CSC369 - Tutorial 10

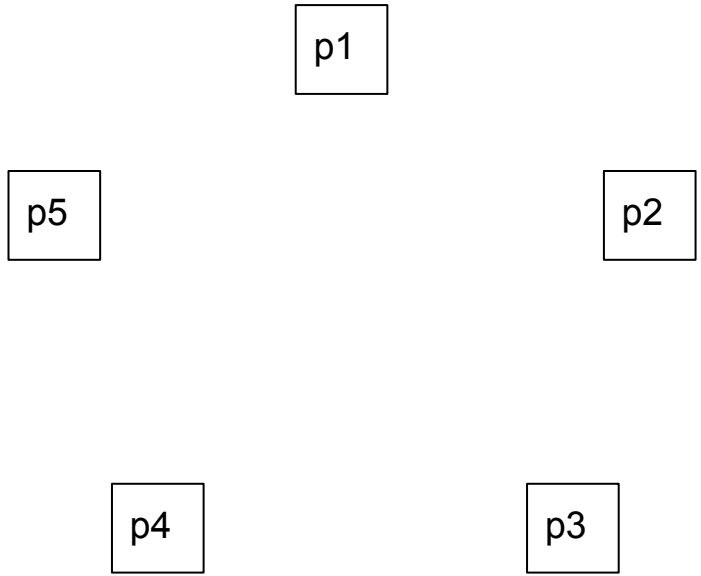
The dining philosophers problem
(handout due the end of the tutorial)
Please **don't read question 3 yet**, it will spoil your chance to come up with a solution on your own :)

Felipe de Azevedo Piovezan, Fall 2017, University of Toronto

The dining philosophers problem

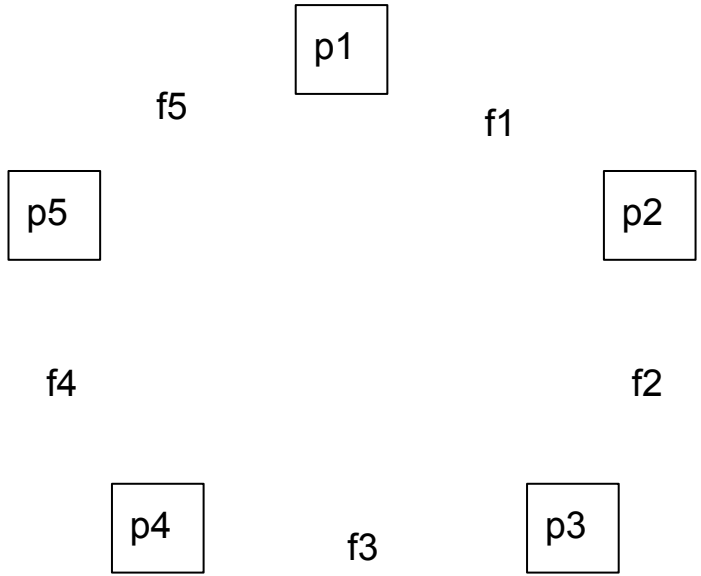
The dining philosophers problem

- 5 philosophers



The dining philosophers problem

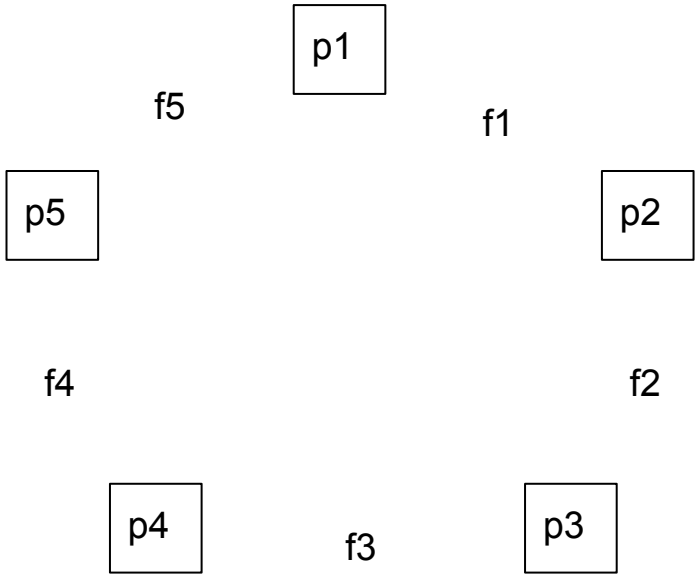
- 5 philosophers
- 5 forks between them



The dining philosophers problem

- 5 philosophers
- 5 forks between them
- They all want to eat:

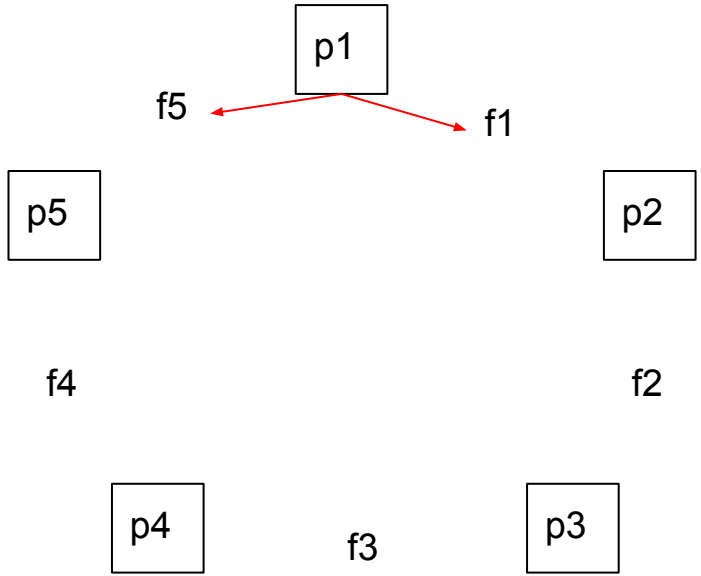
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- 5 philosophers
- 5 forks between them
- They all want to eat:

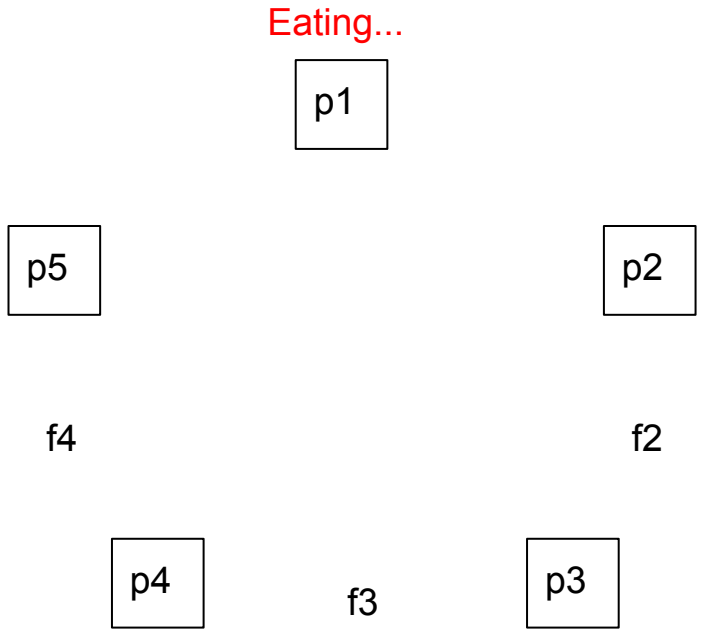
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- 5 philosophers
- 5 forks between them
- They all want to eat:

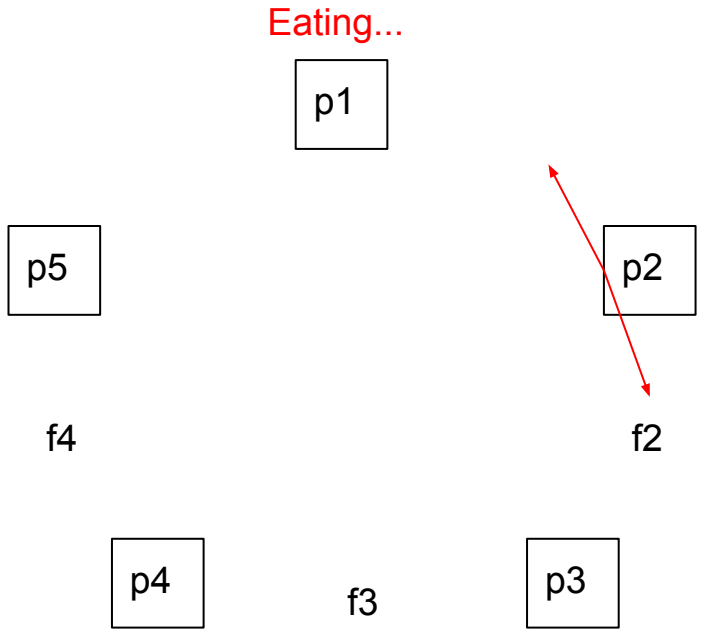
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- 5 philosophers
- 5 forks between them
- They all want to eat:

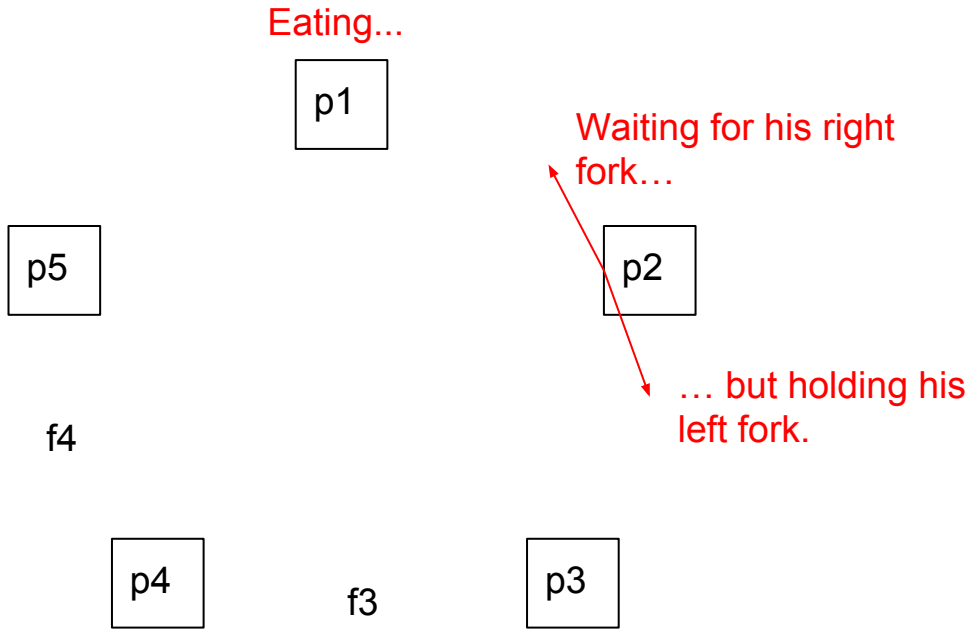
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

