Thread Mgmt: Processes → Threads → ?

- # of resources in a system: finite
- Traditional Unix (life was easy!)
  - single thread of control
  - multiprogramming: 1 process actually executing
  - non-preemptive processes

- Distributed systems, multi-threading etc.
- How do we handle issues of:
  - resource constraints
  - ordering of processes/threads
  - precedence relations
  - access control for global parameters
  - shared memory
  - IPC
  - Scheduling etc.

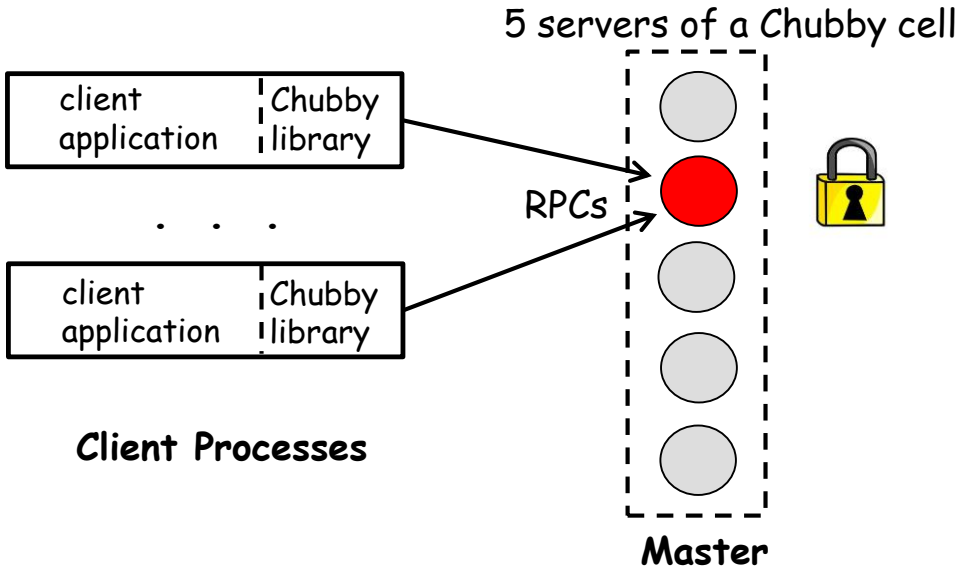
such that the solutions are: fair, efficient, deadlock and race-free?

Resource Sharing & Inter-Process Interactions

- **Basics**: Critical Sections, Mutual Exclusion, Deadlocks…
- **Task Orderings**: Scheduling issues and solutions
- **Algorithmic** Solutions: Races, Ordering, Alternations…
- **Program Level** Solutions: Semaphores, Monitors

Google' Chubby Lock Service: ~100K Servers

Chubby, is a distributed lock service for coarse-grained locking. Key parts of Google's infrastructure (Google FS, BigTable, MapReduce, …) use Chubby to synchronize accesses to shared resources.

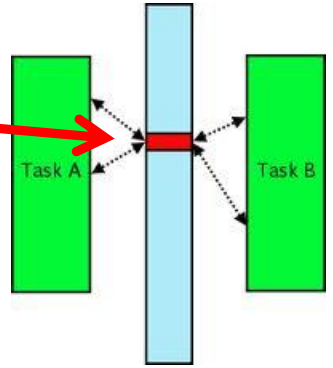


- *1 node is holding a consistency lock that others need to access to proceed. Node holding lock hangs ➔ All servers hang!*
- *Multiple locks get set; old lock not released ➔ inconsistent masters*
- *Server partitions happen ➔ inconsistent resource access; R/W consistency lost*

Critical Section (CS) & Mutual Exclusion (ME)

(or how to do something useful in an OS)

- Lots and lots of processes and threads in an OS – how do we orchestrate their running without contentions?
- How to give each process exclusive access (**ME**) to shared execution areas/resources (**CS**'s)



- If a process starts writing in the same space where another process is reading from? Cloud Storage Consistency: Amazon EC2, Cassandra, Dynamo, GFS, Voldemort, …

- If a new process gets allocated the same resource (registers, variables, memory etc) already allocated to an existing process (before that process gets to finish)?

- If a slow/stalled process blocks other processes from executing?

Four essential conditions need to hold to provide **M**utual **E**xclusion

Unity: No two processes are simultaneously in the CS

Fairness: No assumptions can be made about speeds or numbers of CPUs except that each process executes at non-zero speed

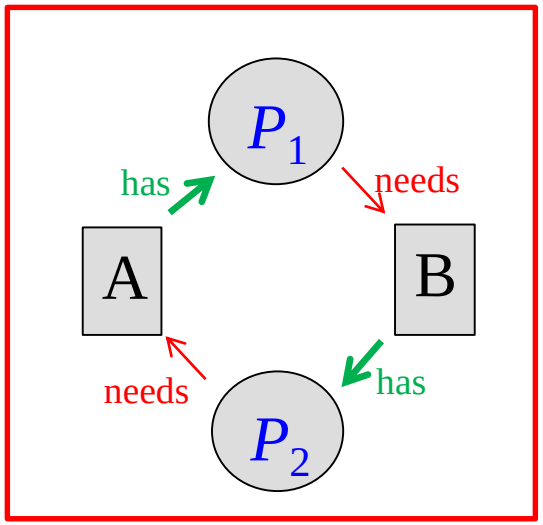
Progress: No process running outside its CS may block another process (from accessing the CS) *...guess where DoS attacks come from?*

