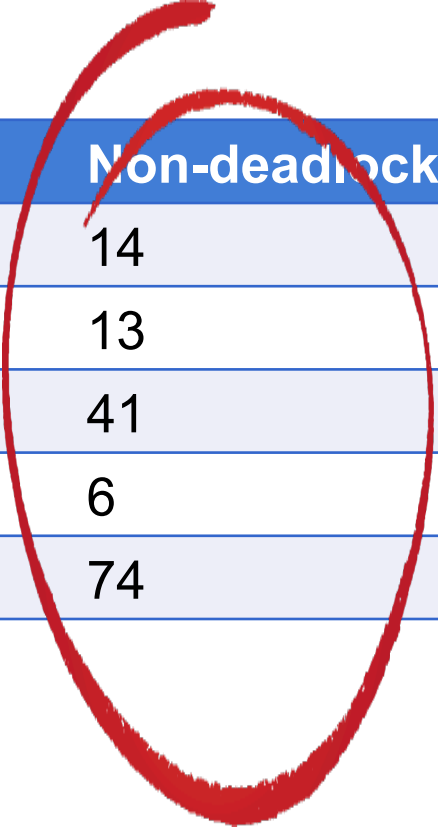
# Concurrency problems

To do ...

- ☐ Non-deadlock bugs
- ☐ Resources & deadlocks
- ☐ Dealing with deadlocks
- ☐ Other issues
- ☐ Next Time: I/O and file systems

# Bugs in some modern applications

- From some key apps, a summary of bugs\*



Application	Description	Non-deadlock	Deadlock
MySQL	Database server	14	9
Apache	Web server	13	4
Mozilla	Web browser	41	16
OpenOffice	Office suite	6	2
Total		74	31

- 97% where atomicity or order violations!

\*S. Lu et al., Learning from Mistakes, ...

<http://pages.cs.wisc.edu/~shanlu/paper/asplos122-lu.pdf>

(examples from Arapci-Duseau's book)

# Non-deadlock bugs – Atomicity-violation

```
// From MySQL
```

```
Thread 1::
```

```
If (thd->proc_info) {
```

```
    ...
```

```
    fputs(thd->proc_info, ...);
```

```
    ...
```

```
}
```

*Where's the bug?*

```
Thread 2::
```

```
thd->proc_info = NULL;
```

“The desired serializability among multiple memory accesses is violated (i.e., a code region is intended to be atomic, but the atomicity is not enforced during execution.”

# Non-deadlock bugs – Atomicity ~~violation~~

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 1::

```
pthread_mutex_lock(&lock);
```

```
If (thd->proc_info) {
```

```
    ...
```

```
    fputs(thd->proc_info, ...);
```

```
    ...
```

```
}
```

```
pthread_mutex_unlock(&lock);
```

Thread 2::

```
pthread_mutex_lock(&lock);
```

```
thd->proc_info = NULL;
```

```
pthread_mutex_unlock(&lock);
```

# Non-deadlock bugs – Order-violation

Thread 1::

```
void init() {
```

```
    ...
```

```
    mThread = PR_CreateThread(mMain, ...);
```

```
    ...
```

```
}
```

*Where's the bug?*

Thread 2::

```
void mMain(...) {
```

```
    ...
```

```
    mState = mThread->State;
```

```
    ...
```

```
}
```

“The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution).”

# Non-deadlock bugs – Order ~~violation~~

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;  
int mtInit = 0;
```

Thread 1::

```
void init() {  
    ...  
    mThread = PR_CreateThread(mMain, ...);  
  
    // Signal that the thread has been created ...  
    pthread_mutex_lock(&mtLock);  
    mtInit = 1;  
    pthread_cond_signal(&mtCond);  
    pthread_mutex_lock(&mtLock);  
    ...  
}
```

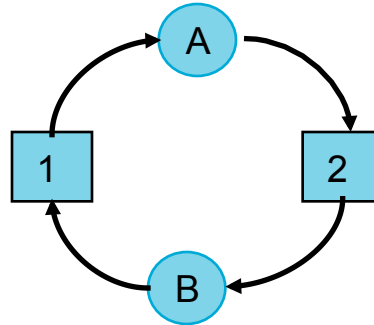
# Non-deadlock bugs – Order ~~violation~~