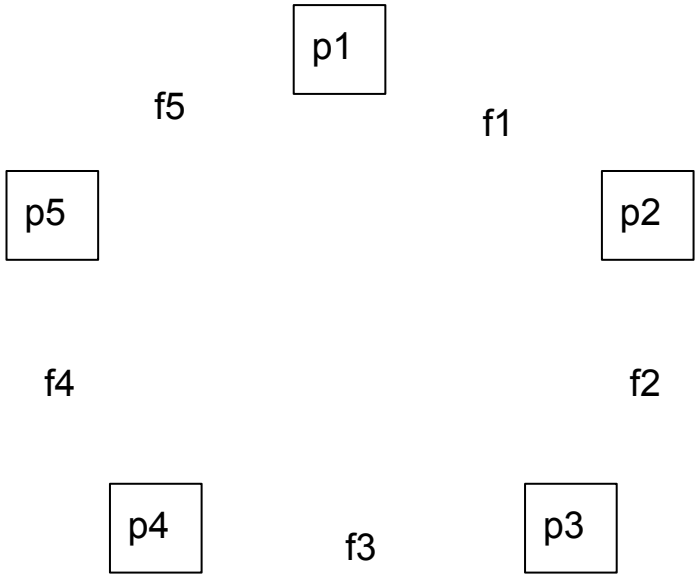
- 5 philosophers
- 5 forks between them
- They all want to eat:

```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- Are there any problems with this algorithm?



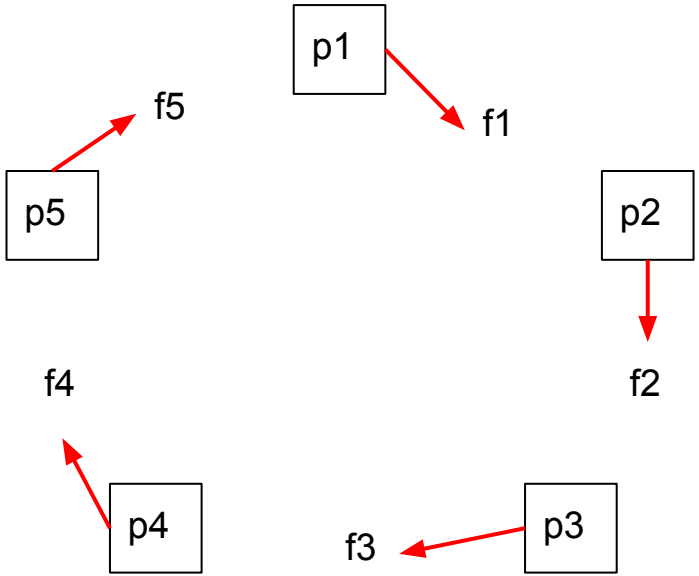

```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```

The dining philosophers problem

- Are there any problems with this algorithm?

Everyone to the left!

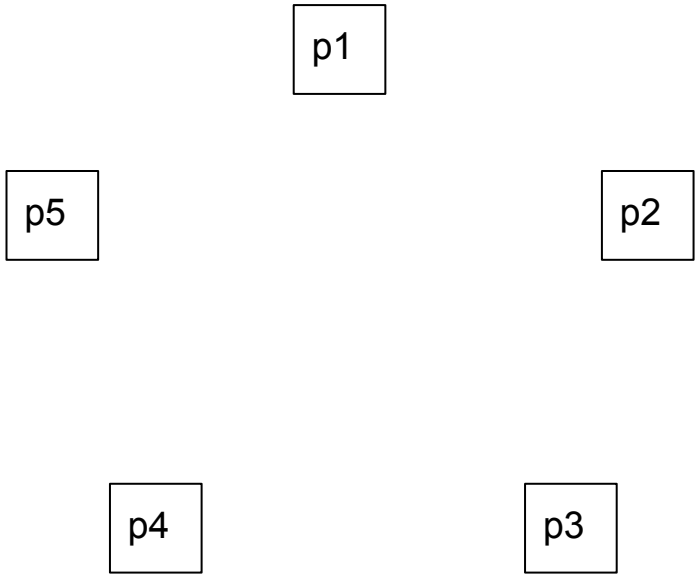
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- Are there any problems with this algorithm?

Everyone to the left!