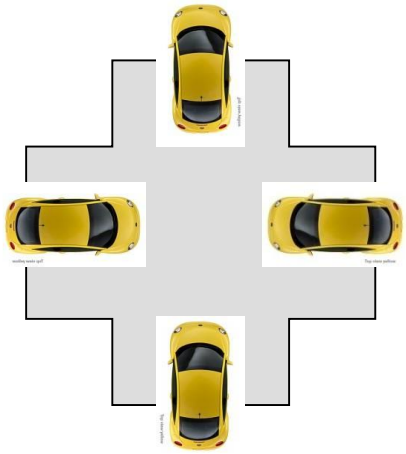
Bounded Waiting: No process must wait forever to enter its CS

- Deadlocks, Livelocks, Races, …
- Interrupt Disabling, Lock variables, Strict alternation (busy waiting), Sleep & Wakeup

Deadlock (& Livelock) …Starvation

* A set of processes is deadlocked if each process waits for a resource being held by another





Deadlock ➔ None of the processes can …run, release resources or be awakened

Livelock ➔ Activity but no progress (e.g pick, see deadlock, put-down, pick…)

Deadlock Characterization

Deadlock can arise if all four conditions hold simultaneously

- **Mutual exclusion:** Each resource is either currently assigned to exactly one process or is available
- **Hold and wait:** A process holding at least one resource has requested and is waiting to acquire additional resources currently held by other processes
- **No preemption:** A resource cannot be acquired forcibly from a process holding it; can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** A circular chain of >1 processes exits! There exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

Deadlock Issues

General strategies for dealing with deadlocks

- Just ignore the problem altogether (…and it will go away)
- Detection (and recovery): Let it happen, detect and recover
- Avoidance and prevention: Avoid, by design…
  - Do resource allocation to avoid situations
  - Negate one of the four necessary conditions for deadlock

1) **Mutual exclusion:** Each resource is either currently assigned to exactly one process or is available
2) **Hold and wait:** A process holding at least one resource has requested and is waiting to acquire additional resources currently held by other processes
3) **No preemption:** A resource cannot be acquired forcibly from a process holding it; can be released only voluntarily by the process holding it, after that process has completed its task
4) **Circular wait:** A circular chain of >1 processes exits! There exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

The Easy (Windows, Unix…) Way!

Pretend there is no problem (and it will go away …)! ☺

- Actually….it works, IF:
  - …the cost of avoidance/prevention is high
  - …deadlocks occur very rarely or may possibly go away with random back off (timeouts!) on service responses
    - **Ethernet:** Exponential back off basis …

- Tradeoff between
  - convenience
  - correctness (repeatable, predictable/deterministic)

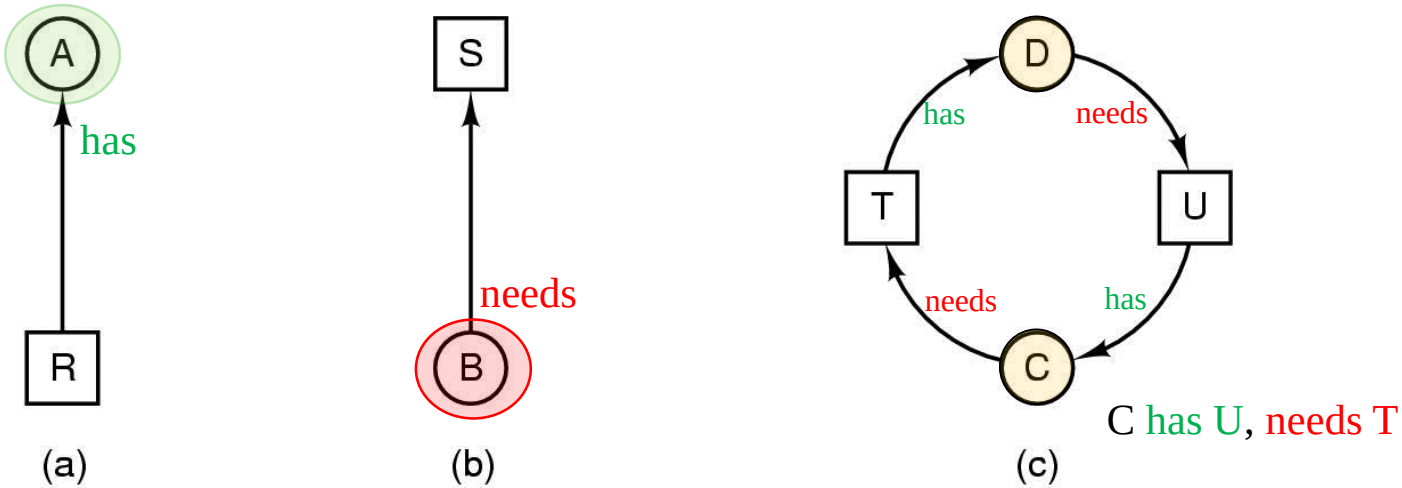
Deadlock Issues

General strategies for dealing with deadlocks

- Just ignore the problem altogether (…and it will go away)
- Detection (and recovery): Let it happen, detect and recover
- Avoidance and prevention: Avoid, by design…
  - Do resource allocation to avoid situations
  - Negate one of the four necessary conditions for deadlock

Deadlock Modeling: Resource Graphs

- Model as directed graphs



  - process A holding resource R <in-arrow to process>
  - process B is waiting (requesting) for resource S <out-arrow from process>
  - process C and D are in deadlock over resources T and U