```
Thread 2::  
void mMain(...) {  
    ...  
    // Wait for the thread to be initialized ...  
    pthread_mutex_lock(&mtLock);  
    while (mtInit == 0)  
        pthread_cond_wait(&mtCond, &mtLock);  
    pthread_mutex_unlock(&mtLock);  
  
    mState = mThread->State;  
    ...  
}
```

# Deadlock bugs

Thread A:

```
lock(L1);  
lock(L2);  
...
```



Thread B:

```
lock(L2);  
lock(L1);  
...
```

*“That’s some catch,  
that Catch-22”*

- Set of threads is deadlocked if each thread is waiting for an event that only another thread in the set can cause
- None of the threads can ...
  - Run, release resources or be awakened



# Introduction to deadlocks

- Assumptions

- Threads or single-threaded processes
- There are no interrupts possible to wake up a blocked thread



- Another “cute” example

- “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up until the other has gone.” An actual law passed by the Kansas legislature ...

# Conditions for deadlock

1. Mutual exclusion - Each resource assigned to 1 thread or available
2. Hold and wait - A thread holding resources can request others resources
3. No preemption - Previously granted resources cannot forcibly be taken away
4. Circular wait – A circular chain of 2+ threads, each waiting for resource held by the next thread

All conditions must hold for a deadlock to occur.

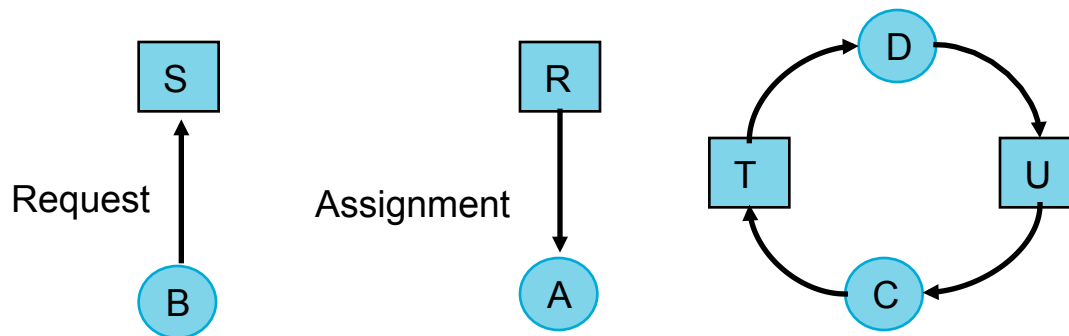
Each of the 1-3 conditions is associated with a policy the system can or not have; break one condition → no deadlock

# System model

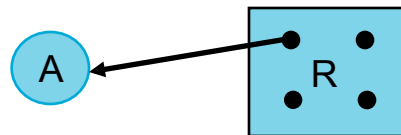
- System – a collection of resources to be shared
- Resources, in types with multiple instances each (printers, files, memory,...)
- Resources can be
  - Preemptable - can be taken away w/o ill effects (e.g. memory)
  - Nonpreemptable - process will fail if resource were taken away (e.g. CD recorder)
- A thread must request a resource before using it & release it once done (`open/close`, `malloc/free`, ...)
