CS330: Operating Systems

Concurrency bugs

Common issues in concurrent programs

- Atomicity issues
- Failure of ordering assumption
- Deadlocks

Concurrency bugs - atomicity issues

```
char *ptr;   // Allocated before use
void T1( )
{
    ...
    strcpy(ptr, "hello world!");
   ...
}
```

```
void T2( )
{
    ...
     if(some_condition){
          free(ptr); ptr = NULL;
    }
    ...
}
```

- This code is buggy. What is the issue?
- T2 can free the pointer before T1 uses it.
- How to fix it?

Concurrency bugs - atomicity issues

```
char *ptr;   // Allocated before use
void T1( )
{
    ...
    if(ptr)  strcpy(ptr, "hello world!");
   ...
}
```

```
void T2( )
{
    ...
     if(some_condition){
          free(ptr); ptr = NULL;
    }
    ...
}
```

- Does the above fix (checking ptr in T1) work?
- Not really. Consider the following order of execution:
- T1: "if(ptr)" T2: "free(ptr)" T1: "strcpy" Result: Segfault

Concurrency bugs - ordering issues

```
1. bool pending;
2. void T1( )
3. {
4.     pending = true;
5.     do_large_processing( );
6.     while (pending);
7. }
```

```
1. void T2( )
2. {
3.     do_some_processing( );
4.     pending = false;
5.     some_other_processing( );
6. }
```

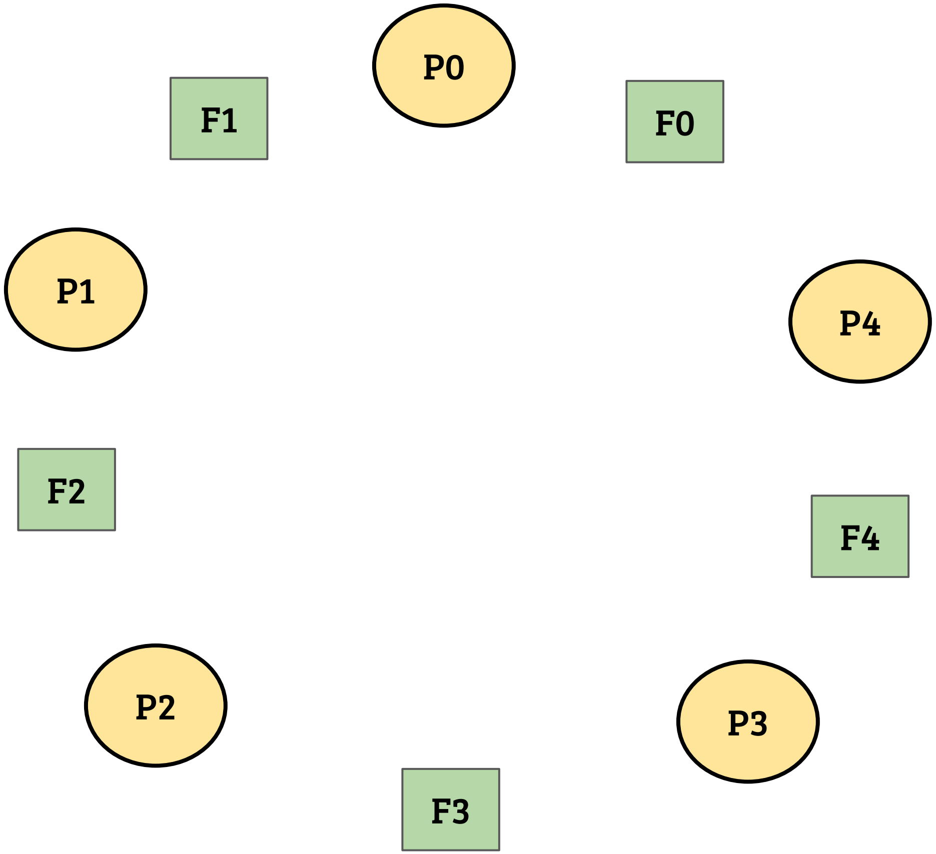
- This code works with the assumption that line#4 of T2 is executed after line#4 of T1
- If this ordering is violated, T1 is stuck in the while loop

Concurrency bugs - deadlocks

```
struct acc_t{
      lock_t *L;
      id_t  acc_no;
      long balance;
}
void txn_transfer( acc_t *src,
                   acc_t *dst,  long amount)
{
   lock(src → L);  lock(dst → L);
   check_and_transfer(src, dst, amount);
   unlock(dst → L); unlock(src → L);
}
```

- Consider a simple transfer transaction in a bank
- Where is the deadlock?
- T1: txn_transfer(iitk, cse, 10000)
    - lock (iitk), lock (cse)
- T2: txn_transfer(cse, iitk, 5000)
    - lock (cse), lock(iitk)

