Deadlocks

Resources

- Something that a process uses
  - Hardware: memory, CPU, printer, …
  - Software: data structure
- Preemptable resources

  Can be removed from a process and restored later (e.g. memory as long as you save a copy)
- Nonpreemptable resources

  Removing resource would cause failure (e.g. ejecting a removable file system during a write)

Resources

- Ownership
  - Resources usually managed by OS, but not always
  - The buffer in a producer-consumer problem is a process-owned resource

Acquisition & Release

- Resource owner provides:
  - Acquire resource
  - Release resource
- Resource owner is responsible for releasing processes held by a process when it exits

Dining philosophers (2.5.1)

- Dijkstra's resource management problem
- Philosophers think and eat, but need two utensils to eat.
- How do we get them to eat without starving?

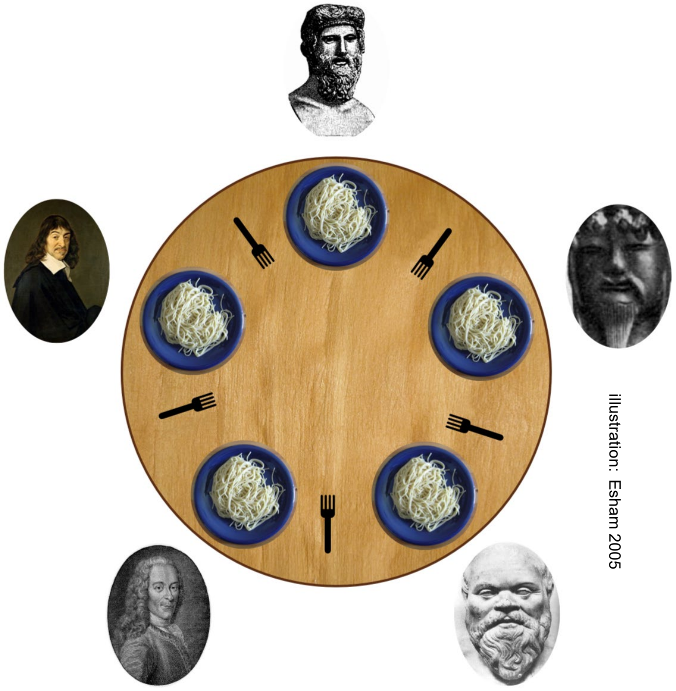


illustration: Esham 2005

Naïve implementation

```
N is number of philosophers

/* code for ith philosopher */
philosopher(i) {
        while (true) {
                think();  // deep thoughts…
                get_utensil(i);  // one on left
                get_utensil((i+1) % N) // one on right
                eat();    // fuel the brain (expensive organ)
                // put down utensils
                release_utensil(i);
                release_utensil((i+1) % N);
        }
}
```

With semaphores

```
// One to the left, one to the right
left(i) {return (i+N-1) % N;}
right(i) {return (i+1) % N;}

shared int state[N];  // all initialized to THINKING
shared semaphore mutex = 1;
shared semaphore s[N];  // Per philosopher sem init to 0.

philosopher(i) {
          think();
          take_utensils();
          eat();
          release_utensils();
}|
```

with semaphores

```
take_utensils(i) {
          mutex.down();  // critical section
          state[i] = hungry;
          test(i);  // increment semaphore if we’re good
          mutex.up(); // exit critical section
          s[i].down();  // blocks if no forks
}

test(i) {
          if (state[i] == hungry &&
             state[left(i)] != eating & state[right(i)] != eating) {
                    state[i] = eating
                    s[i].up();
          }
}
```

with semaphores

```
release_utensils(i) {
          mutex.down();  // critical section
          state[i] = thinking;
          // if neighbors were blocked, we might be able
          // to release them
          test(left(i));
          test(right(i));
          mutex.up(); // exit critical section
}
```

Deadlocks

We have looked at examples of these through the semester

Wikipedia

Everyone pick up the right chopstick
Everyone pick up the left chopstick…

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Coffman's conditions for deadlock

- Mutual exclusion. Each resource is either currently assigned to exactly one process or is available.
- Hold and wait. At least one process is holding a resource and is waiting to acquire a resource held by another process.
- No preemption. Resources already granted to a process may only be released by that process.

Detecting deadlocks

Photo: Ripley’s Believe it or Not
(ostriches don’t really do this)