  - Sequence of events to use a resource: request/use/release

# Deadlock modeling

- Modeled with directed graphs
  - Process B is requesting/waiting for resource S
  - Resource R assigned to process A
  - Process C & D in deadlock over resources T & U

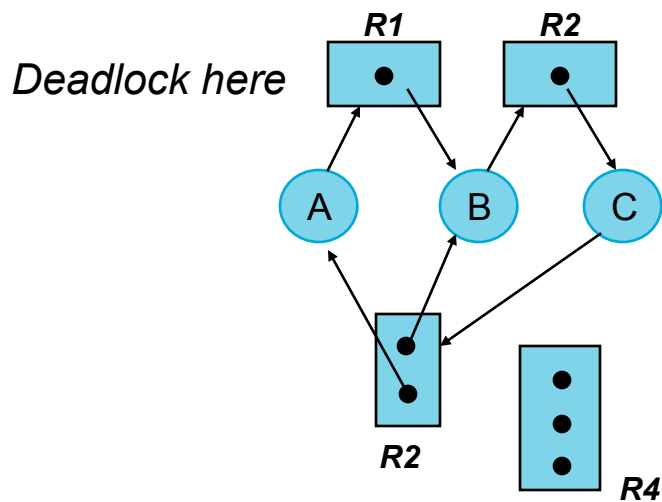


- Multiple resource instances per class

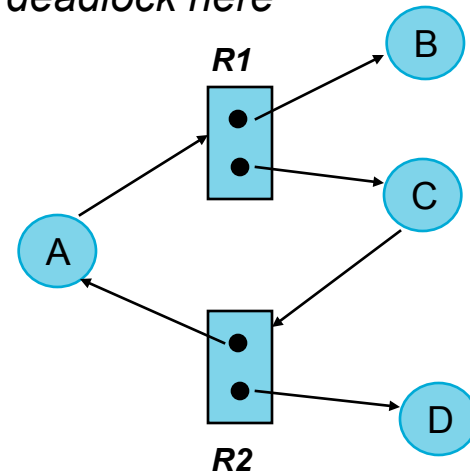


# Basic facts

- If graph contains no cycles  $\Rightarrow$  no deadlock.
- If graph contains a cycle  $\Rightarrow$ 
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, maybe a deadlock.



*No deadlock here*



# Dealing with deadlocks

## Possible strategies

- Ignore the problem altogether – ostrich “algorithm”
- Detection and recovery – do not stop it; let it happen, detect it and recover from it
- Dynamic avoidance – careful resource allocation
- Prevention – negating one of the four necessary conditions

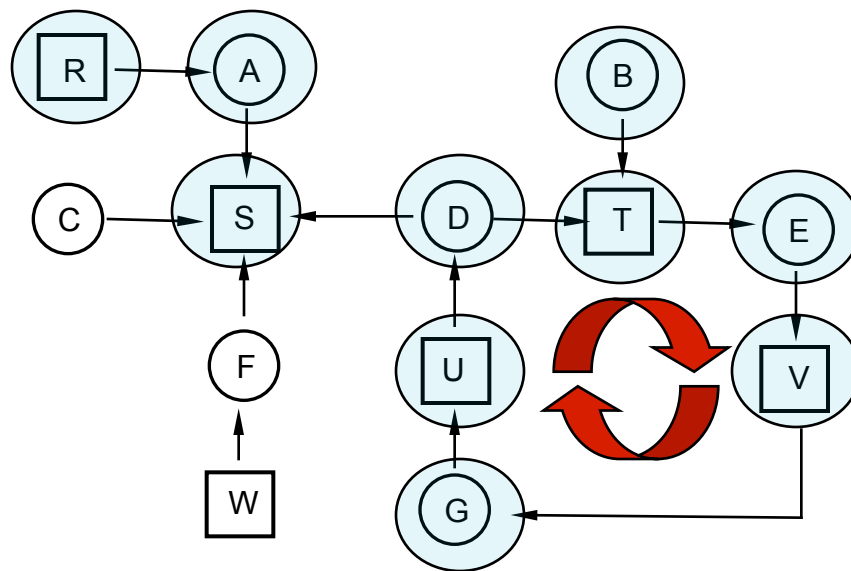
# The ostrich algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX's & Windows' approach
- A clear trade off between
  - Convenience
  - Correctness
- Not quite an option for your code



# Deadlock detection – single instance

- Detect a cycle in the directed graph
  - Simplest case
- *How, when & what*



1.  $L \leftarrow \text{empty}$   
 all arcs set as unmarked

2. For each node  $N$   
 /\* depth-first search \*/  
 2.1. Add  $N$  to  $L$  & check  
 if  $N$  in  $L$  twice there's a deadlock; exit  
 2.2. Pick one arc at random, mark it & follow it to next current node

3. At end, if no arc no deadlock

A linked list of nodes

Arcs:

$A \rightarrow S, A \leftarrow R, B \rightarrow T, C \rightarrow S$   
 $D \rightarrow S, D \leftarrow T, E \rightarrow V, E \leftarrow T$   
 $F \rightarrow S, F \leftarrow W, G \rightarrow V, G \leftarrow V$

$L: [R], L: [R, A], L: [R, A, S]$

$L: [B], L: [B, T], L: [B, T, E], \dots$



# Detection - multiple instances

$n$  processes,  $m$  classes of resources

