Classical Synchronization Problems

This lecture

- Goals:
  - Introduce classical synchronization problems
- Topics
  - Producer-Consumer Problem
  - Reader-Writer Problem
  - Dining Philosophers Problem
  - Sleeping Barber's Problem

3. The Sleeping Barber

- N customer chairs (waiting chairs)
- One barber who can cut one customer's hair at any time
- No waiting customer => barber sleeps
- Customer enters =>
  - If all waiting chairs full, customer leaves
  - Otherwise, if barber asleep, wake up barber and make him work
  - Otherwise, barber is busy – wait in a chair

Sleeping Barber solution (1)

```
#define CHAIRS 5                  /* # chairs for waiting customers */

typedef int semaphore;            /* use your imagination */

semaphore customers = 0;          /* # of customers waiting for service */
semaphore barbers = 0;            /* # of barbers waiting for customers */
semaphore mutex = 1;              /* for mutual exclusion */
int waiting = 0;                  /* customers are waiting (not being cut) */
```

[From Tanenbaum, Modern Operating Systems]

Sleeping Barber solution (2)

```
void barber(void)
{
  while (TRUE) {
            down(customers);
            down(mutex);
            waiting = waiting - 1;
            up(barbers);
            up(mutex);
            cut_hair();
  }
}
```

```
void customer(void)
{
  down(mutex);
  if (waiting < CHAIRS) {
            waiting = waiting + 1;
            up(customers);
            up(mutex);
            down(barbers);
            get_haircut();
  } else {
            up(mutex);
  }
}
```

Note:

`down` means `semWait`

`up` means `semSignal`

[From Tanenbaum, Modern Operating Systems]

Sleeping Barber solution, plus code comments

```
void barber(void)
{
  while (TRUE) {
            down(customers);                  /* go to sleep if # of customers is 0 */
            down(mutex);                      /* acquire access to ’waiting’ */
            waiting = waiting - 1;            /* decrement count of waiting customers */
            up(barbers);                      /* one barber is now ready to cut hair */
            up(mutex);                        /* release ’waiting’ */
            cut_hair();                       /* cut hair (outside critical region) */
  }
}
```

[From Tanenbaum, Modern Operating Systems]

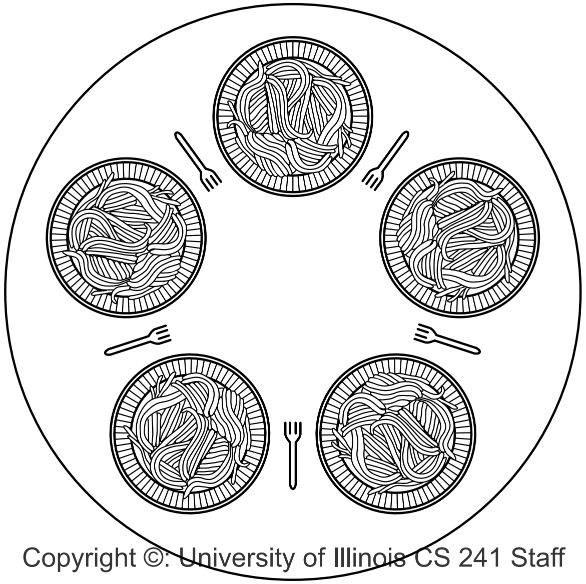
Sleeping Barber solution, plus code comments

```
void customer(void)
{
  down(mutex);                     /* enter critical region */
  if (waiting < CHAIRS) {          /* if there are no free chairs, leave */
          waiting = waiting + 1;   /* increment count of waiting customers */
          up(customers);           /* wake up barber if necessary */
          up(mutex);               /* release access to 'waiting' */
          down(barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut();           /* be seated and be serviced */
  } else {
          up(mutex);               /* shop is full; do not wait */
  }
}
```