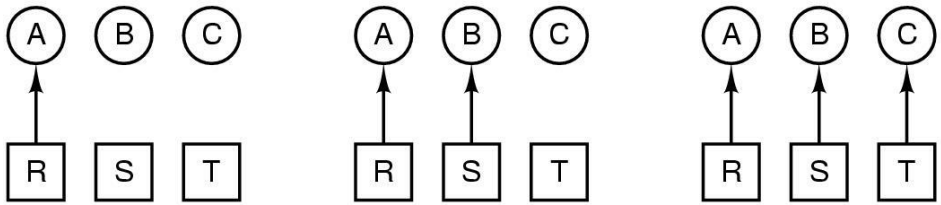
(Visual) Deadlock Modeling

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

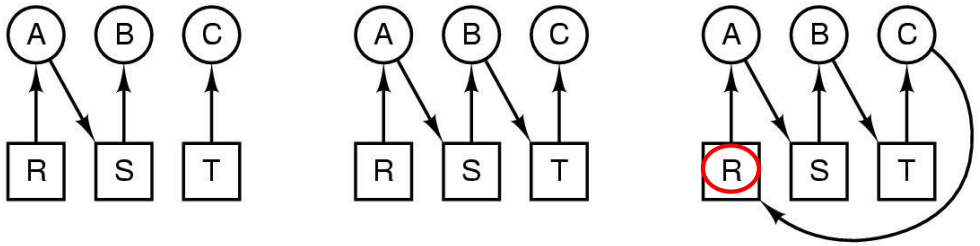
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)

Non-competing A, B, C request patterns – Kernel decides actual allocation



Deadlock Occurrence Based on Request Order

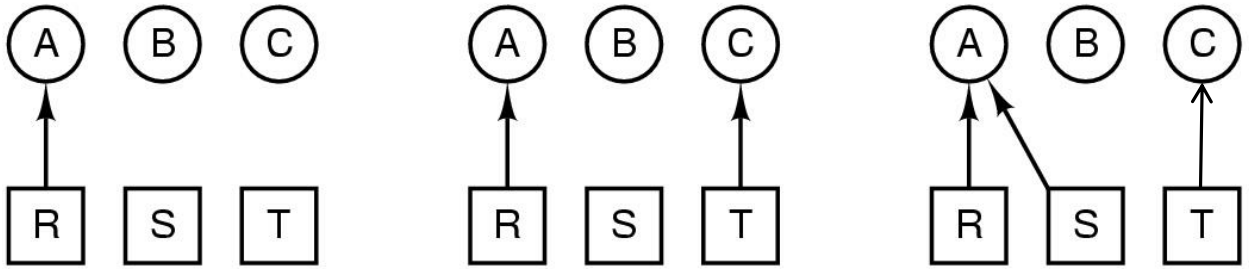


Cycle!

Deadlock Modeling

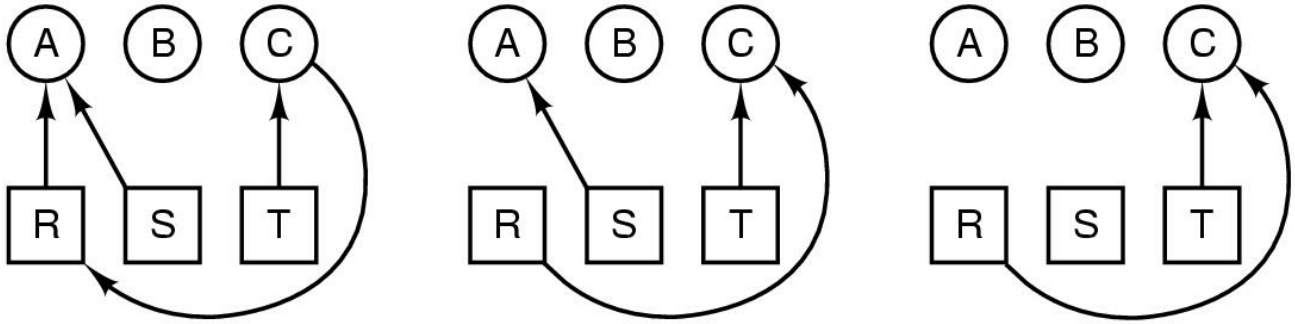
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock



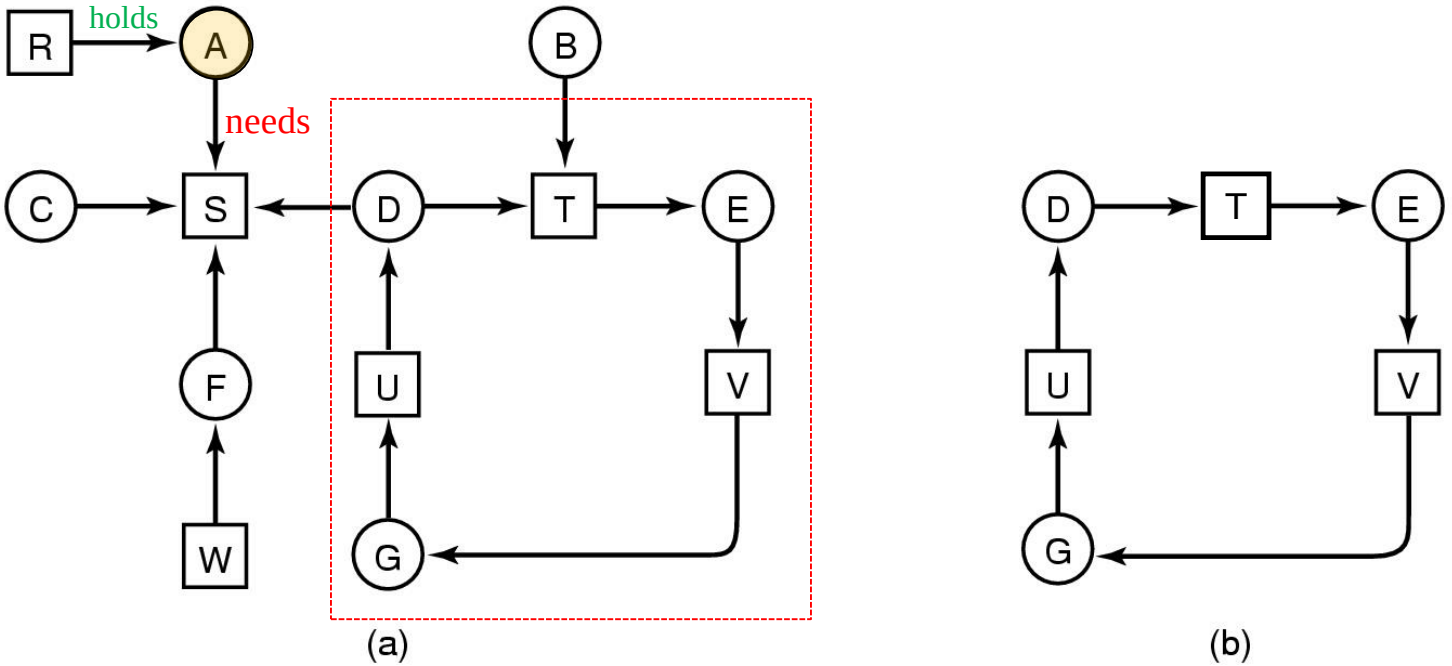
Deadlock Avoidance via Change of Request Order by OS!

