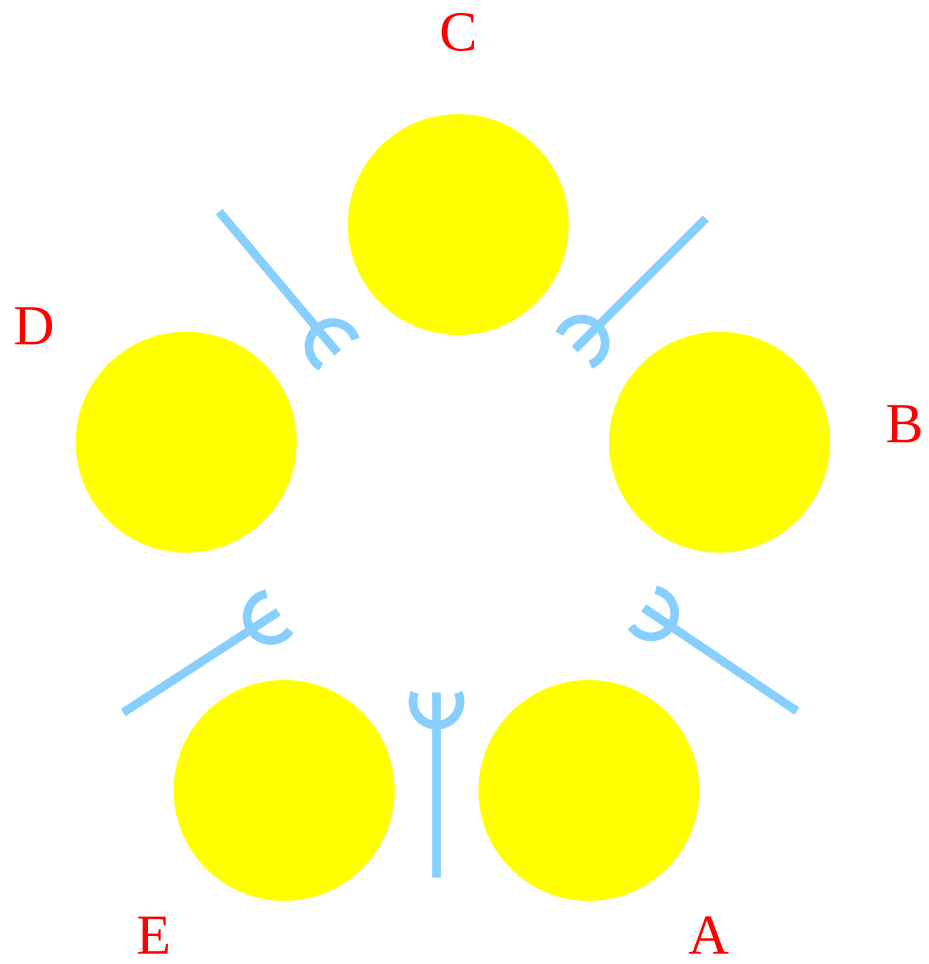
Classic Synchronization Problems

- Invented to demonstrate synchronization primitives
- Not necessarily analogous to real-world problems
- Do illustrate principles real solutions should use

The Dining Philosophers

- Five philosophers sitting around a table
- Each has a plate of spaghetti
- There are forks between each pair of plates
- Philosphers need two forks to eat