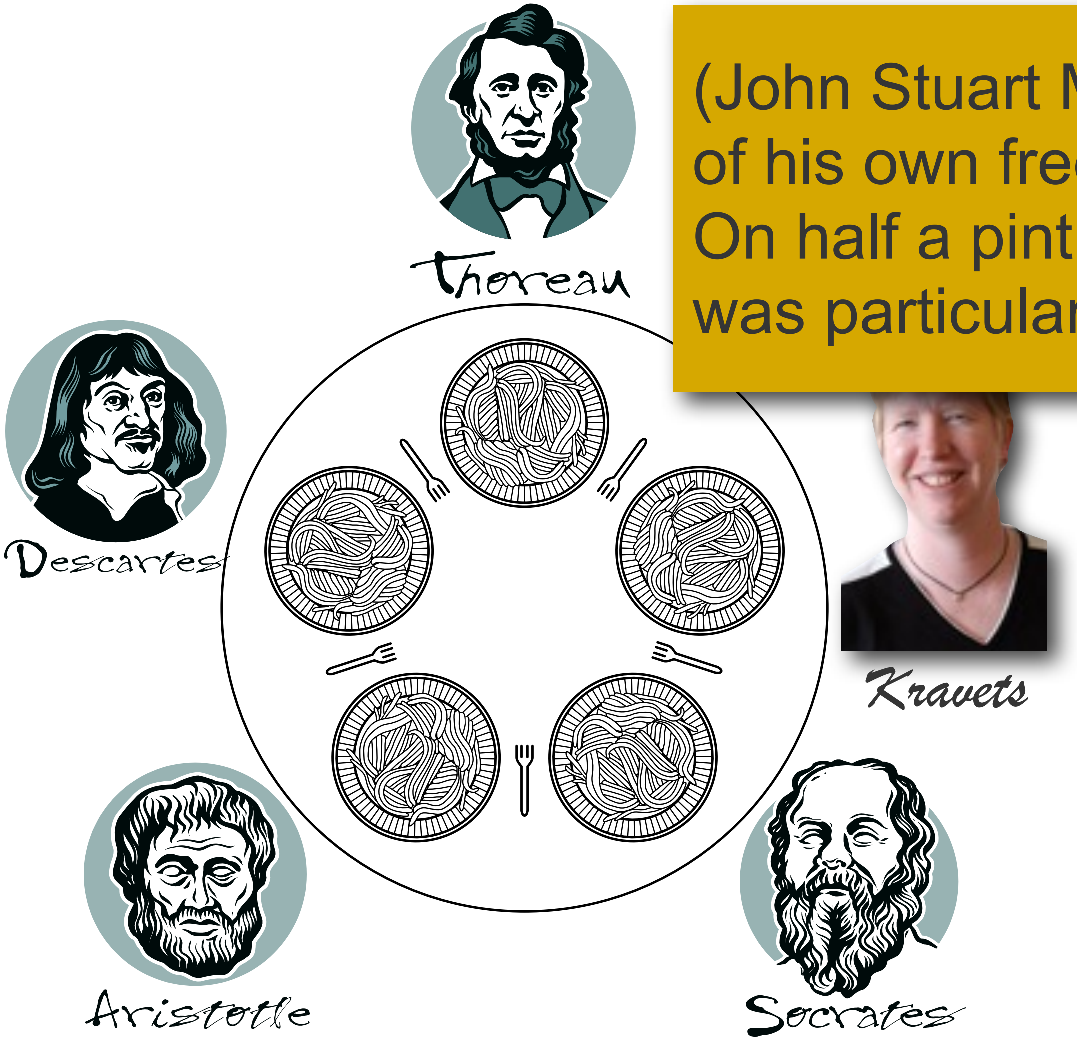
[From Tanenbaum, Modern Operating Systems]

4. Dining Philosophers: an intellectual game

- N philosophers and N forks
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time



4. Dining Philosophers: an intellectual game



(John Stuart Mill,
of his own free will,
On half a pint of shandy
was particularly ill...)

