Critical Sections

1. Pieces of code in which you want only one process running at a time

2. These pieces usually access shared resources or data

3. Before-or-after behavior

   – Each transaction occurs either before or after each other one

4. Example:

   – 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 aquiring processes continue to wait

– release

– unlocks the mutex

Locking IV

– 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

Locking V

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**

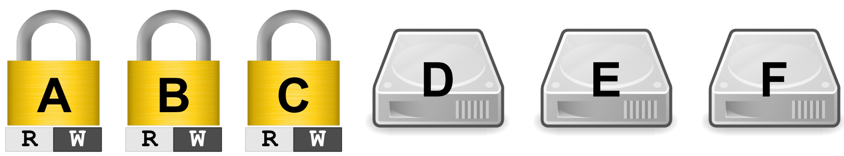
- **acquire L.lockb**

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

Lab 2	Deadlock I	8/13

Deadlock Example

P_1	P_2	P_3
`read_lock(A)`	`read_lock(B)`	`read_lock(C)`
`x = read(D)`	`x = read(E)`	`x = read(F)`
`write_lock(B)`	`write_lock(C)`	`write_lock(A)`
`write(E,x)`	`write(F,x)`	`write(D,x)`
`release(B)`	`release(C)`	`release(A)`
`release(A)`	`release(B)`	`release(C)`



Sample Execution

1. P_1`.read_lock(A)` *(gets lock on A)*
2. P_2`.read_lock(B)` *(gets lock on B)*
3. P_3`.read_lock(C)` *(gets lock on C)*
4. P_1`.write_lock(B)` *(gets ticket on B)*
5. P_2`.write_lock(C)` *(gets ticket on C)*
6. P_3`.write_lock(A)` *(gets ticket on A)*

Dependency graph for deadlock

- Nodes = processes
- Edges = waits-on relation

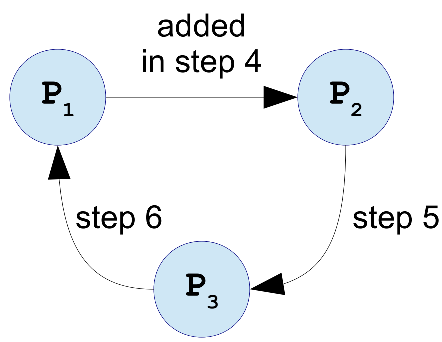
For example,

means that P_1 waits on P_2.

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

Convention:

- **The wait*er* adds the dependency (in e.g. acquire())**
- **The wait*ee* 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

Deadlock IV

Mutexes and Dependencies

process p calls acquire(lock):

- **gets the lock and continues;**
- **or doesn't get the lock, and adds edge (p→q) for q holding the lock**

process q calls release(lock):

- **releases the lock**
- **removes all (p→q) for waiters p** ***(for this lock only)***

Can associate each edge with a mutex

Add an edge (p→q : r) if process p calls acquire(r) while q holds it

Remove all edges (p→q : r) into q for resource r when q calls release(r)

Lab 2	Deadlock V	12/13

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*→*q*) has *q* OFF
  - *Since no one that p waits on can be in a cycle, none can wait on anyone who waits on anyone …who waits on p*
- Whenever there's a NEXT node, turn it OFF.
  - *Any ON at end ⇒ cycle exists*