$E$  – vector of existing resources

$A$  – vector of available resources

$C$  – matrix of currently allocated resources

$R$  – request matrix

$C_{ij}$  –  $P_i$  holds  $C_{ij}$  instances of resource class  $j$

$R_{ij}$  –  $P_i$  wants  $R_{ij}$  instances of resource class  $j$

Invariant –  $\sum_i C_{ij} + A_j = E_j$   
(Currently allocated + available = existing)  
i.e. all resources are either allocated or available

Algorithm:

Idea: See if there's any process that can be run to completion with available resources, mark it and free its resources ...

**All processes unmarked**

**1. Look for unmarked process  $P_i$  for which  $R_i \leq A$**

**2. If found, add  $C_i$  to  $A$ , mark the process and go to 1**

**3. If not, exit**

**All unmarked processes, if any, are deadlock**

# Detection

(existing)

$E = (4\ 2\ 3\ 1)$

(available)

$A = (2\ 1\ 0\ 0)$

What process 1 needs

$C =$	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	1	0	$R =$	<table border="1"><tr><td>2</td><td>0</td><td>0</td><td>1</td></tr></table>	2	0	0	1
0	0	1	0								
2	0	0	1								
	2	0	0	1		1	0	1	0		
	0	1	2	0		2	1	0	0		

What process 1 has

Three processes and 4 resource types

After running process 3

$A = (2\ 2\ 2\ 0)$

Now you can run process 2

$A = (4\ 2\ 2\ 1)$

...

Algorithm:

All processes unmarked

1. Look for unmarked process  $P_i$  for which  $R_i \leq A$
  2. If found, add  $C_i$  to  $A$ , mark the process and go to 1
  3. If not, exit
- All unmarked processes, if any, are deadlock

# *When to check & what to do*

- *When to try*
  - Every time a resource is requested
  - Every fixed period of times or when CPU utilization drops
- *What to do then - recovery*
  - Through preemption
    - depends on nature of the resource
  - Through rollback
    - Need to checkpoint processes periodically
  - By killing a process
    - Crudest but simplest way to break a deadlock
    - Kill one in or not in the deadlock cycle