Requires OS to see entire pattern first (non on-the-fly allocation) and select order!



Conceptually simple but…: (a) Requests are often dynamic, (b) Tasks usually have precedence relations, priorities, timing durations etc: Partial Order & Scheduling Algs

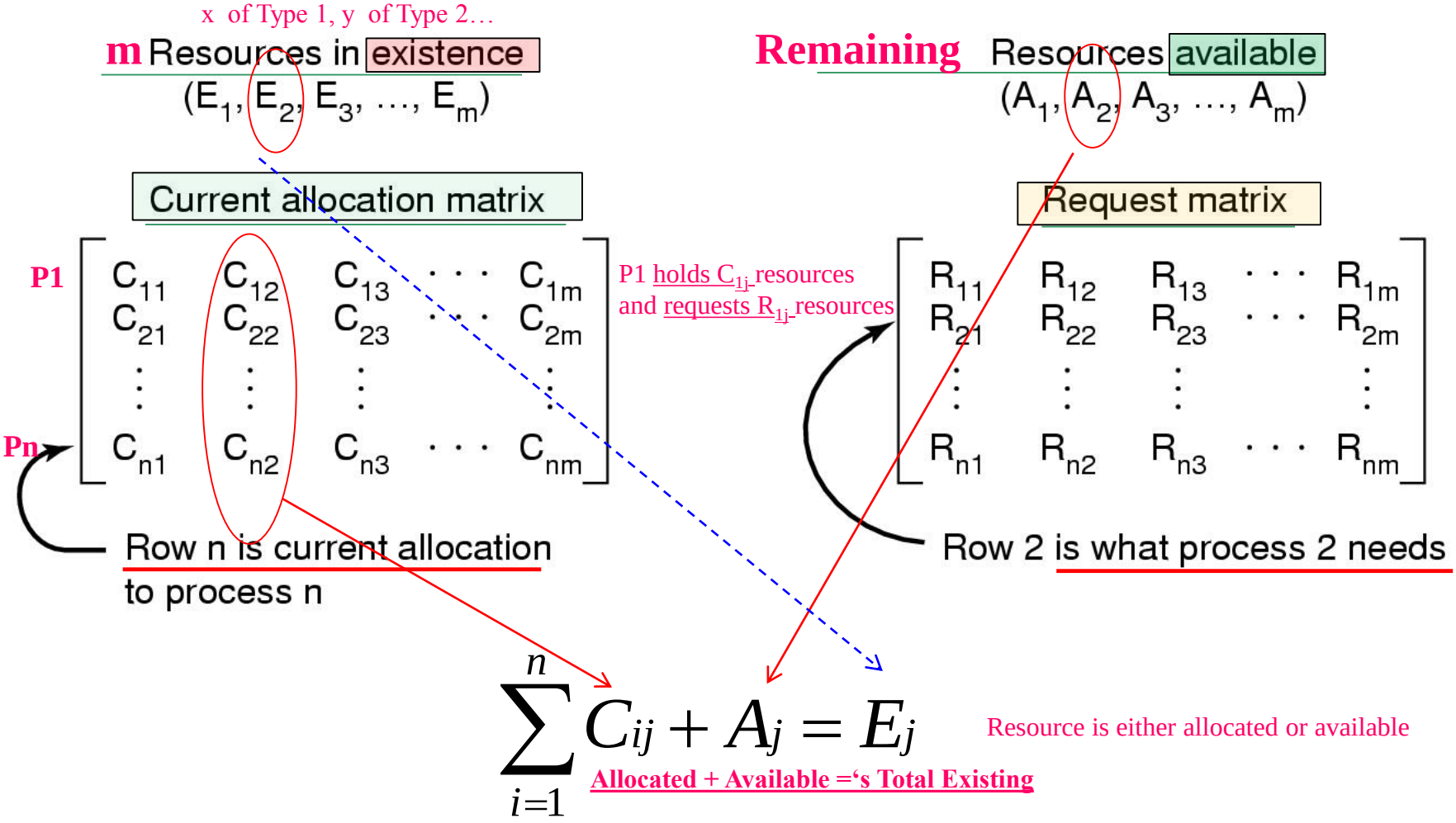
1. **Detection** with One Resource of Each Type



