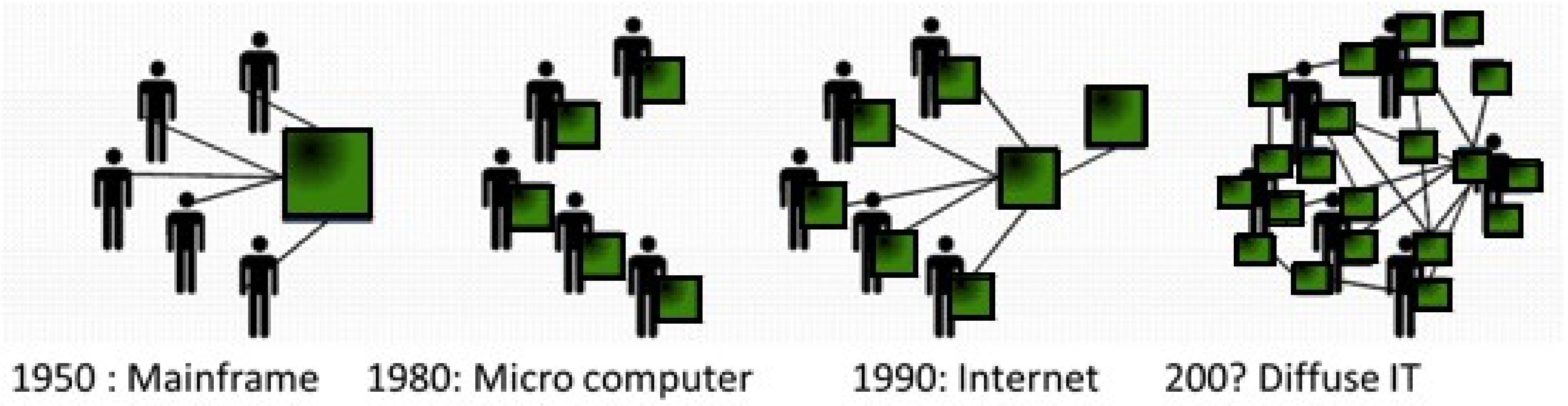
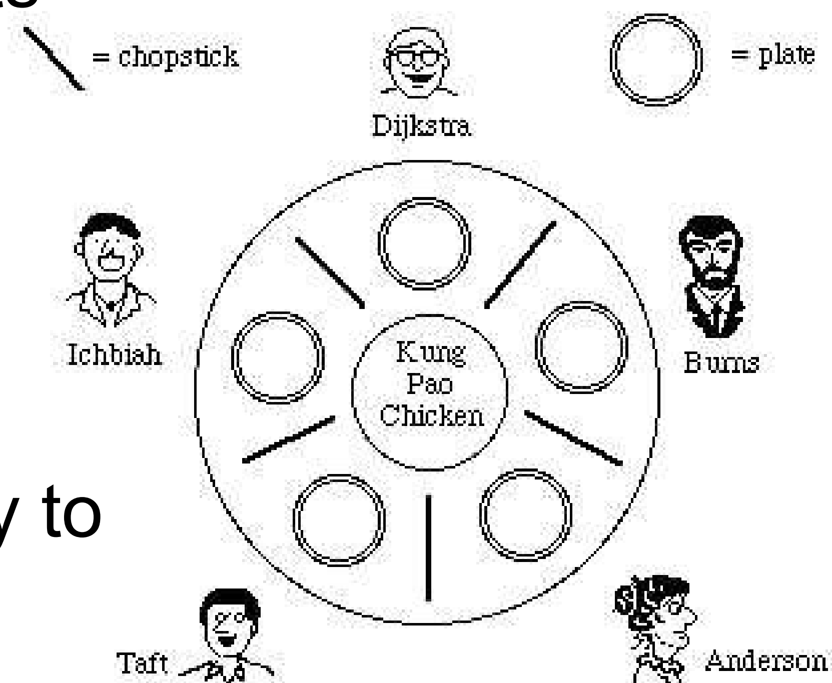
# Disciplina
# Sistemas de Computação

Aula 14



# Dining Philosophers

- It's lunch time in the philosophy dept
- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
- Eating => need two chopsticks, try to pick up two closest chopsticks
  - Block if neighbor has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking