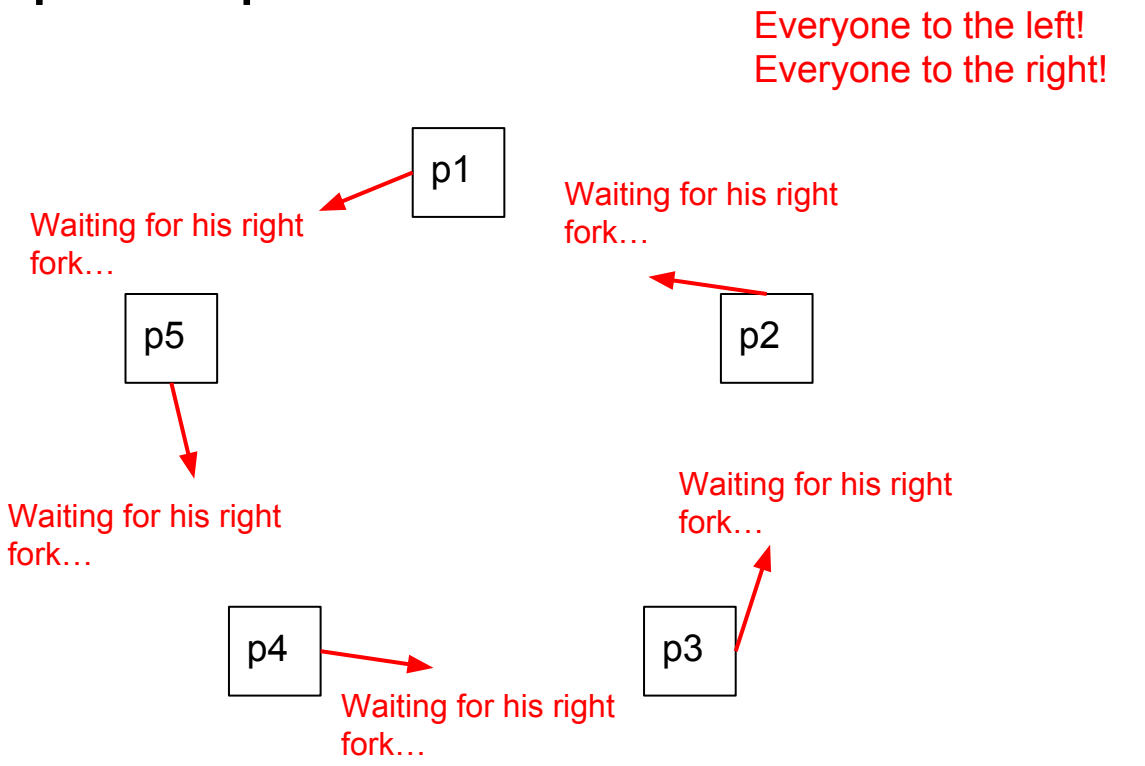



```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```

The dining philosophers problem

- Are there any problems with this algorithm?

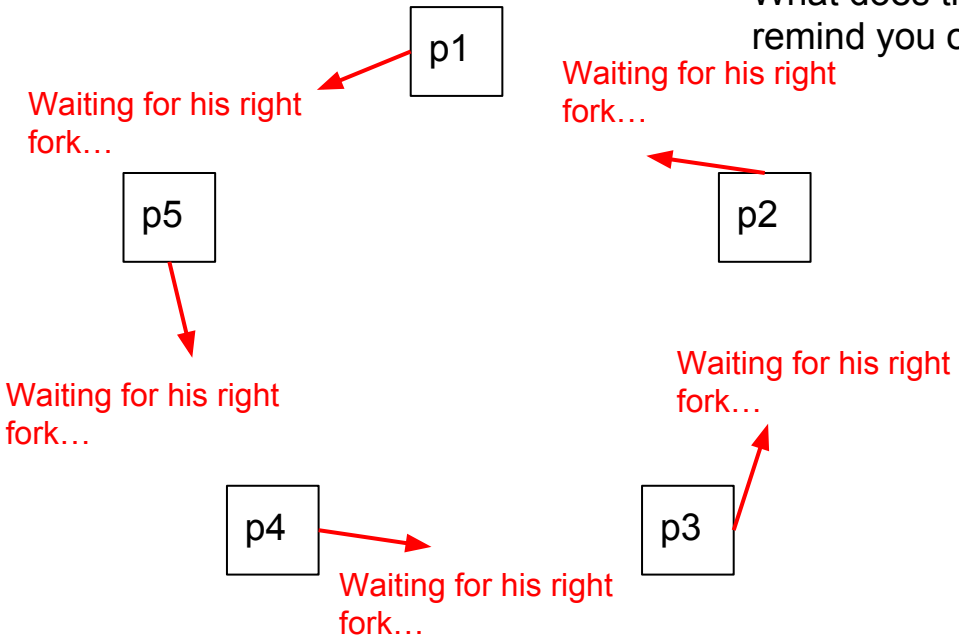
```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



The dining philosophers problem

- Are there any problems with this algorithm?

```
// p is the philosopher's ID
void act(int p) {
      while (1) {
          think();
          get_fork(left(p));
          get_fork(right(p));
          eat();
          put_fork(left(p));
          put_fork(right(p));
      }
}
```



Deadlock conditions:

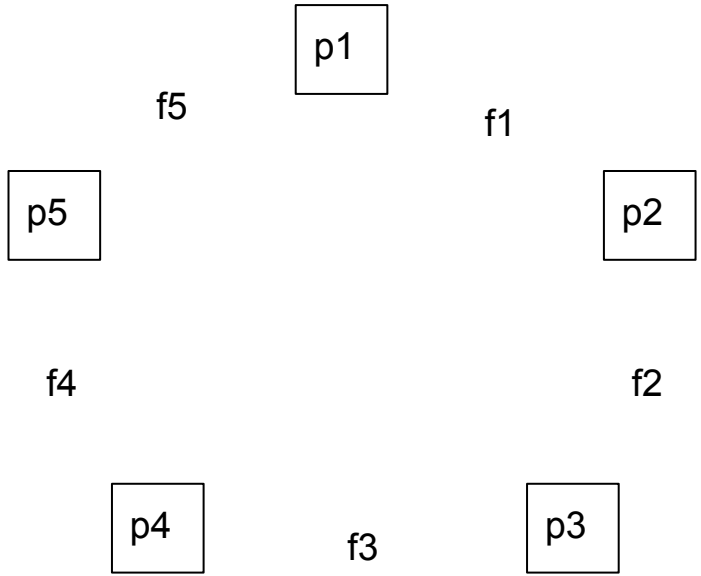
- There are four **necessary and sufficient** conditions for deadlock to occur.
- What do we mean by necessary and sufficient?
- Think of “if and only if”:
  - If your system can deadlock, those conditions are ALL present.
    - The contrapositive might be more useful:
      if only a proper subset of them is present (that is, 1 or 2 or 3 but not all 4 conditions), then your system CANNOT deadlock.
  - If ALL of them are present, then your system can deadlock.

Deadlock conditions:

- Mutual Exclusion
  - Only one process may use a resource at a time
- Hold and wait
  - A process may hold allocated resources while awaiting assignment of others
- No preemption
  - A resource cannot be forcibly removed from a process holding it.
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

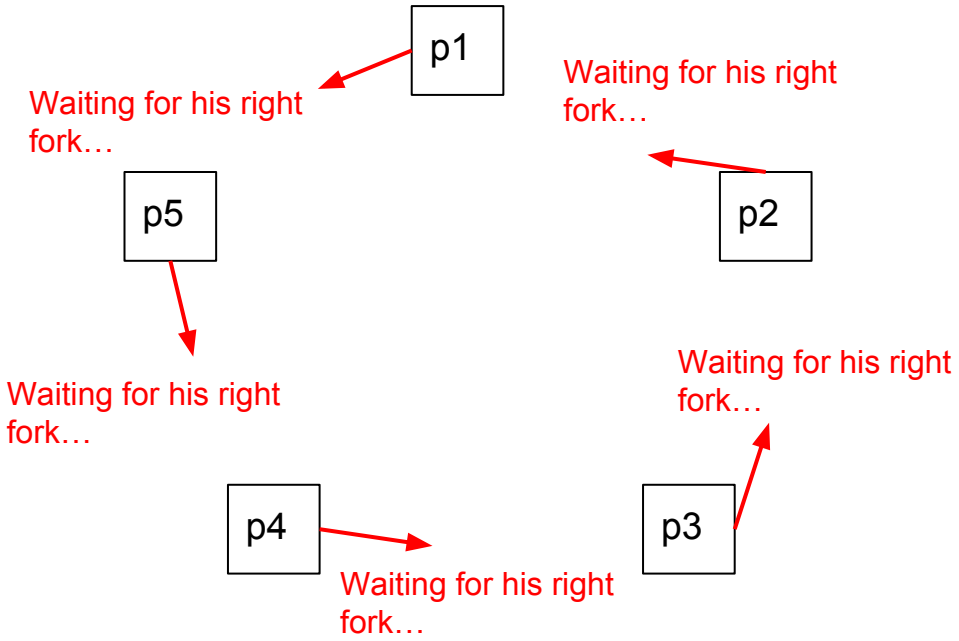
Where are the deadlock conditions in the dining philosophers problem?