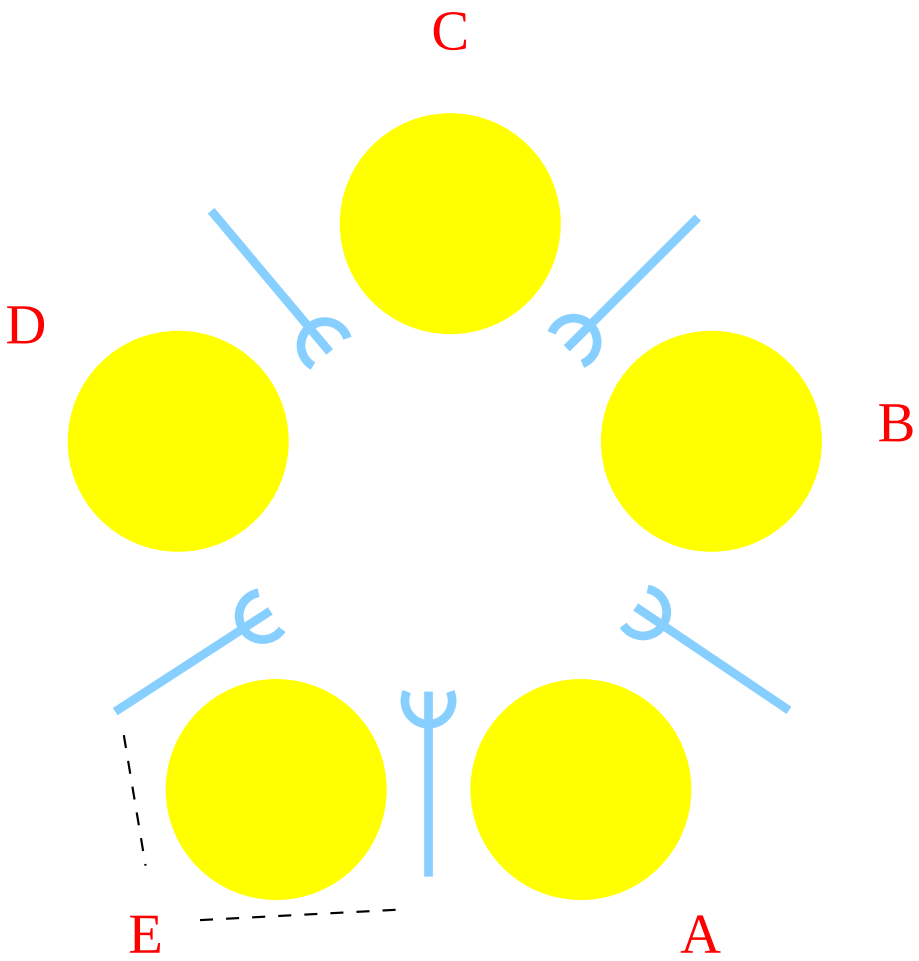
The Table



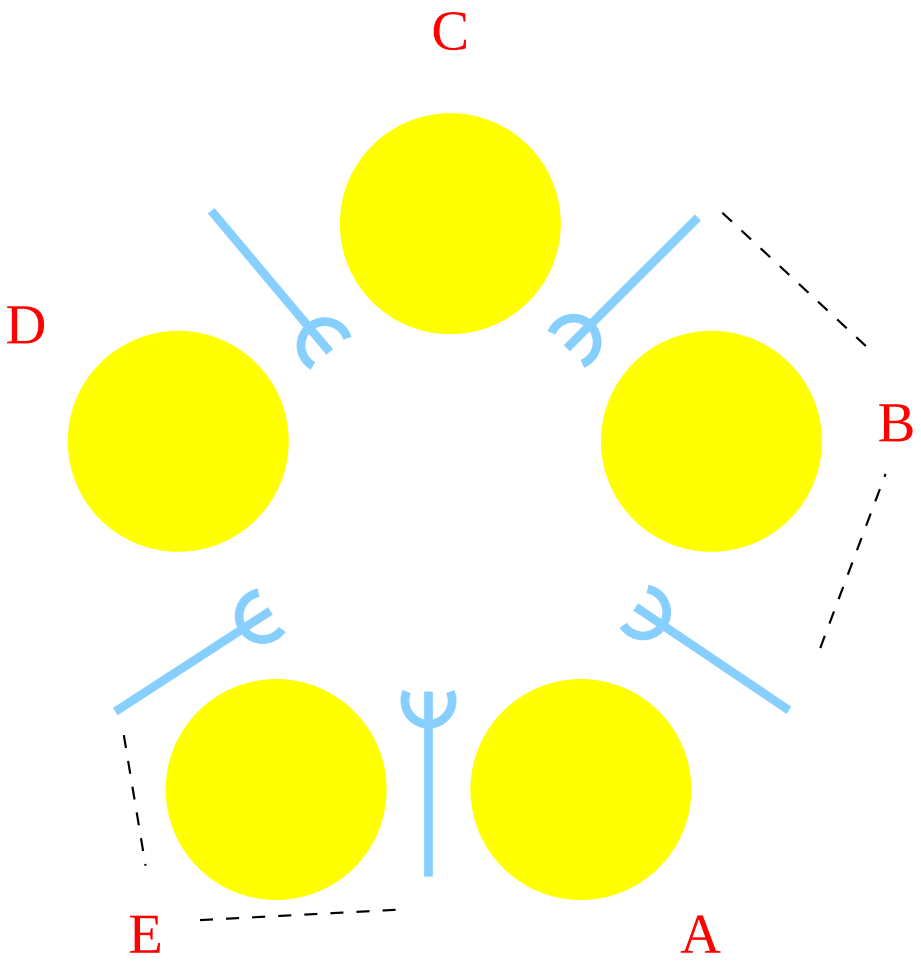
Thinking and Eating

- Philosophers spend their lives eating spaghetti and thinking
- When they're thinking, they don't need any implements
- To eat, they need to pick up the fork on their left and the fork on their right

