CSc 360
Operating Systems
Deadlocks

Wenjun Yang

Fall 2025

Review

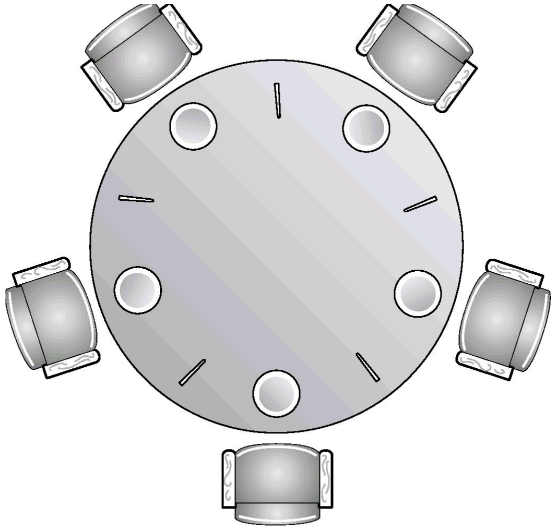
- Ways to process synchronization
  - hardware-assisted solutions
  - mutex, semaphores
  - monitors
- Required properties
  - mutual exclusion
  - making progress (i.e., no deadlock)
  - bounded waiting (i.e., no livelock)

Dining philosophers: semaphores

- Shared data
  - Initially all values are 1

    semaphore chopstick[5];
- Using semaphores, for Philosopher *i*:

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



* what’s the problem? how to implement semaphores with or without busy waiting?

Dining philosophers: monitors

```
monitor DP {
    …
    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
                self[i].signal () ; // no effect if not blocked
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

- Using monitors