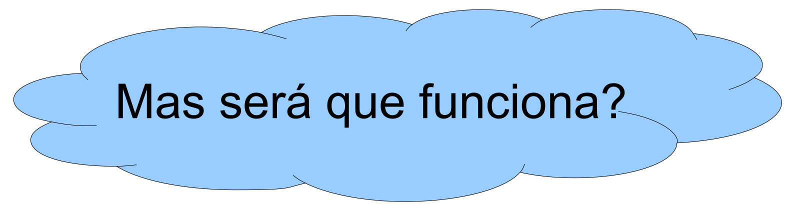


# Dining Philosophers v1

```
Semaphore  chopstick[5];

do{
   wait(chopstick[i]);  // left chopstick
   wait(chopstick[(i+1)%5 ]); // right chopstick
      // eat
   signal(chopstick[i]);  // left chopstick
   signal(chopstick[(i+1)%5 ]); // right chopstick
      // think
   } while(TRUE);
```

# Dining Philosophers v1




```
Semaphore  chopstick[5];

do{
   wait(chopstick[i]);  // left chopstick
   wait(chopstick[(i+1)%5 ]); // right chopstick
      // eat
   signal(chopstick[i]);  // left chopstick
   signal(chopstick[(i+1)%5 ]); // right chopstick
      // think
   } while(TRUE);
```

# Dining Philosophers v2

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N−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 */
     }
}
```

# Dining Philosophers v2

```
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]);
      }
}
```

# Real-world Examples

- Producer-consumer
  - Audio-Video player: network and display threads; shared buffer
  - Web servers: master thread and slave thread
- Dining Philosophers
  - Cooperating processes that need to share limited resources
    - Set of processes that need to lock multiple resources
      - Disk and tape (backup),
    - Travel reservation: hotel, airline, car rental databases

# Deadlocks

- What are deadlocks?
- Conditions for deadlocks
- Deadlock prevention
- Deadlock detection

# Real-world Examples

- Producer-consumer
  - Audio-Video player: network and display threads; shared buffer
  - Web servers: master thread and slave thread
- Dining Philosophers
  - Cooperating processes that need to share limited resources
    - Set of processes that need to lock multiple resources
      - Disk and tape (backup),
    - Travel reservation: hotel, airline, car rental databases

# Deadlocks

- Deadlock: A condition where two or more threads are waiting for an event that can only be generated by these same threads.
- Example:

Process A:

```
printer.Wait();
disk.Wait();

// copy from disk
// to printer

printer.Signal();
disk.Signal();
```

Process B:

```
disk.Wait();
printer.Wait();