# Deadlock avoidance

Clearly, the ordering of operations plays a role

Requests and releases  
of each process

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

... and one particular  
ordering

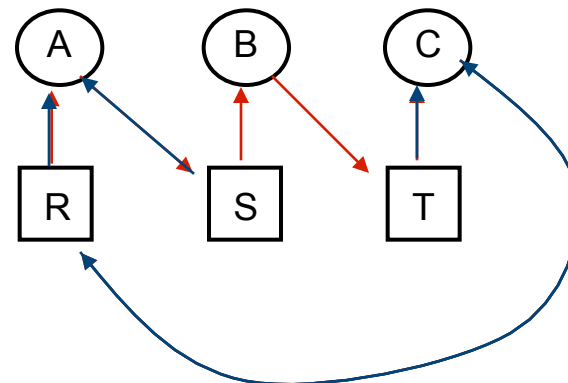
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

....  
**deadlock**

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

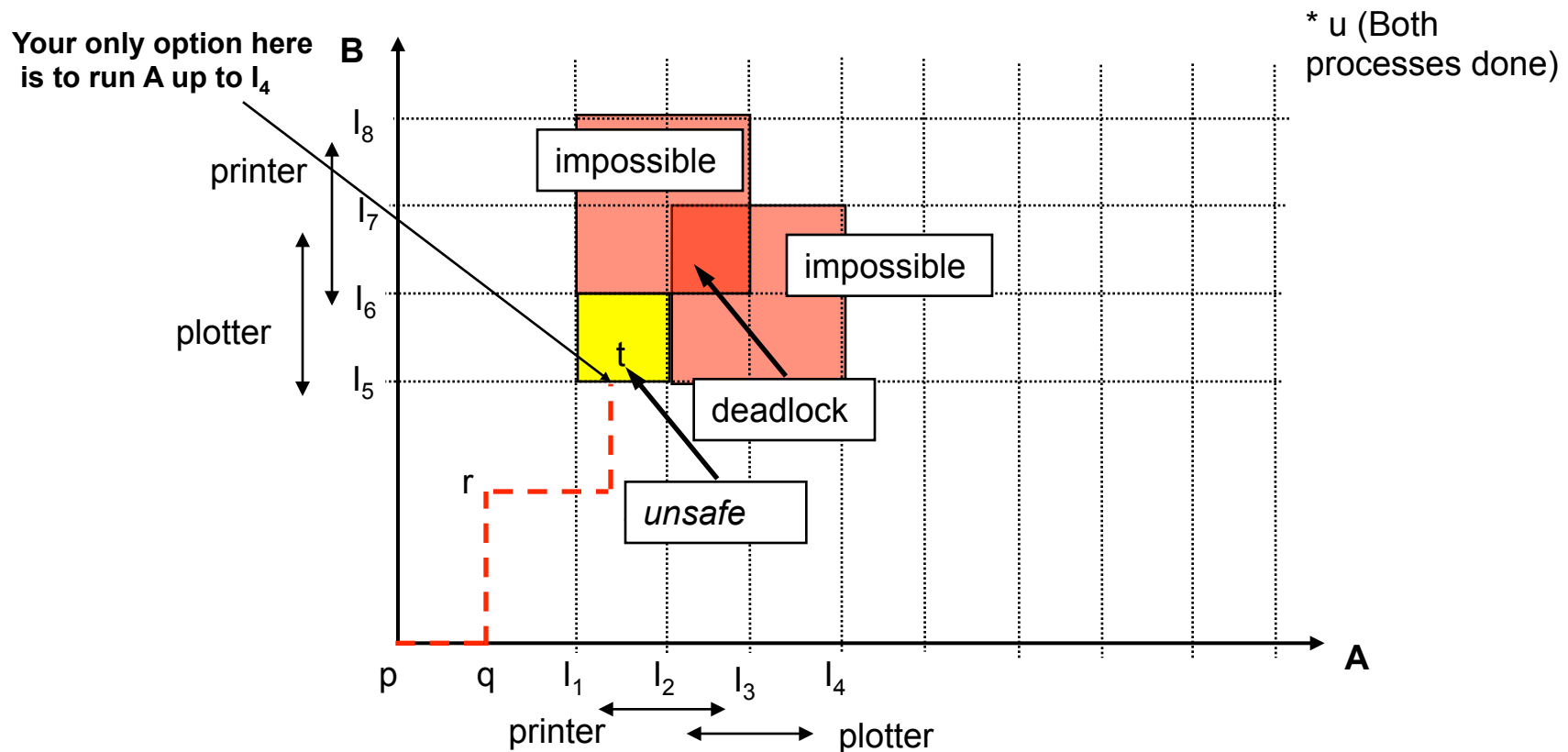
....  
**no deadlock**

But with an  
alternative



# Deadlock avoidance

- Dynamically make sure not to get into a deadlock
- Two process resource trajectories
- Every point in the graph, a joint state of the processes



# Safe and unsafe states

- Safe if
  - There is no deadlock
  - There is a scheduling order by which all processes can finish
- Un-safe is not deadlock – just no guarantee

Example with one resource (10 instances of it)

Free: 3

	Has	Needs
A	3	9
B	2	4
C	2	7

**Safe**

	Has	Needs
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Needs
A	3	9
B	0	-
C	2	7

Free: 5

	Has	Needs
A	3	9
B	0	-
C	7	7

Free: 0

	Has	Needs
A	3	9
B	0	-
C	0	-

Free: 7

Free: 3

	Has	Needs
A	3	9
B	2	4
C	2	7

**Unsafe**

	Has	Needs
A	4	9
B	2	4
C	2	7

Free: 2

	Has	Needs
A	4	9
B	4	4
C	2	7

Free: 0

	Has	Needs
A	4	9
B	0	-
C	2	7

Free: 4

A requests and is granted another instance

In retrospect, A's request should not have been granted

# Banker's algorithm (*Dijkstra, again*)

- Considers
  - Each request as it occurs
  - Sees if granting it leads to a safe state i.e. there are enough resources to satisfy one customer
- With multiple resources
  1. Look for a row  $R_i \leq A$ , if none the system will eventually deadlock
  2. If found, mark  $P_i$  and add  $C_i$  to  $A$
  3. Repeat until processes are terminated or a deadlock occurs
- Very cute, but mostly useless
  - Most processes don't know in advance what they need
  - The lists of processes and resources are not static
  - Processes may depend on each other

# Deadlock prevention

- Avoidance is pretty hard or impossible