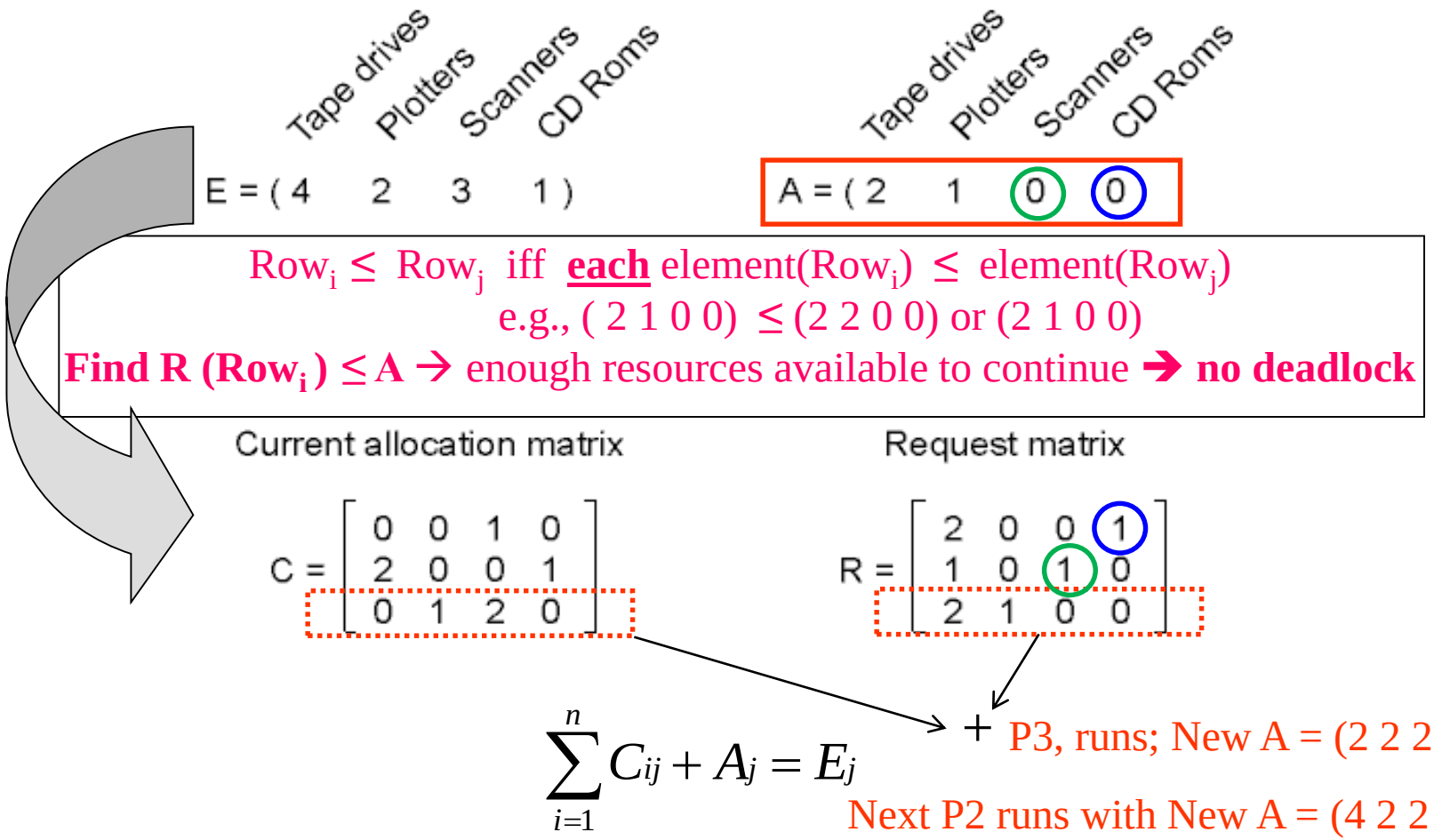
- Develop resource ownership and requests graph (non-dynamic allocations)
- If a cycle can be found within the graph ➔ deadlock

➢ Set up DS (say DFS tree) where cycle is visible by node repetition in DS

Detection with Multiple Resource of Each Type



Detection & Progress via Process Selection!



$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Q: How often to check for free resources? Each check =‘s a New Process!
Costly… plus works only for static & *a priori* known resource requests

- **Each & every** OS has a resource allocation checker that does exactly this static test!

- It is the dynamic testing that varies from
  - rigid (safety-critical OS) to
  - heuristic/adaptive (data farm server load balancing, router paths (hot potato!), internet traffic balancers…)

2. Deadlock **Avoidance***

Requires that the system has some additional *a priori* information available.