// copy from disk
// to printer

printer.Signal();
disk.Signal();
```

# Deadlocks: Terminology

- **Deadlock** can occur when several threads compete for a finite number of resources simultaneously
- **Deadlock prevention** algorithms check resource requests and possibly availability to prevent deadlock.
- **Deadlock detection** finds instances of deadlock when threads stop making progress and tries to recover.
- **Starvation** occurs when a thread waits indefinitely for some resource, but other threads are actually using it (making progress).

  => Starvation is a different condition from deadlock

# Necessary Conditions for Deadlock

- **Deadlock can happen if all the following conditions hold.**
  - **Mutual Exclusion:** at least one thread must hold a resource in non-sharable mode, i.e., the resource may only be used by one thread at a time.
  - **Hold and Wait:** at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
  - **No Preemption:** A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
  - **Circular wait:** A set of waiting threads $\{t_1, ..., t_n\}$ where $t_i$ is waiting on $t_{i+1}$ ($i$ = 1 to $n$) and $t_n$ is waiting on $t_1$.

# Deadlock Detection Using a Resource Allocation Graph

We define a graph with vertices that represent both resources $\{r_1, ..., r_m\}$ and threads $\{t_1, ..., t_n\}$.

- A directed edge from a thread to a resource, $t_i \rightarrow r_j$ indicates that $t_i$ has requested that resource, but has not yet acquired it (*Request Edge*)
- A directed edge from a resource to a thread $r_j \rightarrow t_i$ indicates that the OS has allocated $r_j$ to $t_i$ (*Assignment Edge*)

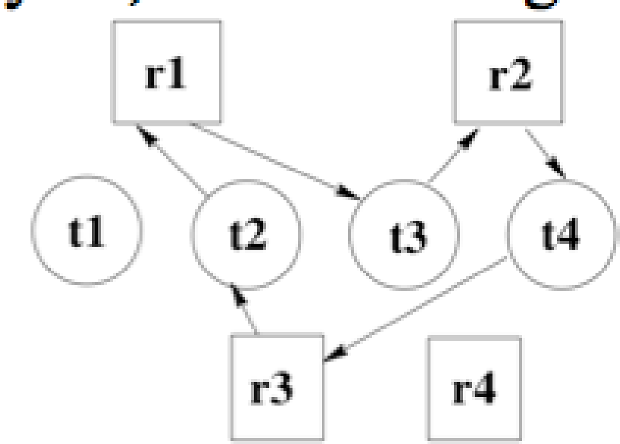
# Deadlock Detection Using a Resource Allocation Graph

We define a graph with vertices that represent both resources $\{r_1, ..., r_m\}$ and threads $\{t_1, ..., t_n\}$.

- A directed edge from a thread to a resource, $t_i \rightarrow r_j$ indicates that $t_i$ has requested that resource, but has not yet acquired it (*Request Edge*)
- A directed edge from a resource to a thread $r_j \rightarrow t_i$ indicates that the OS has allocated $r_j$ to $t_i$ (*Assignment Edge*)

If the graph has no cycles, no deadlock exists.

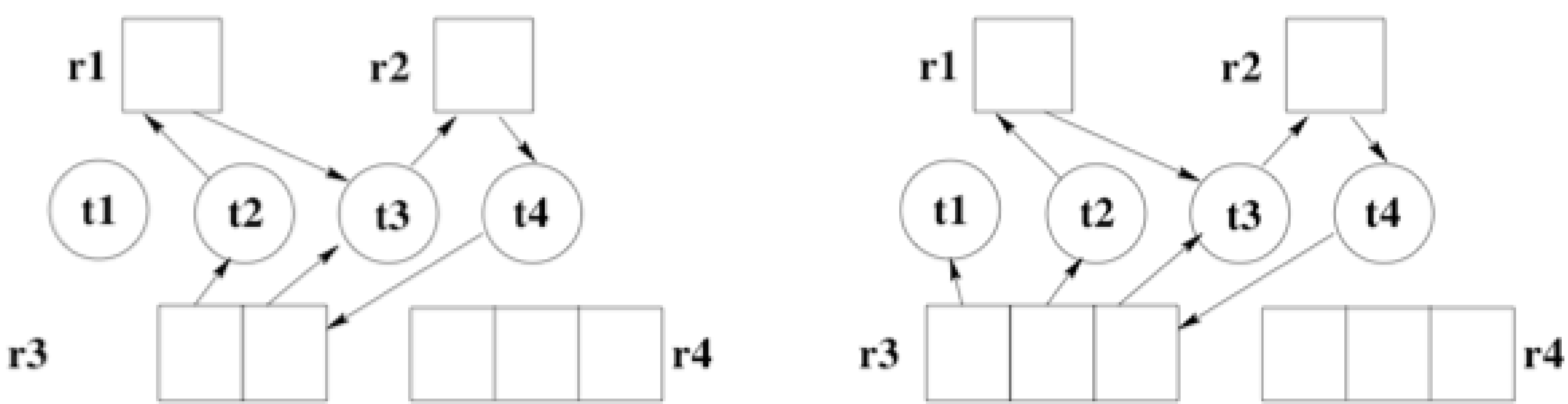
If the graph has a cycle, deadlock might exist.



# Deadlock Detection Using a Resource Allocation Graph

What if there are multiple interchangeable instances of a resource?

- Then a cycle indicates only that deadlock *might* exist.
- If any instance of a resource involved in the cycle is held by a thread not in the cycle, then we can make progress when that resource is released.



# Deadlocks Prevention

**Prevent deadlock:** ensure that at least one of the necessary conditions doesn't hold.

- Mutual Exclusion: make resources sharable (but not all resources can be shared)
- Hold and Wait:
  - Guarantee that a thread cannot hold one resource when it requests another
  - Make threads request all the resources they need at once and make the thread release all resources before requesting a new set.
- No Preemption:
  - If a thread requests a resource that cannot be immediately allocated to it, then the OS preempts (releases) all the resources that the thread is currently holding.
  - Only when all of the resources are available, will the OS restart the thread.
  - Problem: not all resources can be easily preempted.
- Circular wait: impose an ordering (numbering) on the resources and request them in order.