This is what most operating systems do

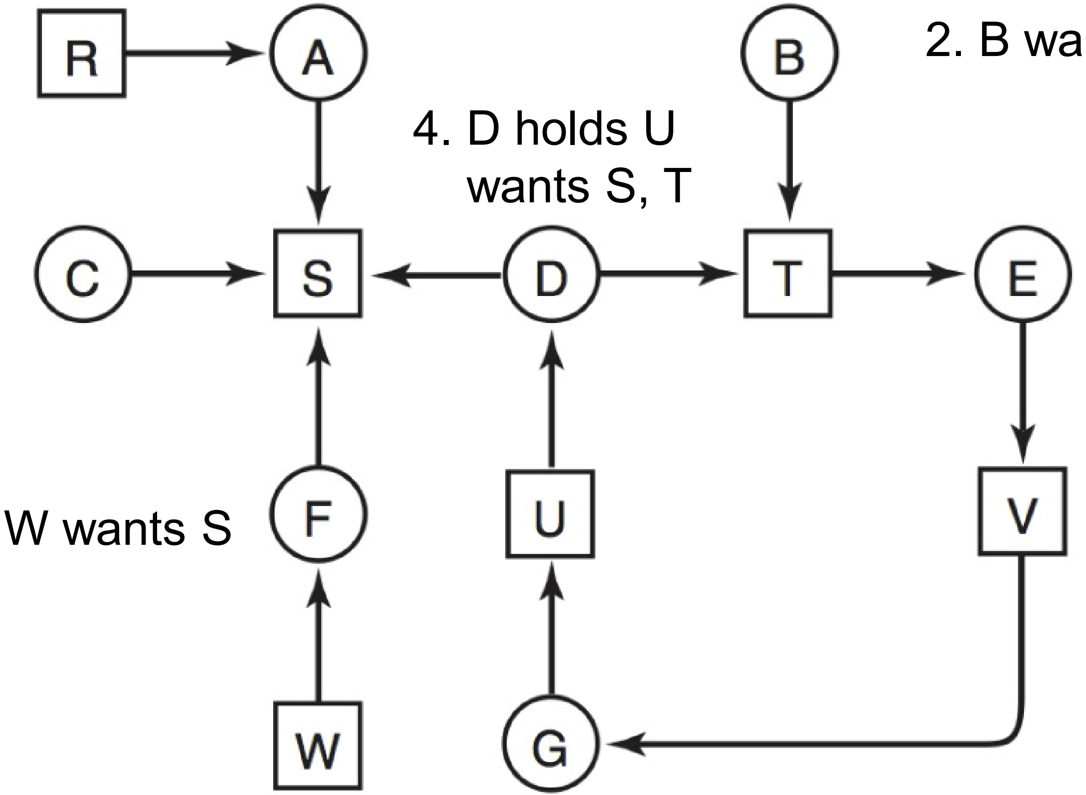
Deadlock strategies

- Ostrich algorithm – do nothing
- Detection and recovery
  - Allow deadlocks to occur
  - Run triggered/scheduled deadlock detection
  - Take corrective action, e.g. kill process
- Negate one of Coffman's conditions to prevent deadlocks from occurring

○ Process

□ Resource

Deadlock detection
Resource allocation graphs



1. A hold R wants S

2. B wants T

3. C wants S

4. D holds U wants S, T

5. E holds T wants V

6. F holds W wants S

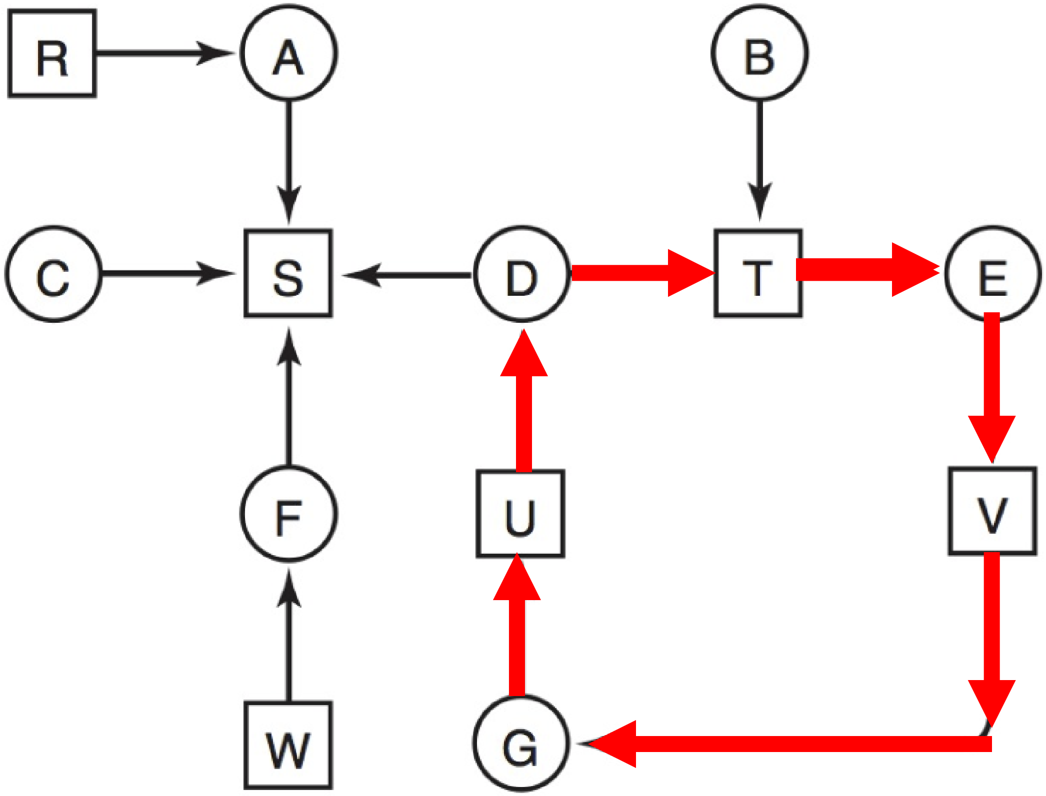
7. W holds V wants U

Fig. 6-5 Tanenbaum & Bos

Is this system deadlocked?