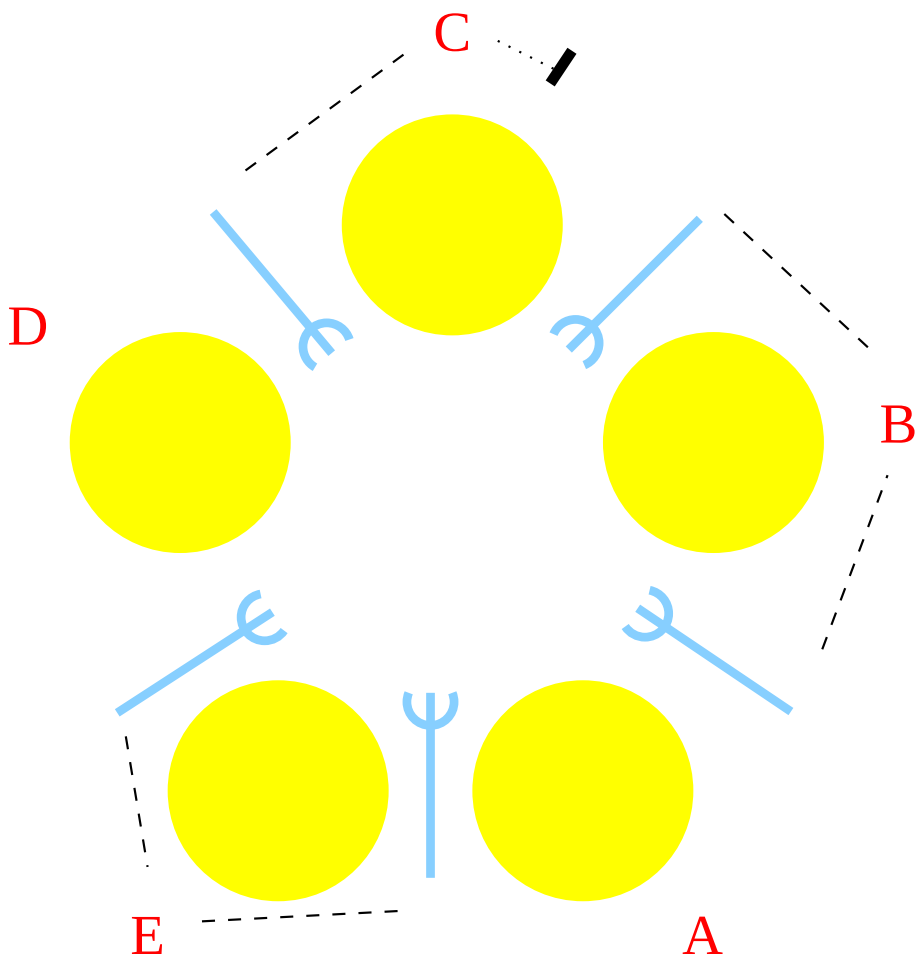
Eve is Dining



Eve and Bob are Dining



But C is Blocked



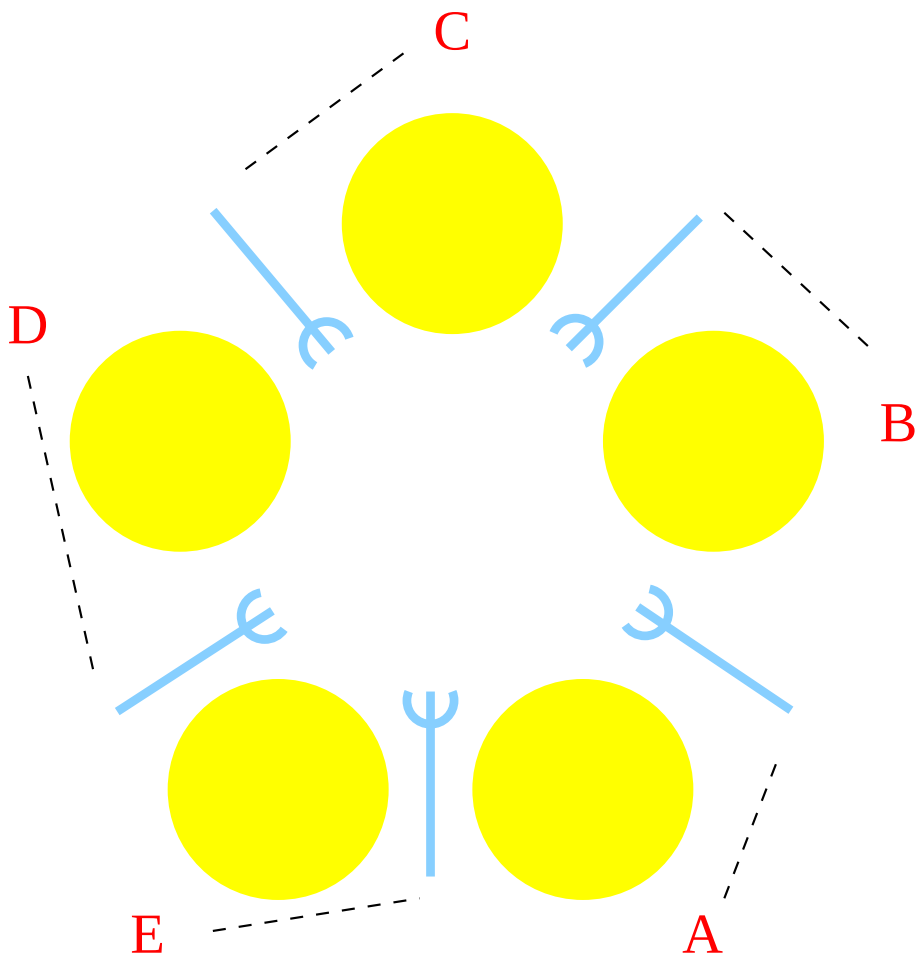
The Problem

- Find an algorithm by which they can all eat
- Must avoid deadlocks

A Bad Solution

```
void philospher(int i)
{
    while(TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

Everyone Grabs One Fork



Repairing the Solution

- Instead of **`take_fork((i+1) % N)`**, see if it's available first
- (Of course, that check and the grab are done inside a critical region)
- If unavailable, sleep for a while and retry
- Still no good — what if everyone tries the left fork, waits, retries simultaneously, etc.
- Everyone can run, but no one makes progress: *starvation*

Randomization

- Suppose the philosophers sleep for a random time if they fail to get the second fork
- Usually, that will work fine
- Some applications require a *guaranteed* solution

A Related Quote from Knuth

"Finally, we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible! . . . Therefore hash tables are inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake."

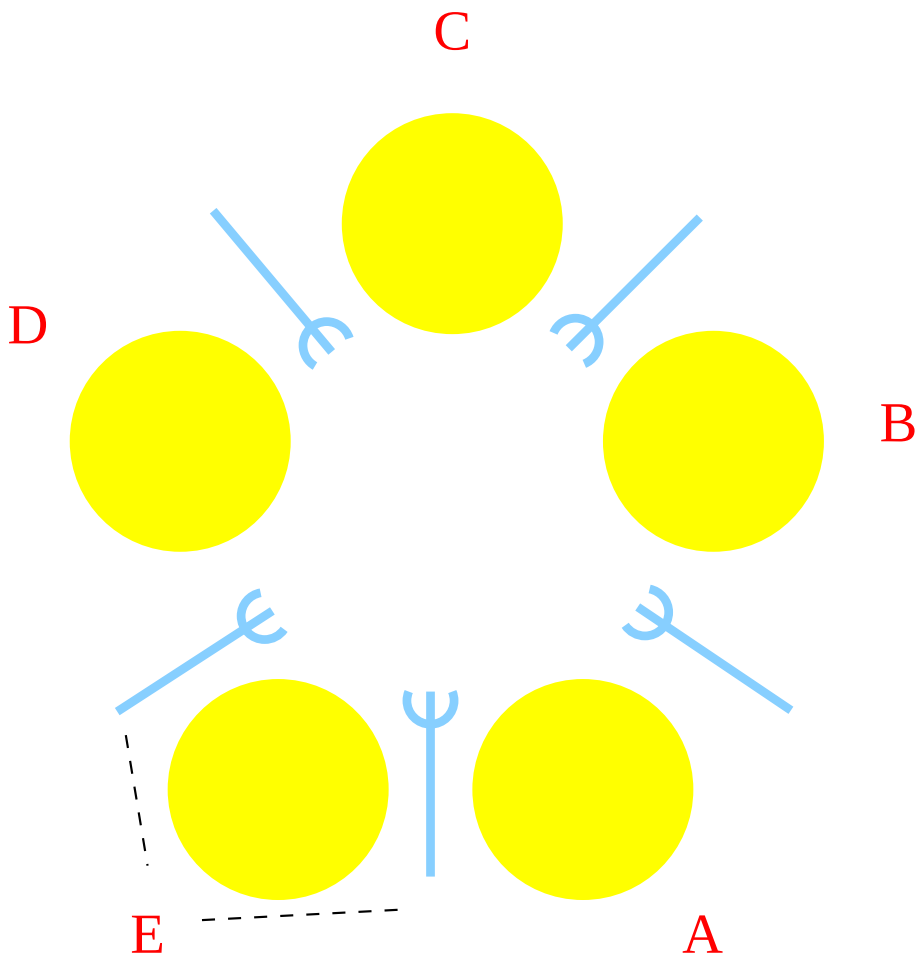
A Simple Guaranteed Solution

- Before taking a fork, a philosopher locks a mutex
- The philosopher can then take two forks, with no interference
- When done eating, the forks are replaced and the mutex is released

What's Wrong?

- As presented, only one philosopher can eat at a time
- Let's modify the algorithm: suppose that the philosopher — say, Eve — releases the mutex before eating
