

Lab 2	Locking I	1/13
<p>Critical Sections</p> <ol style="list-style-type: none">1. Pieces of code in which you want only one process running at a time2. These pieces usually access shared resources or data3. Before-or-after behavior<ul style="list-style-type: none">– Each transaction occurs either before or after each other one4. Example:<ul style="list-style-type: none">– Printer: if two print jobs enter a printer, want it to print all of one and then all of the other without interleaving pages		

Synchronization Objects**– Mutexes**

- These are basically the simplest and are available to use in lab 2 (spinlocks are mutexes)**

– Locks with types and semantics

- for example, read and write locks**

– Locks that unlock in order

- for example, wait queues**

Mutexes**– Two operations**

- `acquire(mutex r)`**
- `release(mutex r)`**

– acquire

- waits until the mutex is available, then locks it**
- any other acquiring processes continue to wait**

– release

- unlocks the mutex**

- The code base uses an `osp_spin_lock_t` for each mutex

```
osp_spin_lock(osl);  
// critical section: access shared items here  
osp_spin_unlock(osl);
```

- Can use one lock to protect multiple CSs
 - Every CS protected by a given lock is locked whenever any such CS is locked
- Anything that is shared, read from, and written to probably needs to be accessed in a CS
 - And protected by locks or other synchronization mechanisms

Counter Lock

- Counts the number of processes waiting to get a lock on a CS
- Has three methods:
 - `acquire(counterlock L)`
 - `release(counterlock L)`
 - `nwaiting(counterlock L)`

struct counterlock:

- `int _nwaiting`
 - counts the number of waiting processes
- `mutex wlock`
 - locks access to `_nwaiting`
- `mutex lockb`
 - locks access to the CS

nwaiting(counterlock L):

- return L._nwaiting

release(counterlock L):

- release L.lockb

acquire(counterlock L):

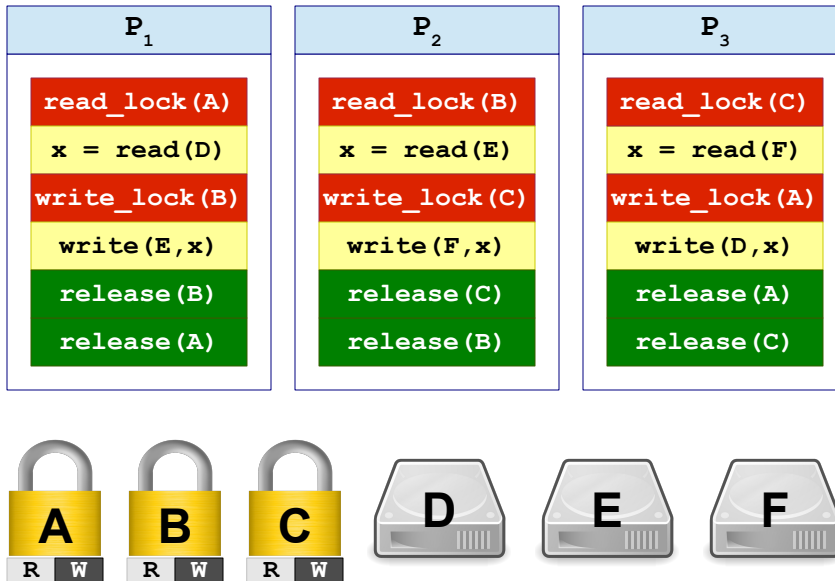
- acquire L.wlock
- L._nwaiting ++
- release L.wlock

– acquire L.lockb

- acquire L.wlock
- L._nwaiting --
- release L.wlock

**You may find it useful
to read the code notes
before continuing**

Deadlock Example



Sample Execution

- 1 $P_1.\text{read_lock}(A)$
(gets lock on A)
- 2 $P_2.\text{read_lock}(B)$
(gets lock on B)
- 3 $P_3.\text{read_lock}(C)$
(gets lock on C)
- 4 $P_1.\text{write_lock}(B)$
(gets ticket on B)
- 5 $P_2.\text{write_lock}(C)$
(gets ticket on C)
- 6 $P_3.\text{write_lock}(A)$
(gets ticket on A)