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



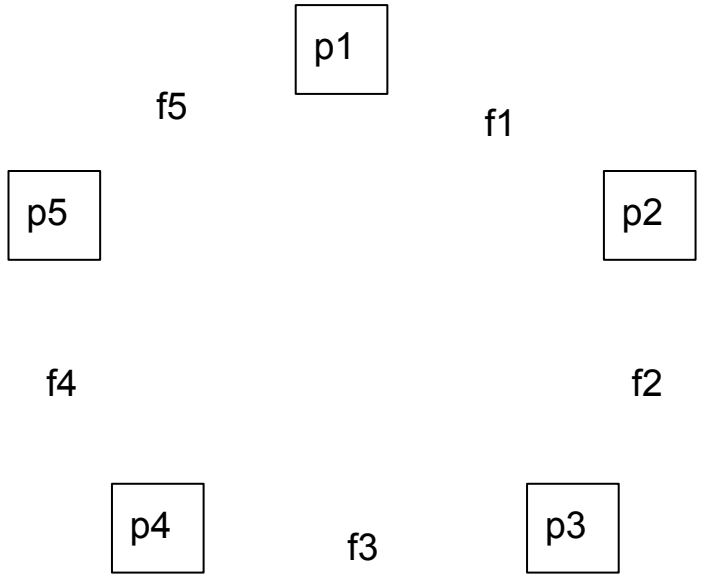
In the lecture slides...

- “Circular wait implies hold and wait”
- Can you see why now?
- To get a:
  - “closed chain of philosophers”, such that “each holds one fork needed by the next” (which is circular wait),
- They need to be allowed to:
  - hold a fork while trying to get the other (which is hold and wait).



How to prevent deadlock here?

- Think about A2.
- Think about a different algorithm for acquiring/releasing locks.
- Maybe something else entirely?
- In all these cases, which deadlock condition(s) are you preventing?



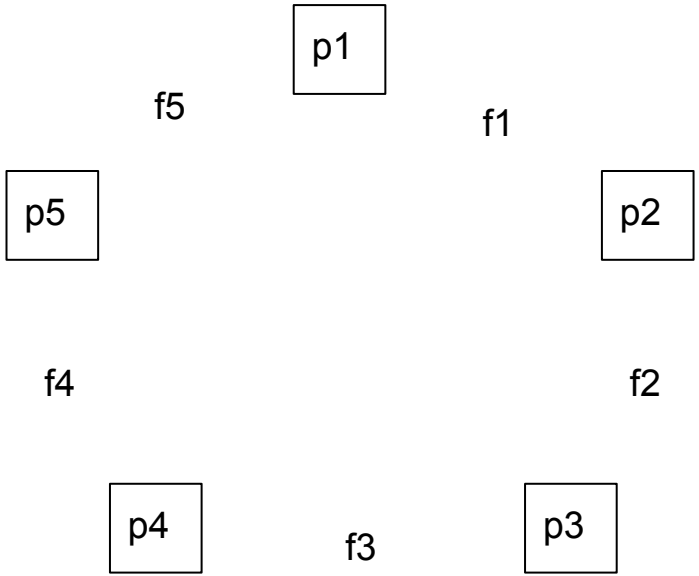
How to prevent deadlock here?

- Now look at question 3:

Consider that some philosophers always pick up their left forks first (name these philosophers “lefties”), while other philosophers always pick up their right forks first (we’ll call them “righties”).

Let’s also consider that there is at least one “lefty” and at least one “righty” at the table.

Can deadlock occur?
Is starvation possible (you can assume that we have a fair scheduling policy)?



Banker's algorithm

4) Deadlock avoidance and the Banker's algorithm.
Suppose that we have a system with 8 pages of memory and 3 processes: A, B, and C, which need 4, 5, and 5 pages to complete respectively (Assume no eviction).

If they take turns requesting one page each, and the system grants requests in order, the system will deadlock.

A (needs 4)	1	1	1	2	2	2	3	3	3	W	W
B (needs 5)	0	1	1	1	2	2	2	3	3	3	W
C (needs 5)	0	0	1	1	1	2	2	2	W	W	W
Total allocated	1	2	3	4	5	6	7	8	8	8	8

First, explain to your partner(s) why this is a deadlock.

Which allocation is the last safe state? In other words, which is the last state where there is still some allocation that would allow the processes to complete. Remember that once a process has acquired all of its required resources (pages in this case), it will eventually release them all.