Dining philosophers




```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      think( );
      acquire(forks[id]);
      acquire(forks[(id+1) % 5]);
      eat( );
      release( forks[(id+1) % 5]);
      release(forks[id]);
   }
}
```

Conditions for deadlock

- Mutual exclusion: exclusive control of resources (e.g, thread holding lock)
- Hold-and-wait: hold one resource and wait for other
- No resource preemption: Resources can not be forcibly removed from threads holding them
- Circular wait: A cycle of threads requesting locks held by others. Specifically, a cycle in the directed graph G (V, E) where V is the set of processes and (v1, v2) $\in$ E if v1 is waiting for a lock held by v2

All of the above conditions should be satisfied for a deadlock to occur

Solutions for deadlocks

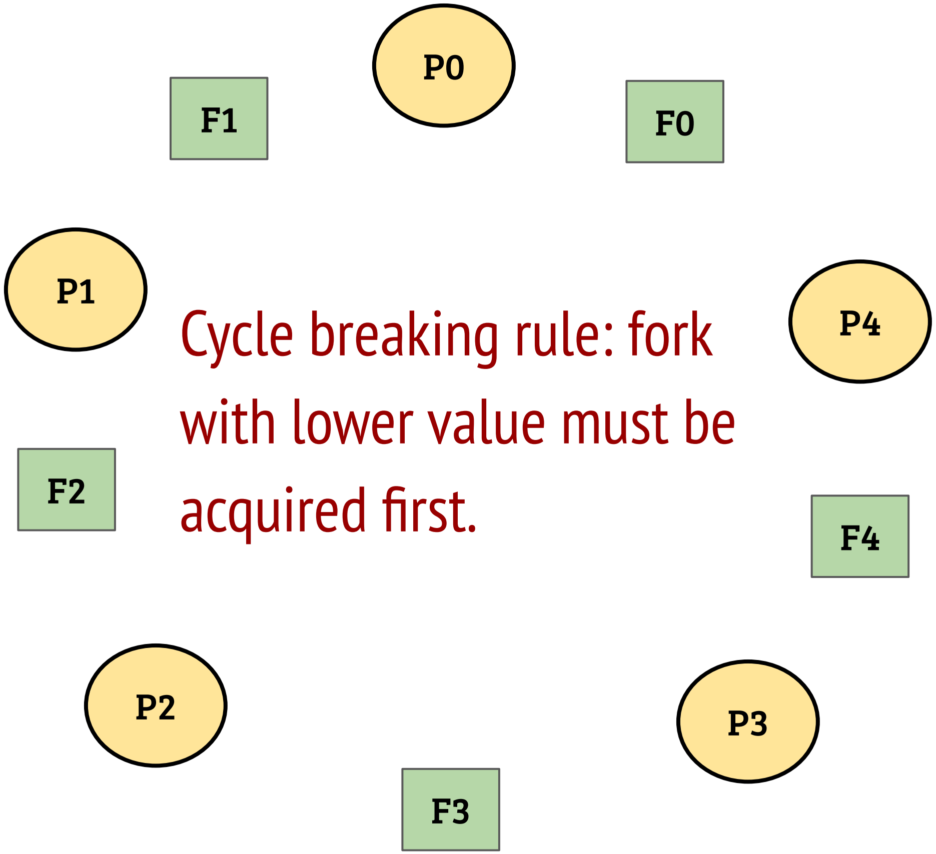
- Remove mutual exclusion: lock free data structures
- Either acquire all resources or no resource
    - trylock(lock) APIs can be used (e.g., pthread_mutex_trylock( ))
- Careful scheduling: Avoid scheduling threads such that no deadlock occur
- Most commonly used technique is to avoid circular wait. This can be achieved by ordering the resources and acquiring them in a particular order from all the threads.

Concurrency bugs - avoiding deadlocks

```
struct acc_t{
        lock_t *L;
        id_t  acc_no;
        long balance;
}
void txn_transfer( acc_t *src,
                   acc_t *dst,  long amount)
{
    lock(src → L);  lock(dst → L);
    check_and_transfer(src, amount);
    unlock(dst → L); unlock(src → L);
}
```

- Deadlock in a simple transfer transaction in a bank
- While acquiring locks, first acquire the lock for the account with lower “acc_no” value
- Account number comparison performed before acquiring the lock

Dining philosophers: breaking the deadlock



Cycle breaking rule: fork with lower value must be acquired first.

```
atomic_t forks[5];
Philosopher( int id)
{
   while (1) {
      if(id == 4){
         acquire(0);
         acquire(4);
      }else{
        acquire(forks[id]);
        acquire(forks[id+1]);
      }
      ...
}
```