- What happens now? Does this solve the probelm?

Eve is Dining



No

- Alice, Bob, Carol, and Dave are now contending for the other three forks
- We're back to the same problem: how do we solve the problem fairly?

Principles of a Good Solution

- Won't go over the code here in detail
- Minor point: lock a mutex while trying to grab forks
- If they're both available, eat
- If one isn't available, mark yourself hungry and sleep
- When replacing forks, check if neighbors are hungry; if so, wake them

Why This Works: A Sketch of a Proof

- Suppose Eve is eating
- Alice and Dave can't eat; at most one of Bob and Carol can eat
- Assume Bob and Carol are thinking. Both Alice and Dave will wake up when Eve finishes, so things will obviously progress. Therefore, assume that Carol is eating, too.
- Everyone else who wants to eat is blocked
- When Eve finishes, Alice and Dave are awakened. Dave is still blocked, but Alice can run
- When Carol finishes, Bob can run
- Note: must make assumption that eating is a finite activity, and that there is a queue on each semaphore

Readers and Writers

- Suppose we have a database
- Any number of processes can read from it simultaneously; only one process can write to it at once
- No one can read while writing is taking place

Solution Sketch

- Keep track of how many readers there are (use a mutex)
- When the first reader comes along, lock the database mutex
- For subsequent readers, just count them
- When all readers are gone, release the database mutex
- When a writer comes along, lock the database mutex
- The write will block until all readers are done
- Danger: what if readers keep coming forever?

Lessons Learned

- Simple mutexes are not enough; we often need more than one
- Complex algorithms are hard to write and hard to understand — are they correct?
- Define how much “fairness” and “efficiency” you need — things like starvation and indefinite overtaking may be the real problems
- Watch for deadlocks

Deadlock — The Deadly Embrace

- Alice has taken the scanner, and wants to use the printer
- Bob has taken the printer, and wants to use the scanner
- Neither can progress until the other does
- Oops

History

- Studied a lot in the early days of operating systems
- Beloved by theorists
- Today, less an OS issue than an application issue, especially with databases
- Techniques are generally applicable

Resources

- Physical devices
- Database records
- Sharable things like main memory
- Highly system-dependent

Preemption

- Sometimes, an allocation can be preempted
- (If Alice outranks Bob, she can take the printer from him; he'll get both devices when she's done)
- Deadlocks don't occur if resources can be preempted
- Whether or not preemption can occurs depends on the nature of the resource — a printer isn't preemptable in the middle of a job; main memory is if there's a swap area on disk

Sequence of Operations

- Request a resource; loop or block on failure
- Use it
- Release it
- The danger comes with two or more resources