Does this solve the problem?

```
#define N 5                             /* number of philosophers */

void philosopher(int i)                 /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
        think();                        /* philosopher is thinking */
        take_fork(i);                   /* take left fork */
        take_fork((i+1) % N);           /* take right fork; % is modulo operator */
        eat();                          /* yum-yum, spaghetti */
        put_fork(i);                    /* put left fork back on the table */
        put_fork((i+1) % N);            /* put right fork back on the table */
  }
}
```

A **non-**solution to the dining philosophers problem

- Deadlock: everyone picks up their left fork first, then waits for right fork…

[From Tanenbaum, Modern Operating Systems]

Necessary and sufficient conditions for deadlock

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

- Dining Philosophers has all four of these properties.

Necessary and sufficient conditions for deadlock

- Mutual exclusion: Exclusive use of chopsticks
- Hold and wait: Hold 1 chopstick, wait for next
- No preemption: cannot force another to release held resource
- Circular wait: Each waits for next neighbor to put down chopstick

Dining Philosophers solution

```
#define N             5          /* number of philosophers */
#define LEFT     (i-1)%N          /* number of i’s left neighbor */
#define RIGHT  (i+1)%N            /* number of i’s right neighbor */
#define THINKING      0          /* philosopher is thinking */
#define HUNGRY        1          /* philosopher is trying to get forks */
#define EATING        2          /* philosopher is eating */

typedef int semaphore;           /* semaphores are a special kind of int */
int state[N];                    /* array to keep track of everyone’s state */
semaphore mutex = 1;             /* mutual exclusion for critical regions */
semaphore s[N];                  /* one semaphore per philosopher */

void philosopher(int i)          /* i: philosopher number, from 0 to N-1 */
{
  while (TRUE) {                 /* repeat forever */
          think();               /* philosopher is thinking */
          take_forks(i);         /* acquire two forks or block */
          eat();                 /* yum-yum, spaghetti */
          put_forks(i);          /* put both forks back on table */
  }
}
```

[From Tanenbaum, Modern Operating Systems]

Dining Philosophers solution

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N-1 */
{
  down(&mutex);                           /* enter critical region */
  state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
  test(i);                                /* try to acquire 2 forks */
  up(&mutex);                             /* exit critical region */
  down(&s[i]);                            /* block if forks were not acquired */
}
```

```
void put_forks(i)                         /* i: philosopher number, from 0 to N-1 */
{
  down(&mutex);                           /* enter critical region */
  state[i] = THINKING;                    /* philosopher has finished eating */
  test(LEFT);                             /* see if left neighbor can now eat */
  test(RIGHT);                            /* see if right neighbor can now eat */
  up(&mutex);                             /* exit critical region */
}
```