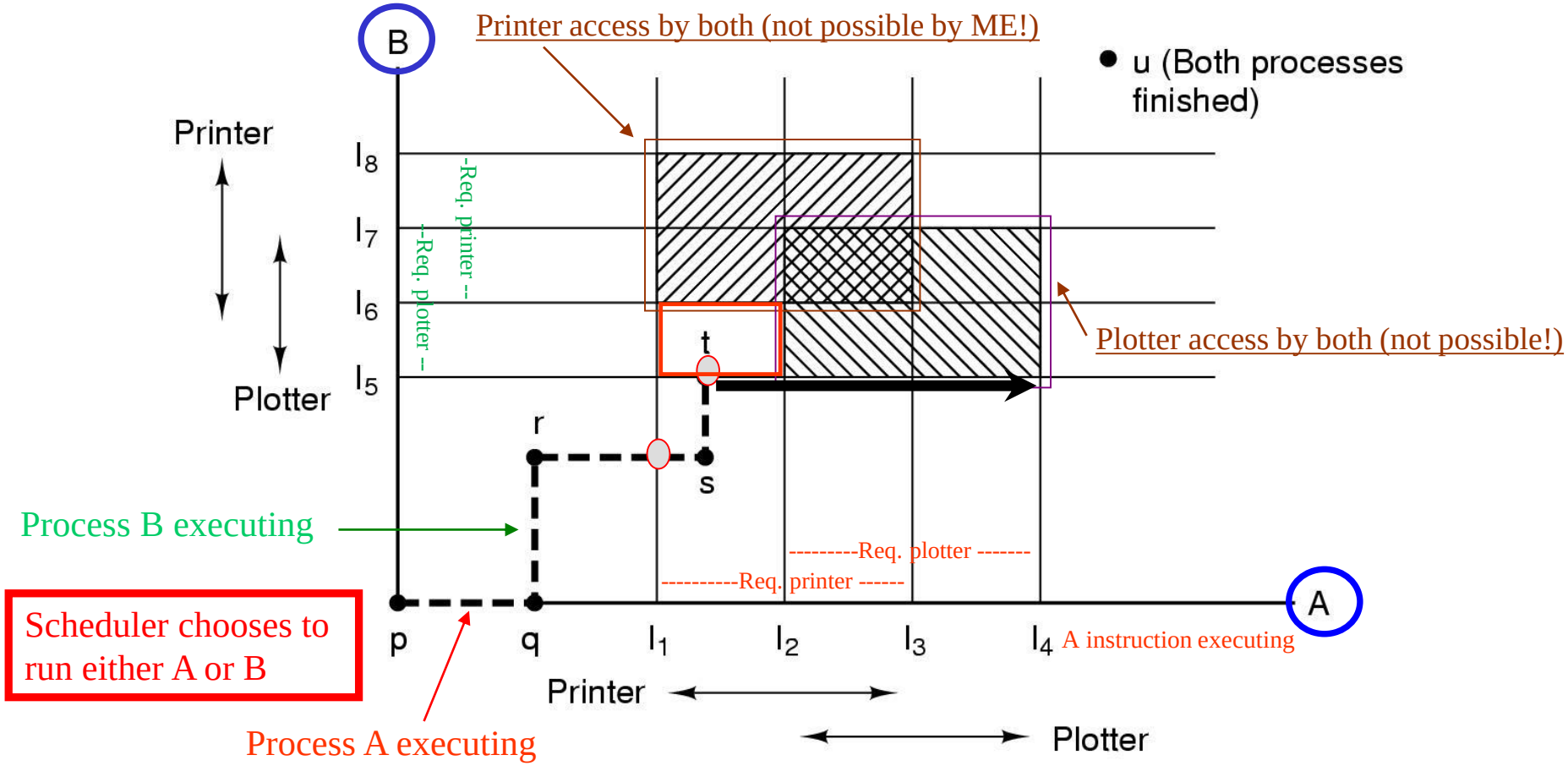
- Simplest and most useful model requires that each process *a priori* declare the *maximum number* of resources of each type that it may need

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition (safe versus unsafe scenarios)

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock **Avoidance**

Execution & Resource Trajectories



- Shaded area ➔ ME condition prevents entry there.
- If trajectory goes into Box(I_1,I_2,I_6,I_5) it only leads to deadlock

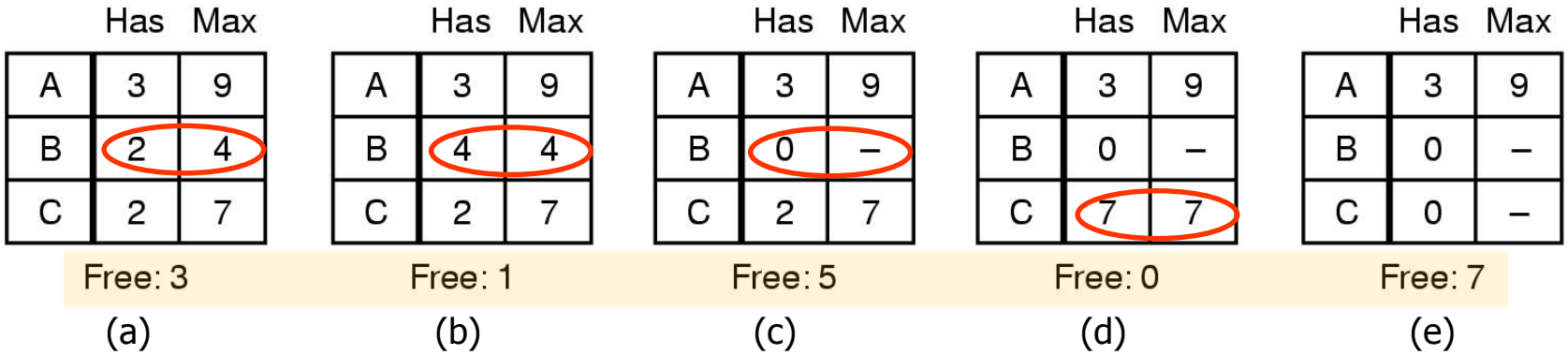
➔ Box @ t indicates "unsafe" state. Thus block B & let A run till I4

Safe and Unsafe States & Execution Orders

Safe: If there is a scheduling order that satisfies all processes even if they all request their maximum resources ?