```
process_A() {
    down(&resource_1);
    down(&resource_2);
    use_resources();
    up(&resource_1);
    up(&resource_2);
}
process_B() {
    down(&resource_1);
    down(&resource_2);
    use_resources();
    up(&resource_1);
    up(&resource_2);
}
```

```
process_A() {
    down(&resource_1);
    down(&resource_2);
    use_resources();
    up(&resource_1);
    up(&resource_2);
}
process_B() {
    down(&resource_2);
    down(&resource_1);
    use_resources();
    up(&resource_1);
    up(&resource_2);
}
```

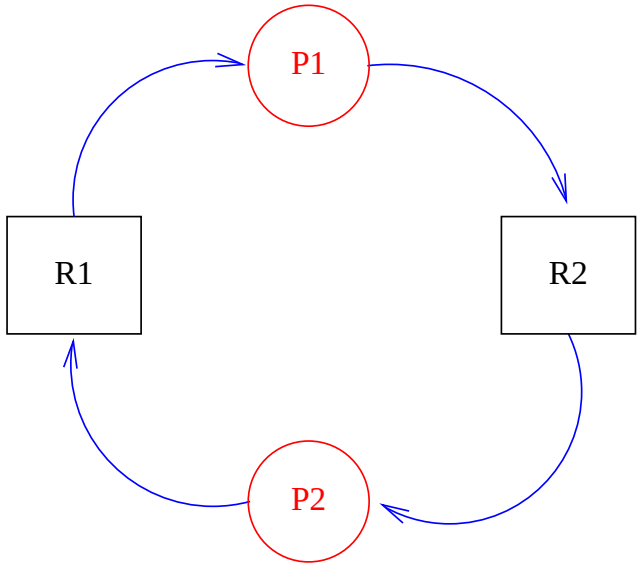
Conditions for Deadlock (Coffman, 1971)

1. Mutual exclusion: Each resource is either available or assigned to exactly one process
2. Hold and wait: Processes holding resources can ask for more
3. No preemption: Resources cannot be taken away from a process involuntarily
4. Circular wait: There must be a chain of two or more processes waiting for the next member of the chain

If any of these conditions aren't present, deadlocks can't occur. Note that these conditions are tied to operating system policy.

Modeling a Deadlock with Directed Graphs

- An arc from a resource to a process means the resource is allocated to the process
- An arc from a process to a resource means the process is blocked waiting for the resource



Avoiding Deadlock

- An operating system can use this to watch for, and avoid, deadlock
- When a resource request occurs, see if granting it will create a cycle
- If so, delay granting that resource; run another process instead
- Wait for some other resources to be released — arcs to be cut — before granting the resource

General Philosophies

- Ignore the problem — maybe it isn't serious enough
- Detect it and recover
- Avoid it by careful resource allocation
- Prevent it by avoiding one of the four conditions

Ignoring Deadlocks

- Perhaps the chances of a deadlock are low
- Example: my desktop can have 532 processes running, and 1772 open files; each process can open 256 files
- Suppose every process that wanted to open a file and couldn't would sleep for a while and retry
- Easy to see that a deadlock could occur if lots of processes open lots of files
- Realistically, it just doesn't happen
- The problem is avoidable, but only at high cost

Detection and Recovery

- Simple case: one resource of each type
- Construct the digraph discussed earlier
- See if any cycles exist in the graph (algorithms are well-known)

Complex Case: Resource Classes

- There is more than one of certain resources (i.e., several printers)
- Let E_j be the total number of resource j; let A_j be the number unallocated.
- Create two matrices, C (current allocations) and R (requests). In each, row i corresponds to process i
- Invariant over n processes:

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

- That is, all instances of resource j plus the number of j allocated is the total
- Vector comparison: $A \leq B$ iff $\forall j, A_j \leq B_j$

Algorithm

Start with all processes “unmarked”. Marked processes are able to run to completion

1. Find an unmarked process P_i for which $R_i \leq A$ (i.e., a process whose resource requests can be satisfied now)
2. If found, let $A + C_i \rightarrow A$ (when this process finishes, its resource will become available); go back to step 1
3. If there are no such processes, the algorithm terminates

When the algorithm terminates, all unmarked processes are deadlocked

Example

A system has 4 tape drives, 2 plotters, 3 scanners, 1 CD-ROM

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Process 1 has one scanner; process 2 has two tapes drives and a CD-ROM; process 3 has a plotter and two scanners

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Each process wants more

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Run the Algorithm

- Process 1 isn't runnable; no CD-ROMs are available
- Process 2 can't run; no scanners are available
- Process 3 is runnable; when it's done, $A = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$
- That lets process 2 run; when it's done, $A = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$
- That's enough to satisfy process 1, since $\begin{pmatrix} 2 & 0 & 0 & 1 \end{pmatrix} \leq \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$
- No deadlock!
- Rerun the algorithm with $R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 \end{bmatrix}$
- $\begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \not\leq \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$ — we have a deadlock

Recovery From Deadlock

- If possible, preempt a resource
- Figure out which resources are preemptible, and which process should be delayed
- Checkpoint state of process periodically. When deadlock occurs, roll some process back to before it acquired some resources
- Killing a process: find a process with resource, in the cycle or not, and kill it. Try to pick one that can be rerun without bad side-effects. . .