```
void test(i)                              /* i: philosopher number, from 0 to N-1 */
{
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
  }
}
```

[From Tanenbaum, Modern Operating Systems]

What if...

- Made picking up both left and right chopsticks an atomic operation?
  - That works (i.e., prevents deadlock)
  - This is essentially what we just did!
- Or, N philosophers & N+1 chopsticks?
  - That works too!
- And we'll see another solution later...

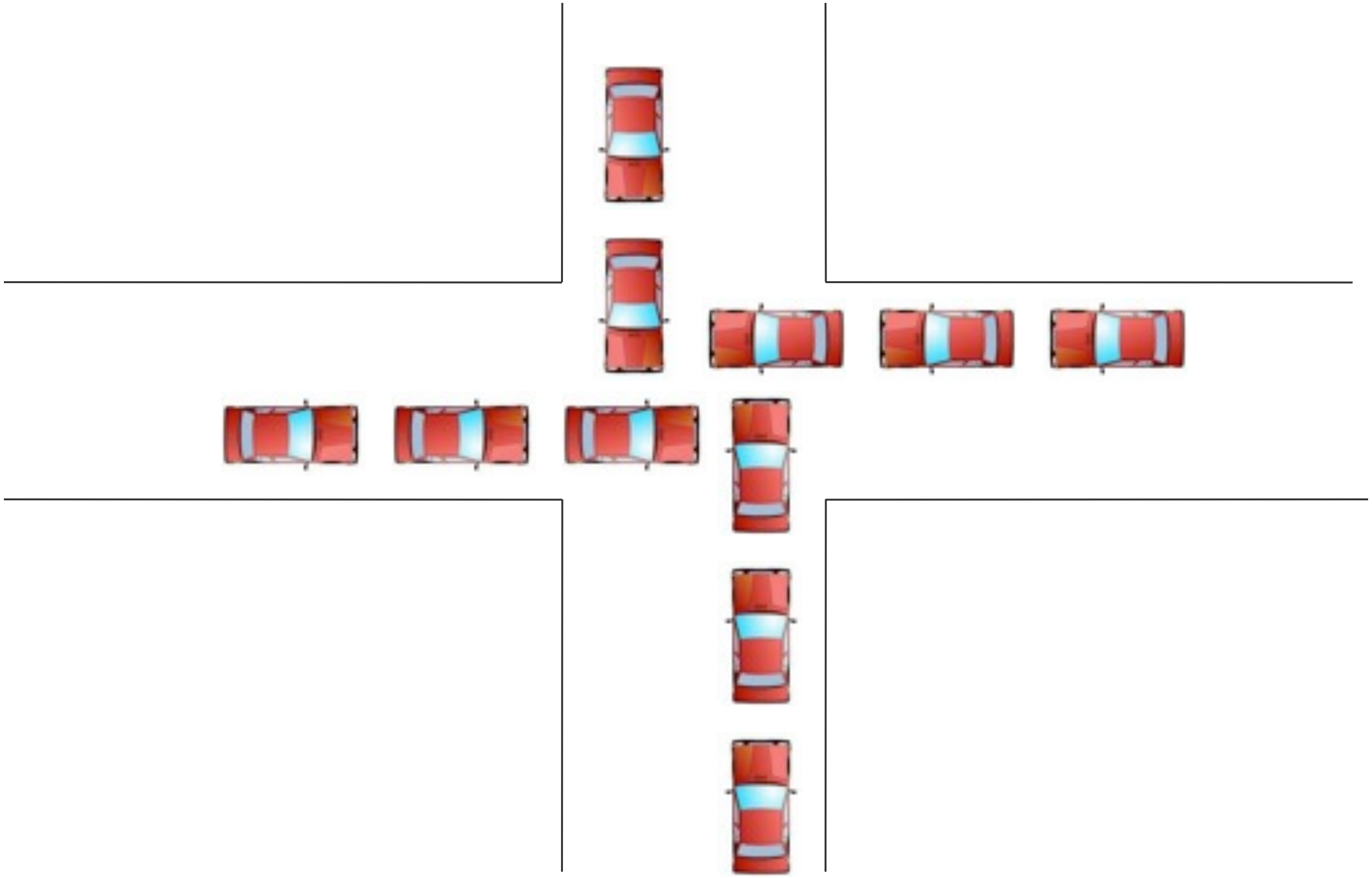
Summary

Classical synchronization problems

- Producer-Consumer Problem
- Reader-Writer Problem
- Sleeping Barber’s Problem
- Dining Philosophers Problem

Deadlocks

Deadlock



Deadlock Definition

- A process is **deadlocked** if it is waiting for an event that will never occur.
    - Typically, but not necessarily, more than one process will be involved together in a deadlock
- Is deadlock the same as starvation (i.e., indefinitely postponed)?
    - A process is **indefinitely postponed** if it is delayed repeatedly over a *long* period of time while the attention of the system is given to other processes. (Logically the process may proceed but the system never gives it the CPU.)