```
dp.pickup (i)
    …
  EAT
    …
dp.putdown (i)
```

* what's the problem? how to implement monitors with or without condition variables?

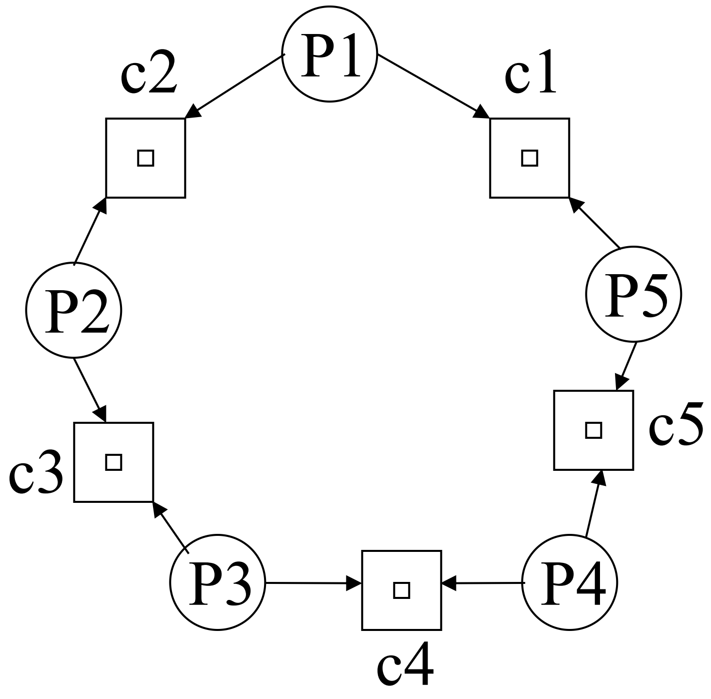
Deadlocks

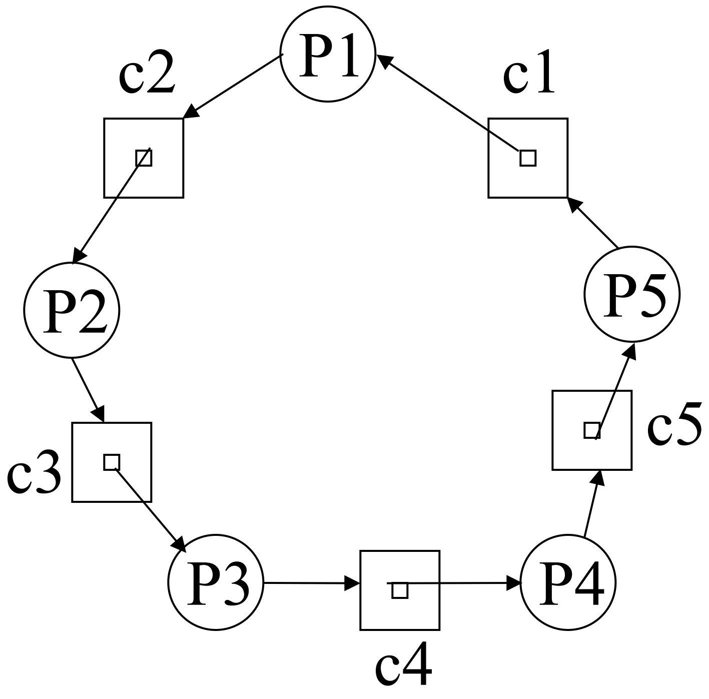
- Deadlock *can* occur if **all** are true
  - mutual exclusion
    - wait(chopstick[i]);
  - hold-and-wait
    - **wait(chopstick[i]);** wait(chopstick[(i+1)%5]);
  - no-preemption
    - wait();
  - circular-wait
    - chopstick[(i+1)%5]

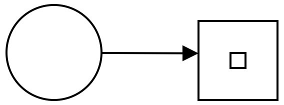
* do you have deadlocks in your p2 design?

Q: necessary conditions

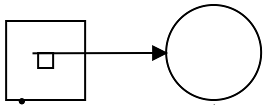
Resource-allocation graph





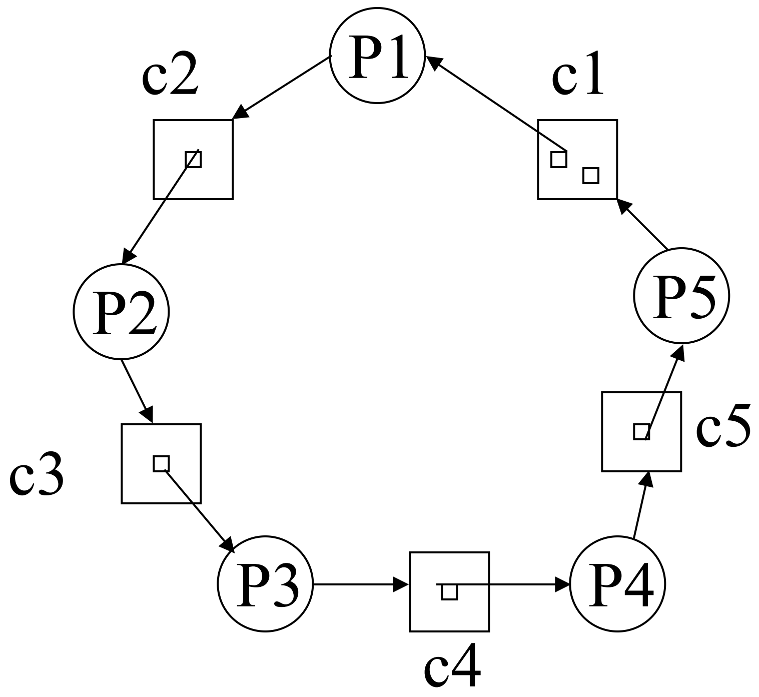


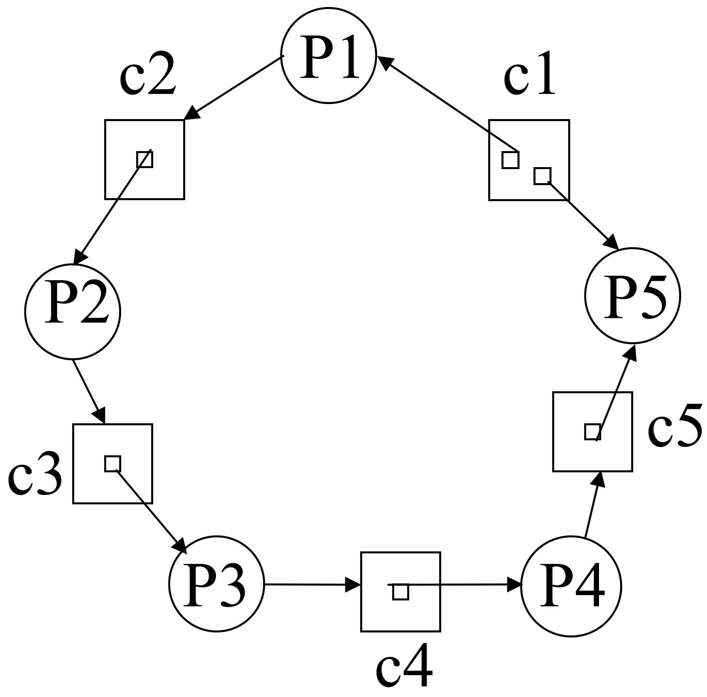
request edge



assignment edge

How about this?





- Directed circle
  - one instance per resource type
    - deadlock
  - otherwise: *maybe*!

- No directed circle
  - no deadlock

Preventing deadlocks

- Prevention
  - mutual exclusion
    - only when mutual exclusion is really necessary
  - hold-and-wait
    - all-or-none
  - non-preemption
    - give up on request
  - circular-wait
    - strictly ordered

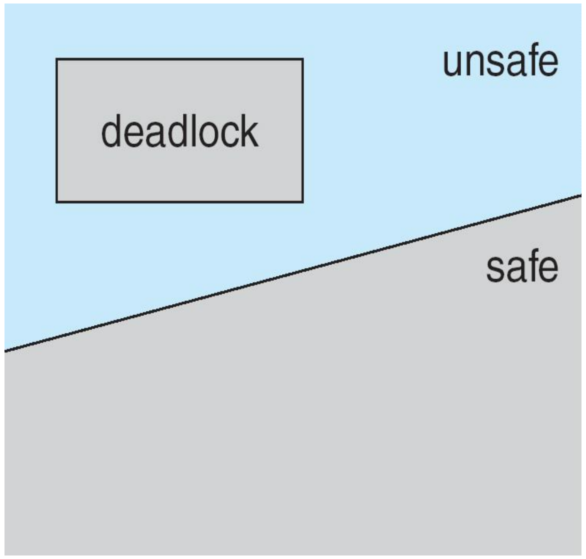
```
void test (int i) {
            if ( (state[(i + 4) % 5] !=
EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) )
{
                        state[i] = EATING ;
                        self[i].signal () ;
            }
} void pickup (int i) {
              state[i] =
  HUNGRY;
              test(i);
              if (state[i] !=
  EATING)
                          self
  [i].wait;
```

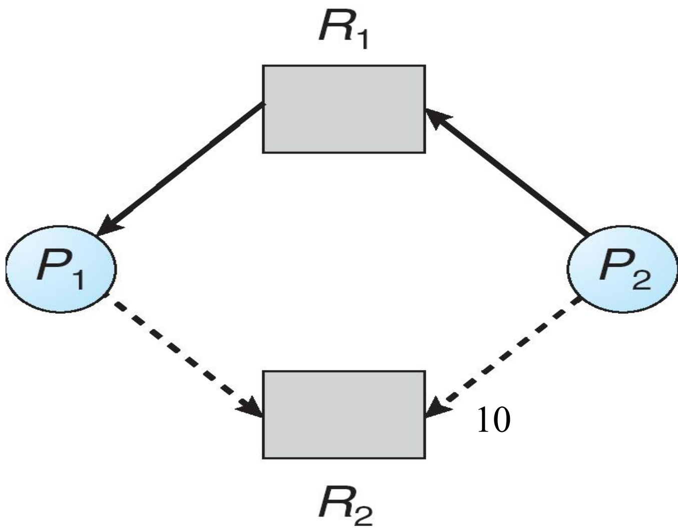
Avoiding deadlocks

- Avoidance
  - declare maximal resource usage in advance
    - claim edge
  - check against currently admitted processes
  - admit if safe (e.g., no circular-wait)
    - a sequence of P_i, such as P_i is satisfied with all $P_{j<i}$
    - single instance resource: resource-allocation graph
    - multi-instance resource: banker's algorithm