➔ **Keep in mind that only 1 process can execute at a given time!**

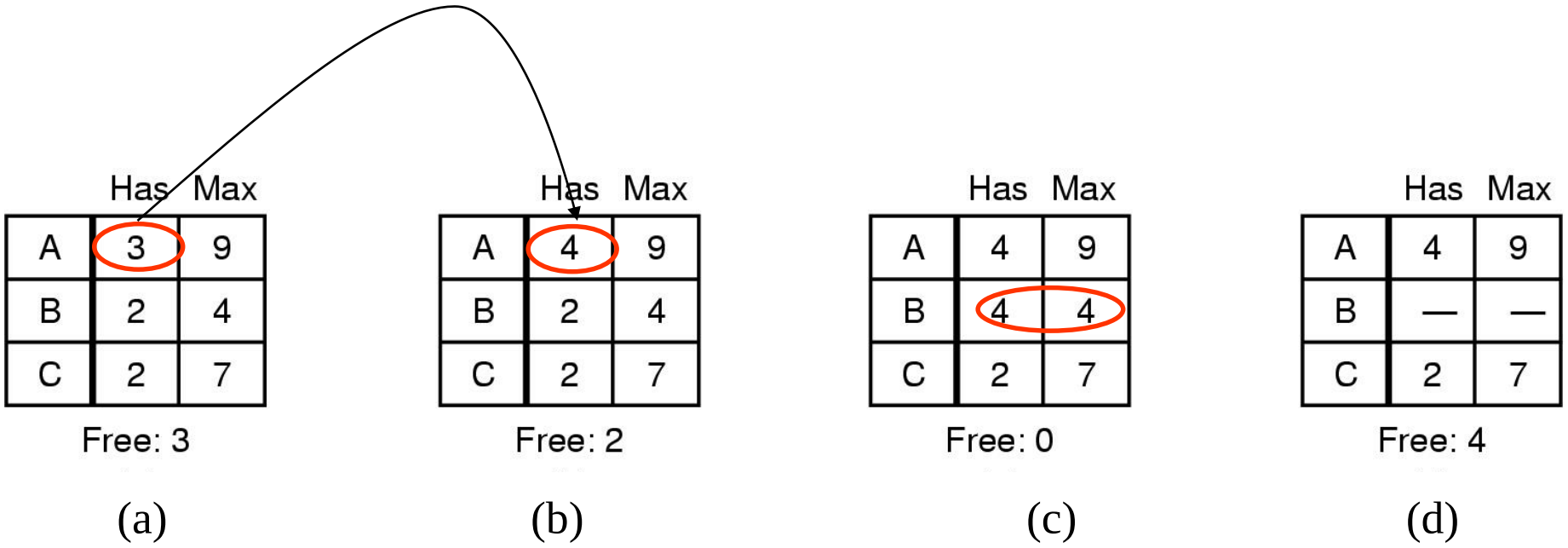
Available Resources = **10**



(a) B runs with 2; asks for (2 more) for max 4 [OK as Free = 3 → B can run; A & C wait for their max]
(b) B gets 2 more; runs & finishes
(c) Free = 5 → C can now get its max of 5
(d) C gets 5; runs & finishes; Free = 7; A needs 6 → A runs

Safe access path starting at (a) <Safe ='s Guarantee for finishing exists>

Unsafe Execution Order



"Potential" deadlock state as both A or C can ask for 5 resources and only 4 are currently free!

Unsafe access starting from (b)

Note: This is not a deadlock – just that the "potential" for a deadlock exists **IF** A or C ask for the max. If they ask for <max, the system works just fine!

Banker's (State) Algorithm for a Single Resource

If **safe order** possible from current "state", then grant access; else deny request

	Has	Max credit
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a) safe

Safe: Can satisfy max A, B, C or D one at a time – safe order exists!

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b) safe

Safe: Can satisfy max C for progress & delay A, B, C. On C finishing, Free = 4 allowing B or D to ask for max (delay A) and then finish max A

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

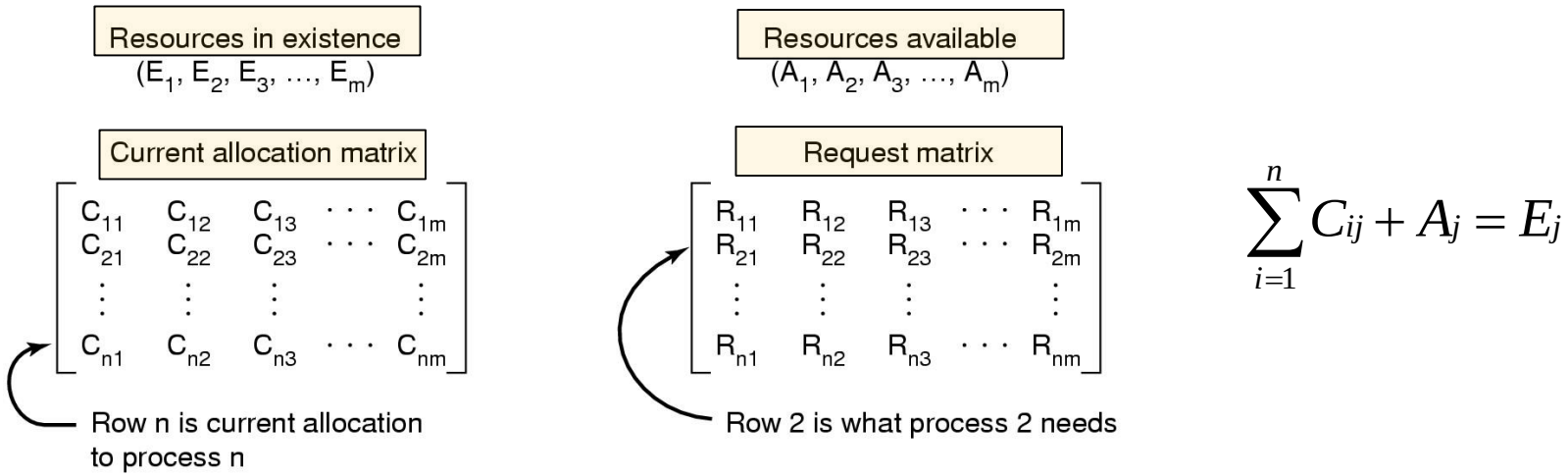
With Free =1, cannot satisfy any max A, B, C request regardless of request order: "potential for deadlock" if all would make requests!