Necessary Conditions for Deadlock

- **Mutual exclusion**
  - Processes claim **exclusive** control of the resources they require
- **Hold-and-wait (a.k.a. wait-for) condition**
  - Processes hold resources already allocated to them while waiting for additional resources
- **No preemption condition**
  - Resources cannot be removed from the processes holding them until used to completion
- **Circular wait condition**
  - A **circular chain** of processes exists in which each process holds one or more resources that are requested by the next process in the chain

Dining Philosophers had it all

- Mutual exclusion
    - Exclusive use of chopsticks
- Hold and wait condition
    - Hold 1 chopstick, wait for next
- No preemption condition
    - Cannot force another to undo their hold
- Circular wait condition
    - Each waits for next neighbor to put down chopstick

Mutual Exclusion

- Processes claim **exclusive** control of the resources they require

- How to break it?
  - Grant non-exclusive access only (e.g., read-only)

Hold and Wait Condition

- Processes hold resources already allocated to them while waiting for additional resources
- How to break it?
  - Allow processes to either access all its required resources at once, or none of them

No Preemption Condition

- Resources cannot be removed from the processes holding them until used to completion
- How to break it?
  - Allow processes to be pre-empted and forced to abort themselves or release held resources

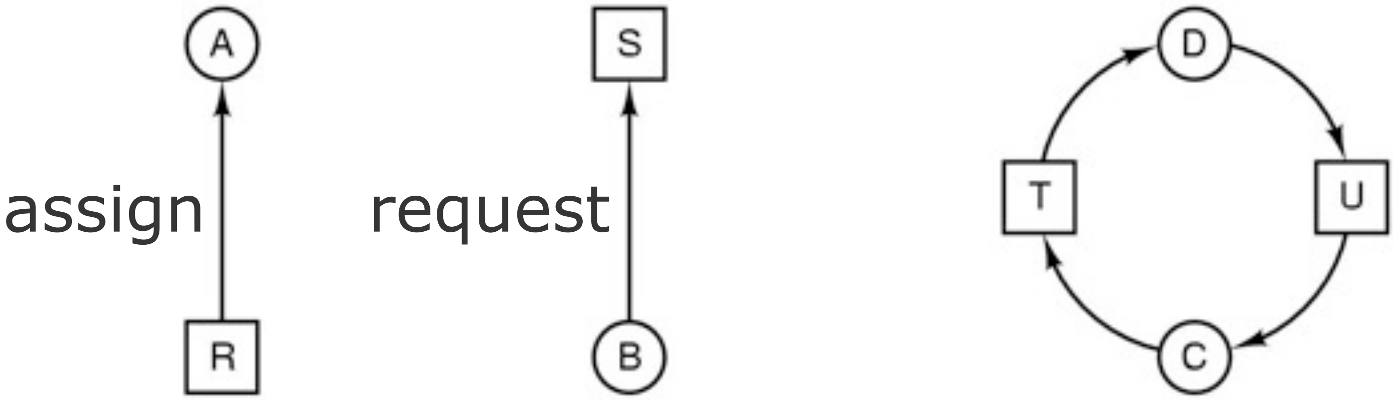
Circular Wait Condition

- A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain
- How to break it?
    - Allow processes to access resources only in increasing order of resource id

Resource Allocation Graph

- **Nodes**
  - Processes
  - Resources
- **Arcs**
  - From resource to process = **resource assigned to process**
  - From process to resource = **process requests (and is waiting for) resource**

Resource Allocation Graph



(a) (b) (c)

(a) resource R assigned to process A

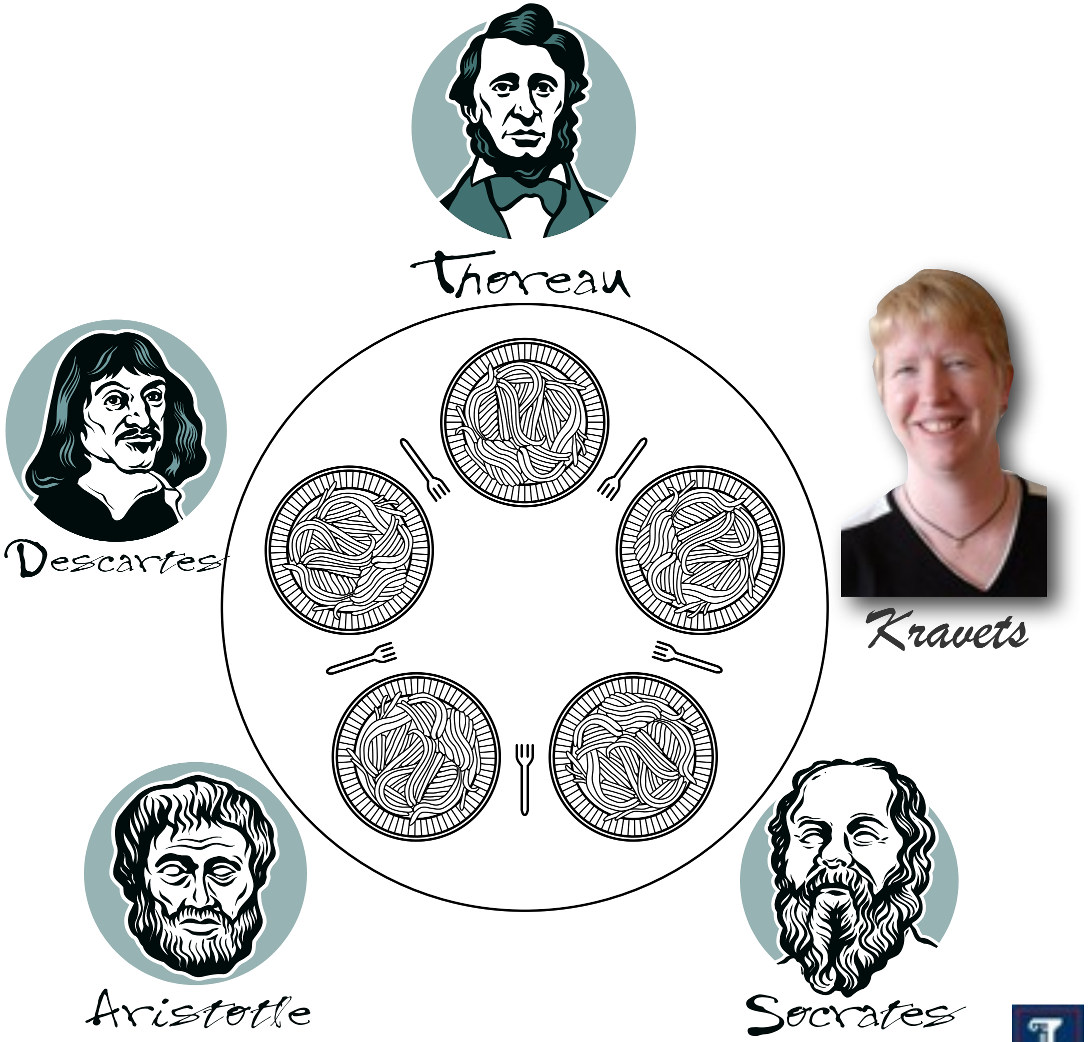
(b) process B is requesting/waiting for resource S

(c) process C and D are in deadlock over resources T and U

Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