Dependency graph for deadlock

- Nodes = processes
- Edges = waits-on relation

For example,

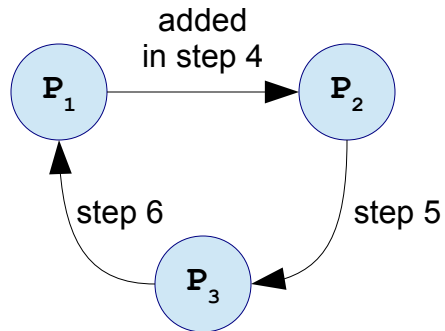


means that P_1 waits on P_2 .

This is a "wait graph" rather than a precedence graph

Convention:

- The waiter adds the dependency (in e.g. `acquire()`)
- The waitee frees the dependency (in e.g. `release()`)



Everything is fine until we add the edge $P_3 \rightarrow P_1$!

Proposition:

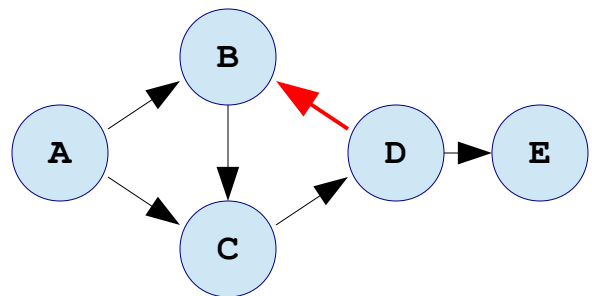
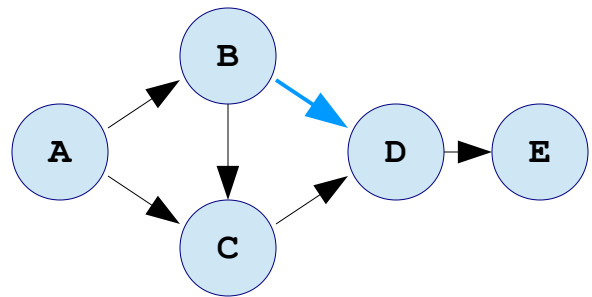
There is a deadlock situation if and only if there is a cycle in the wait graph.

– Can prove under some assumptions

Lab 2	Deadlock IV	11/13
<p>Mutexes and Dependencies</p> <p>process p calls acquire(lock):</p> <ul style="list-style-type: none"> – gets the lock and continues; – or doesn't get the lock, and adds edge (p→q) for q holding the lock <p>process q calls release(lock):</p> <ul style="list-style-type: none"> – releases the lock – removes all (p→q) for waiters p <i>(for this lock only)</i> 		<p>Can associate each edge with a mutex</p> <p>Add an edge (p→q : r) if process p calls acquire(r) while q holds it</p> <p>Remove all edges (p→q : r) into q for resource r when q calls release(r)</p>

Checking for directed cycles

- Initially, mark every node as ON
 - We turn it OFF when we know for certain that it can't be in any cycle
- Call an ON node p a NEXT node if every $(p \rightarrow q)$ has q OFF
 - Since no one that p waits on can be in a cycle, none can wait on anyone who waits on p ...who waits on p
- Whenever there's a NEXT node, turn it OFF.
 - Any ON at end \Rightarrow cycle exists



Lab 2	Deadlock and Lab 2	13/13
<p>Spin locks act like mutexes, but the read and write locks that you're implementing are a little more complicated:</p> <ul style="list-style-type: none">– Two levels: read and write– Must process in ticket order– 4 (= NOSPRD) devices, each with its own read and write locks and ticketing– What do these mean in terms of how many and what types of edges are added? <p>Also, there are several ways to store graphs in memory. The easiest is a list of all edges; but to look up, add, or remove an edge requires a scan of the list.</p>		