➔Avoiding deadlock by design of permitted safe states!

In reality, criteria for giving "credits/loans" can be based on (a) past behavior of processes (b) bursty traffic analysis (likelihood of run on the bank?), (c) constrained maxima : these form the classical basis for router/server loading decisions for web services!!! DoS attacks (ab)use this!!!

Algorithm for Multiple Resources?

Remember the Deadlock Detection Approach?



$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

1. Look for a row in *R*, whose unmet resource needs are all smaller than or equal to *A* (available resource vector). If no such row exists, the system will eventually deadlock since no process can run to completion.
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the vector *A*.
3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

C = (5 3 2 2)

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
C = (5322)
A = (1020)

If Row (eg D) whose unmet resources are less than A, exists ➔ progress

B req Scanner; 1 < 2; OK, allows D, A, E, C to finish

E wants Scanner (after B); (1020) reduced to (1000) holding up Others ➔ defer E

Is this algorithm useful in practice?

Is it realistic for a process to declare its max resource needs in advance?

Notes:

- Do these schemes work for dynamic requests?
- Do these schemes work for distributed node/locks?
  - Difficult: DS's need a progressive series of functions to make this happen, and mostly about getting distributed states consistent in order to make meaningful alignments.
  - Distributed consensus – Paxos/GFS; Zookeeper, …

3. **Recovery** from Deadlocks

- Recovery through preemption
  - Take a resource from an executing process
    - Depends on nature of the resource (bandwidth, memory) and if the process is interruptible (computation *vs* I/O *vs* RT deadlines)
- Recovery through rollback or back-offs
  - Checkpoint a process periodically
  - Use this saved state
  - Restart the process if it is found deadlocked
- Recovery through killing processes (Reactive or Proactive)
  - Crudest but simplest way to break a deadlock
    - Kill one of the processes in the deadlock cycle; release resources
    - ❖ Choose process that "can" be re-run (either from beginning or a checkpoint) and does not hold up other active processes with precedence dependencies (eg. I/O, transaction chain)!

4. Deadlock Prevention

Constrain the ways in which requests can be made.

- **Mutual exclusion:** Each resource is either currently assigned to exactly one process or is available
- **Hold and wait:** A process holding at least one resource has requested and is waiting to acquire additional resources currently held by other processes
- **No preemption:** A resource cannot be acquired forcibly from a process holding it; can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** A circular chain of >1 processes exits! There exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

A. Attacking the Mutual Exclusion Condition

Mutual exclusion: Each resource is either currently assigned to exactly one process or is available

Change: Allow multiple access to a resource (via serializer)

- Spooled/buffered printer access via printer daemon
  - only the (single) printer daemon uses printer resource (no contention!)
  - deadlock for printer eliminated

** Not all devices can be spooled (spooler contention deadlock?)

** Not all processes can be (fully) buffered, or daemon policy may not support streams.

In general:

- Minimize shared resource assignment (Realistic? Trends → Concurrency)
- As few processes as possible actually claim the resource (Dynamic alloc?)

B. Attacking the Hold and Wait Condition

Hold and wait: A process holding at least one resource has requested and is waiting to acquire additional resources currently held by other processes

To break: Guarantee that whenever a process requests a resource, it does not hold any other resources.

- Require process to request & be allocated all its resources "before" it begins execution (no wait), or allow process to request resources only when the process has none.
  - May not know required resources at start of run plus dynamic cases?
  - Ties up resources other processes could be using → low resource utilization; starvation possible.

- Variation:
  - Process must give up all resources; then request all immediately needed
  - ➢ Realistic?

C. Attacking the No Preemption Condition

No preemption: A resource cannot be acquired forcibly from a process holding it; can be released only voluntarily by the process holding it, after that process has completed its task

- **Changes**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- Not always a viable option to preempt – depends on resource & process type
  - Process given printer access & forcibly take away printer halfway through its job???

D.Attacking the Circular Wait Condition

Circular wait: A circular chain of >1 processes exits!

Option 1: No chain - Process entitled to only 1 resource at a time; Has to give up current resource before requesting another.

→ Process wants to copy large file from memory to printer – so ??? Fails

Option 2: Requests granted in some numerical or precedence sequence (e.g next request only to a higher numbered resource)

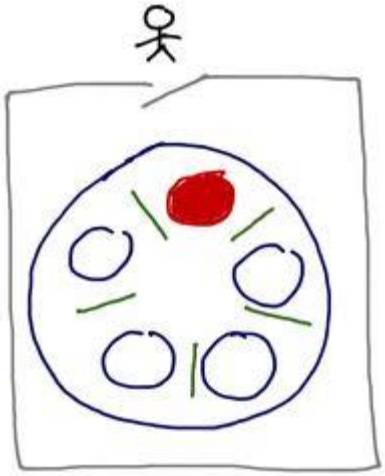
→ Poor efficiency - long access times; varied resource response characteristics

→ Ordering complexity with large resource set or short request/access patterns

→ Consistent global numbering across distributed systems?

Alternatives?

- Algorithms to allocate a resource
  - E.g., let's give resources to shortest job first
  - Works great for multiple short jobs in a system
  - May cause indefinite postponement of long jobs even though not blocked (starvation?)



- Other Solutions?
  - First-come, first-served (FIFO) policy
  - Shortest job first, highest priority, deadline first etc…

Scheduling Policies!