Cycle indicates deadlock

DEG deadlocked!



If we maintain a directed resource graph, we can use a cycle checking algorithm to detect a deadlock.

Preventing deadlocks

- Not practical to negate:
  - mutual exclusion
  - no preemption
- Leaves us with 2 remaining conditions to consider:
  - Hold & wait
  - No circular wait

Negating hold and wait

- All resources must be requested at the same time.
- If we need resources dynamically… each time we need a new resource:
  - Release all held resources
  - Acquire new set that is needed

Negating circular wait

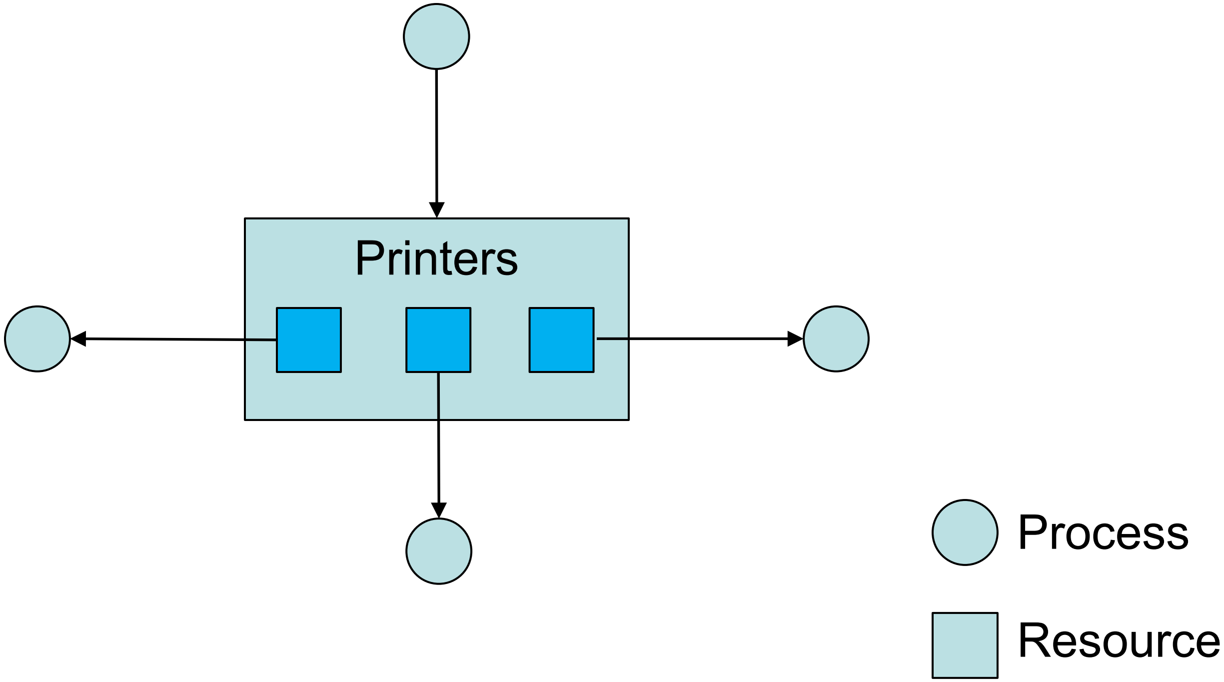
- An ordering is defined on resources (e.g. they are numbered).
  - If a process needs resource 1, 7, and 9, they must be acquired in that order.
  - If the process later needs resource 8, it must release 9 before acquiring 8.
- Breaks circular wait, but makes it very hard to write portable code

Multiple instances of resources

A printer room at
Curtin University, Australia

Multiple instance of resources

Resource graphs now have multiple instances



Multiple instance deadlock detection

Existing resources

$$E = \left(\begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{Blu-rays} \\ 4 & 2 & 3 & 1 \end{array} \right)$$

Available resources

$$A = \left(\begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{Blu-rays} \\ 2 & 1 & 0 & 0 \end{array} \right)$$

Current allocation matrix

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

Request matrix

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

Each row of allocation and request matrix show what has been allocated to or requested by process I, and

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

Multiple instance deadlock detection

- Define $R_i \leq A$ to mean
  For each requested instance, there are enough resources available ($\forall j \; R_{i,j} \leq A_j$)

```
while (not done) {
  Find an unprocessed row of request matrix $R_i \leq A$
  if (found) {
     Add count of allocations $C_i$ to $A$.  (As we can satisfy $P_i$, its allocated resources will eventually be released and available to others)
  } else {done = true, remaining processes are deadlocked}
}
```