- Can we break one of the condition?
  - Mutual exclusion
  - Hold & wait
  - No preemption
    - Not a viable option
    - How can you preempt a printer?
  - Circular wait



# Attacking mutual exclusion

- Some devices can be spooled (printer)
  - Only the printer daemon uses printer resource
  - Thus deadlock for printer eliminated
- But not all devices can be spooled – process table?
- Principle:
  - Assigning resource only when absolutely necessary
  - Reduce number of processes that may claim the resource

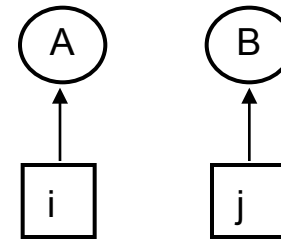
# Attacking hold & wait

- Processes request all resources at start (hold & ~~wait~~)
  - Process never has to wait for what it needs
- But
  - May not know required resources at start
  - It ties up resources others could be using
- Variation (~~hold~~ & wait)
  - Process must release all resources to request a new one

# Attacking circular wait

- Impose total order on resources
- Processes request resources in order
- If all processes follow order, no circular wait occurs

Deadlock if  $i \rightarrow A \rightarrow j$  &  $j \rightarrow B \rightarrow i$   
If  $i < j$  then  $A \rightarrow j \dots$



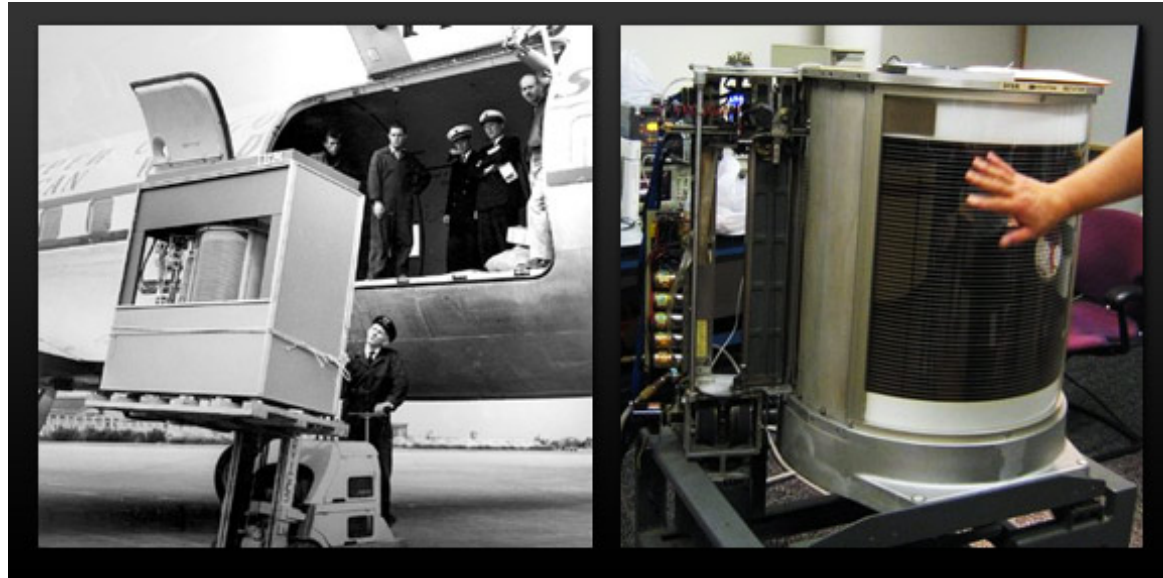
- Process cannot request resource lower than what it's holding
- Advantage - Simple
- Disadvantage - Arbitrary ordering

# Related issues

- Non-resource deadlocks
  - Each is waiting for the other to do some task
  - E.g. communication deadlocks:
    - A sends a request and blocks until B replies, message gets lost!
    - Timeout!
- Livelocks – try, sleep and try again
  - There's some action, just not progress
- Starvation
  - SJF to allocate resources – consider allocation of a printer
    - May cause long job to be postponed indefinitely
      - even though not blocked
  - FIFO?

# Next time

- I/O devices
- File abstraction and
- File systems ...



First Hard Disk

IBM Model 350 Disk File, part of IBM 305 RAMAC computer in 1956 ~5MB capacity