Deadlock avoidance

- Basic fact
  - in safe state: no deadlocks
  - in unsafe state:
    - possible deadlocks
  - avoidance: not in unsafe state
- Single instance of resource
  - resource-allocation graph
  - claim vs request edge
  - assignment edge





Banker's algorithm

Let n = number of processes, and m = number of resources types.

- ***Available****:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- ***Max****:* n x m matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- ***Allocation****:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of $R_{j.}$
- ***Need****:* n x m matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

  $$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Safety algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize:

   $Work = Available$

   $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.

2. Find an i such that both:

   (a) $Finish[i] = false$

   (b) $Need_i <= Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i]$ == true for all i, then the system is in a safe state.

Resource-request algorithm

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i <= Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i <= Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe: the resources are allocated to Pi.*
- *If unsafe: Pi must wait, and the old resource-allocation state is restored*

example on board or after class

An example

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- A pending request (1, 0, 2) from P1
  - Is the request legitimate?
  - Is it safe after the grant?
    - Yes: Grant
    - No: Reject

This lecture

- Deadlocks
  - deadlock characteristics
  - how to prevent deadlocks
  - how to avoid deadlocks
  - how to detect and resolve deadlocks
    - after-class reading
- Explore further
  - CSC 464: Concurrency

Next few lectures

- File system design and implementation
  - as a preparation for P3
- Memory management
  - Main memory
  - Virtual memory
- I/O management