```
void philosopher(i) {
 while true {
   take left fork;
   take right fork;
   eat();
   put left fork;
   put right fork;
 }
}
```

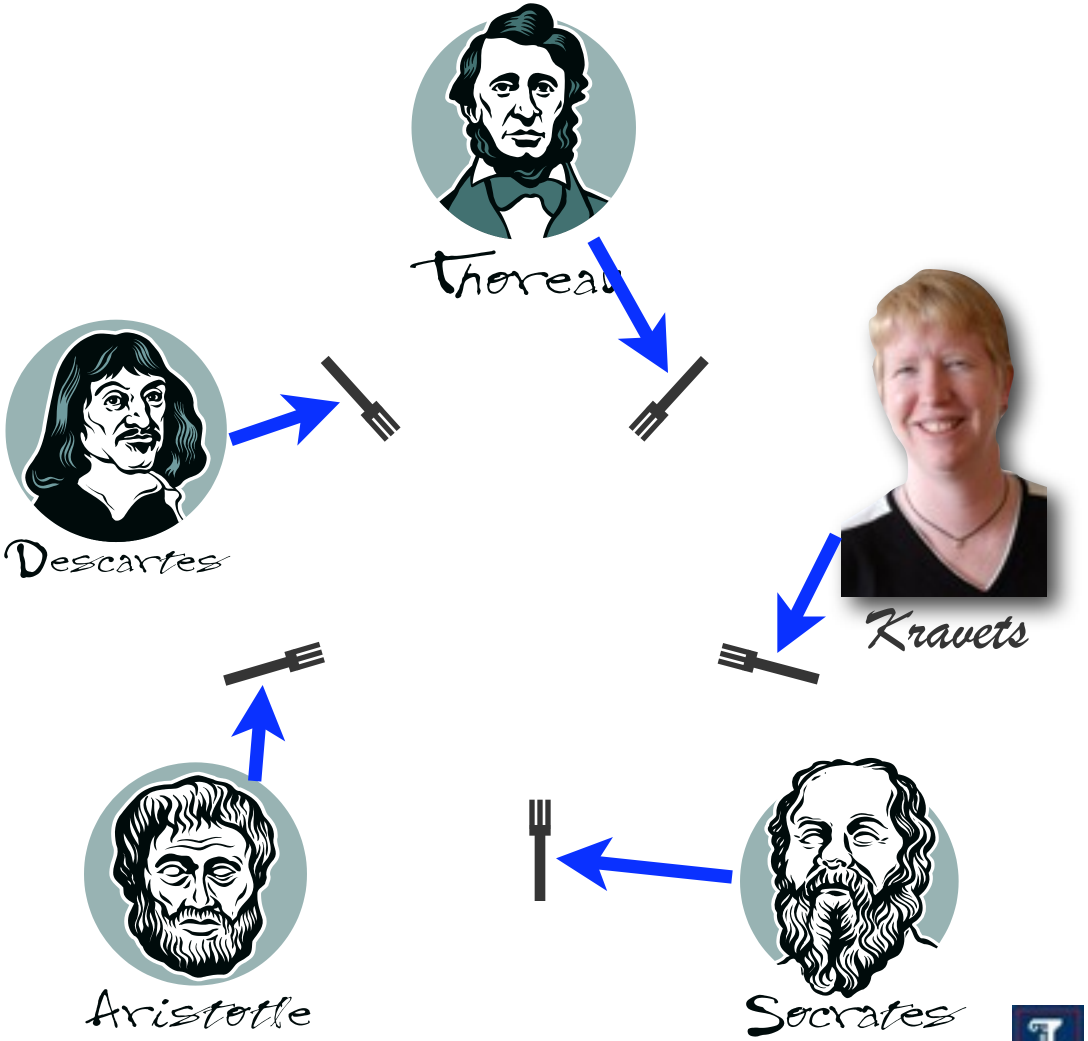


Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

One node per philosopher and per fork

1. Everyone tries to pick up left fork (request edges)

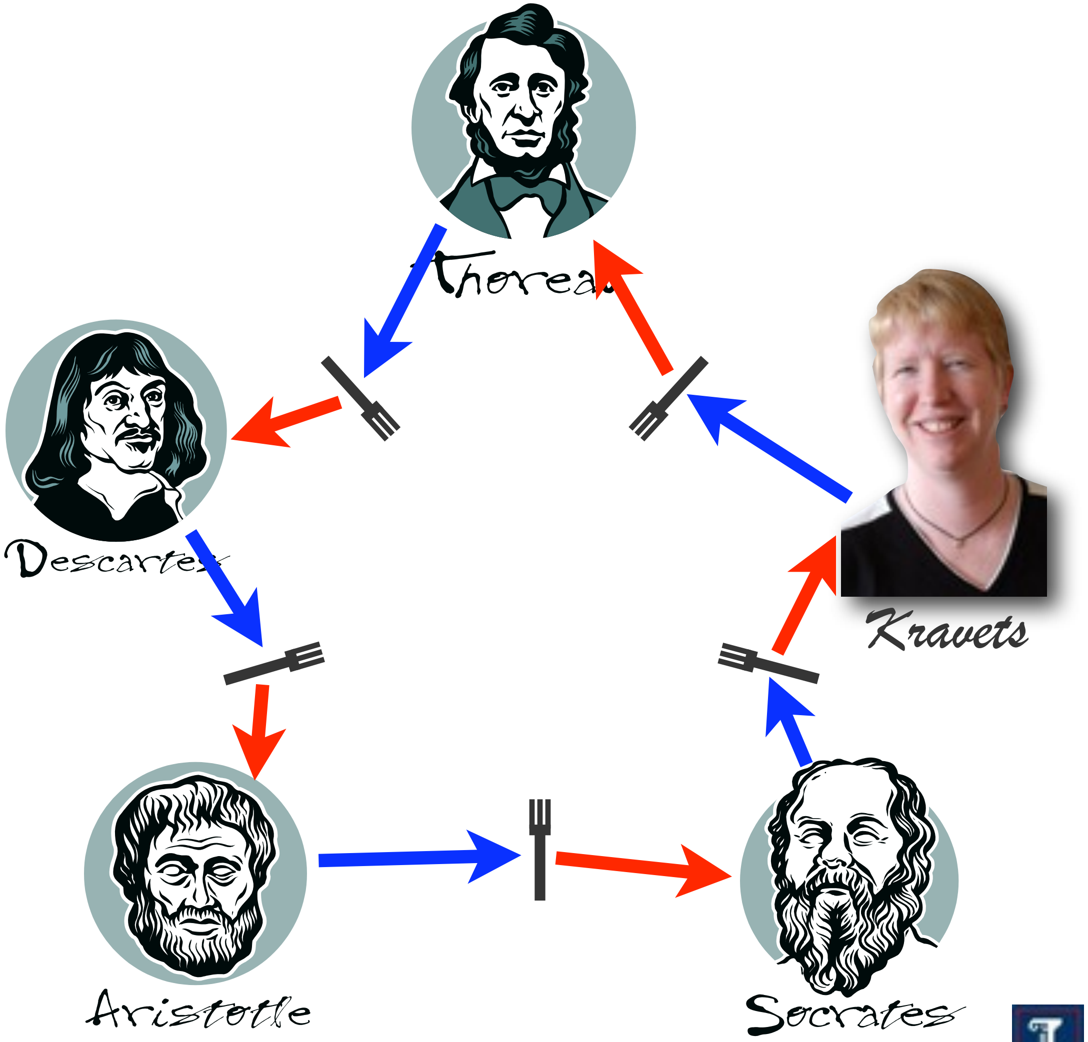


Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

One node per philosopher and per fork

1. Everyone tries to pick up left fork (request edges)
2. Everyone succeeds! (request edges become assignment edges)
3. Everyone tries to pick up right fork (request edges)
4. Cycle => deadlock



Summary

- Definition of deadlock
- 4 conditions for deadlock to happen
  - How to tell when circular wait condition happens: cycle in resource allocation graph
- Next time: How to deal with deadlock