Advanced Operating Systems (CS 202)

Synchronization (Part II)

What are the sources of concurrency?

- Multiple user-space processes
  - On multiple CPUs
- Device interrupts
- Workqueues
- Tasklets
- Timers

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
→ if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos]) {
      goto out;
    }
  }
```

Scull is the Simple Character Utility for Locality Loading (an example device driver from the Linux Device Driver book)

Pitfalls in **scull**

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Pitfalls in `scull`

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Pitfalls in **`scull`**

- *Race condition*: result of uncontrolled access to shared data

```
if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL)
  if (!dptr->data[s_pos]) {
    goto out;
  }
}
```

Memory leak

Synchronization primitives

- Lock/Mutex
  - To protect a shared variable, surround it with a lock (critical region)
  - Only one thread can get the lock at a time
  - Provides mutual exclusion
- Shared locks
  - More than one thread allowed (hmm…)
- Others? Yes, including Barriers (discussed in the paper)

Synchronization primitives (cont'd)

- Lock based
  - Blocking (e.g., semaphores, futexes, completions)
  - Non-blocking (e.g., spin-lock, …)
    - Sometimes we have to use spinlocks
- Lock free (or partially lock free ☺)
  - Atomic instructions
  - seqLocks
  - RCU
  - Transactions (next time)

Naïve implementation of spinlock

- Lock(L):

```
While(test_and_set(L));
//we have the lock!
//eat, dance and be merry
```

- Unlock(L)

```
L=0;
```

Why naïve?

- Works? Yes, but not used in practice
- Contention
  - Think about the cache coherence protocol
  - Set in test and set is a write operation
    - Has to go to memory
    - A lot of cache coherence traffic
    - Unnecessary unless the lock has been released
    - Imagine if many threads are waiting to get the lock
- Fairness/starvation

Better implementation
Spin on read

- Assumption: We have cache coherence
  - Not all are: e.g., Intel SCC
- Lock(L):

```
while(L==locked); //wait
if(test_and_set(L)==locked) go back;
```

- Still a lot of chattering when there is an unlock
  - Spin lock with backoff

Bakery Algorithm

```
struct lock {
    int next_ticket;
    int now_serving; }
```

- Acquire_lock:

```
int my_ticket = fetch_and_inc(L->next_ticket);
while(L->new_serving!=my_ticket); //wait
//Eat, Dance and me merry!
```

Still too much chatter

- Release_lock:

```
L->now_serving++;
```

Comments? Fairness? Efficiency/cache coherence?

Anderson Lock (Array lock)

- Problem with bakery algorithm:
  - All threads listening to next_serving
    - A lot of cache coherence chatter
  - But only one will actually acquire the lock
  - Can we have each thread wait on a different variable to reduce chatter?

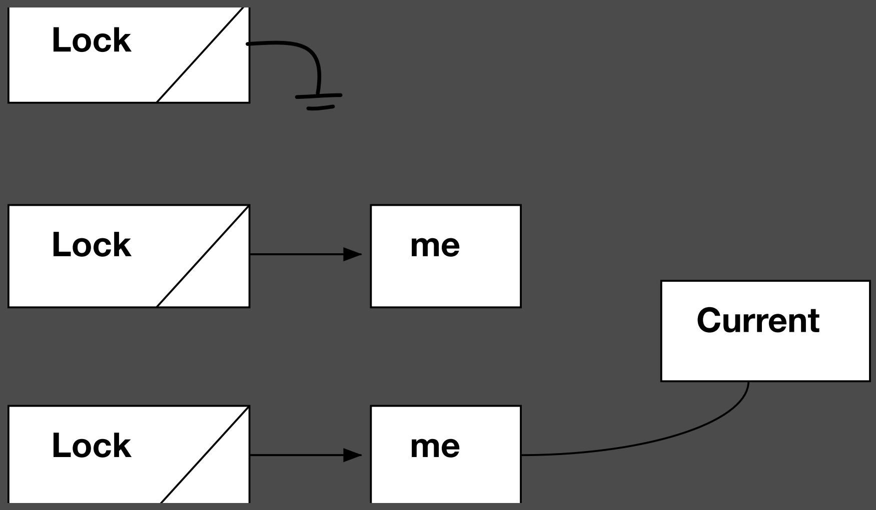
Anderson's Lock

- We have an array (actually circular queue) of variables
  - Each variable can indicate either lock available or waiting for lock
    - Only one location has lock available

```
Lock(L):
    my_place = fetch_and_inc (queuelast);
    while (flags[myplace mod N] == must_wait);
Unlock(L)
    flags[myplace mod N] = must_wait;
    flags[mypalce+1 mod N] = available;
```

Fair and not noisy – compare to spin-on-read and bakery algorithm
Any negative side effects?

MCS Lock



- Each node has:

```
struct node {
bool got_it;
Next; //successor}
```

```
Lock(L, me)
join(L); //use fetch-n-store
while(got_it == 0);
```

```
Unlock(L,me)
 remove me from L
 signal successor
(setting got it to 0)
```

Race condition!

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked                // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil         // no known successor
        if compare_and_swap (L, I, nil)
            return
            // compare_and_swap returns true iff it swapped
        repeat while I->next = nil            // spin
    I->next->locked := false
```

- What if there is a new joiner when the last element is removing itself

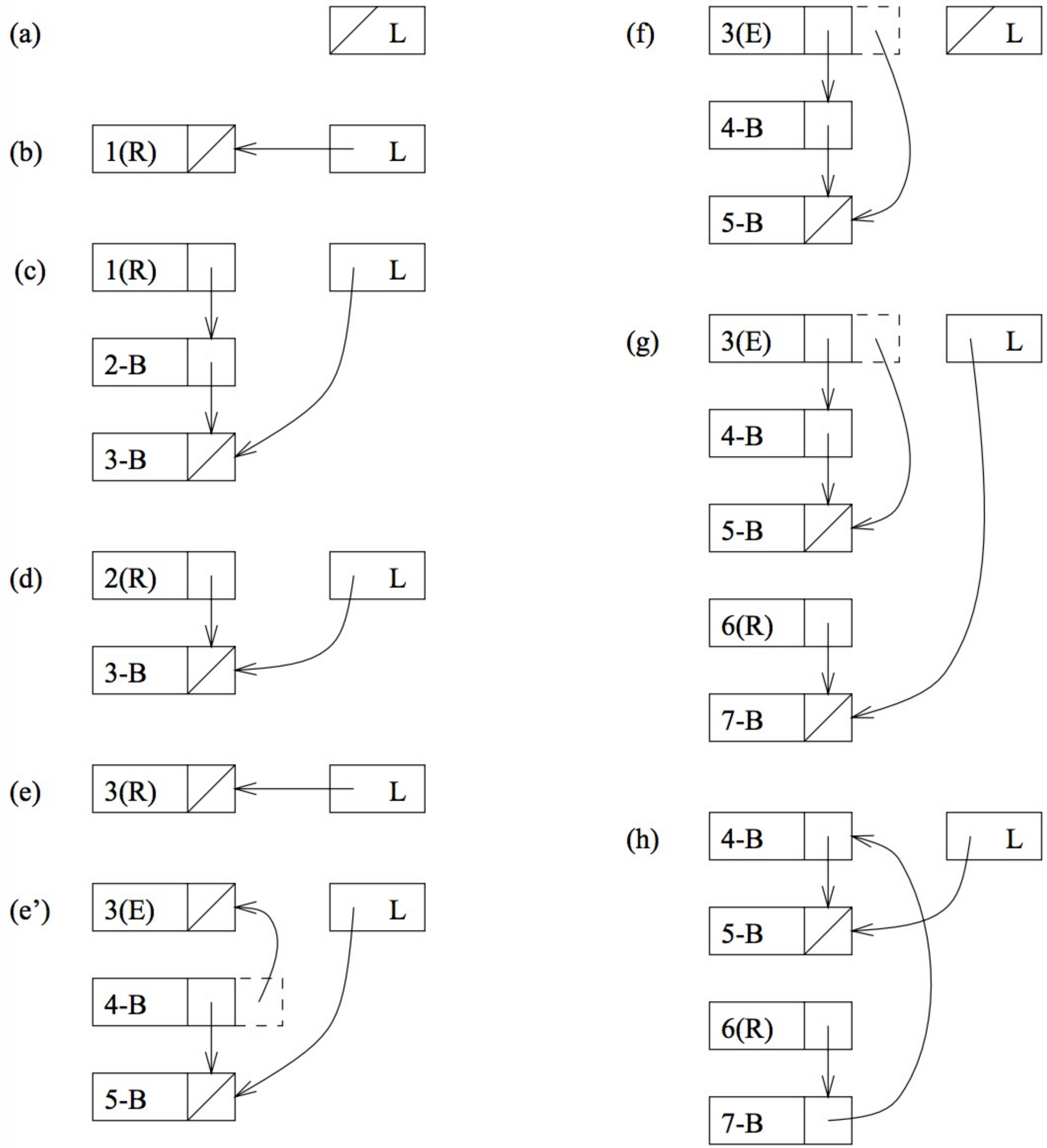
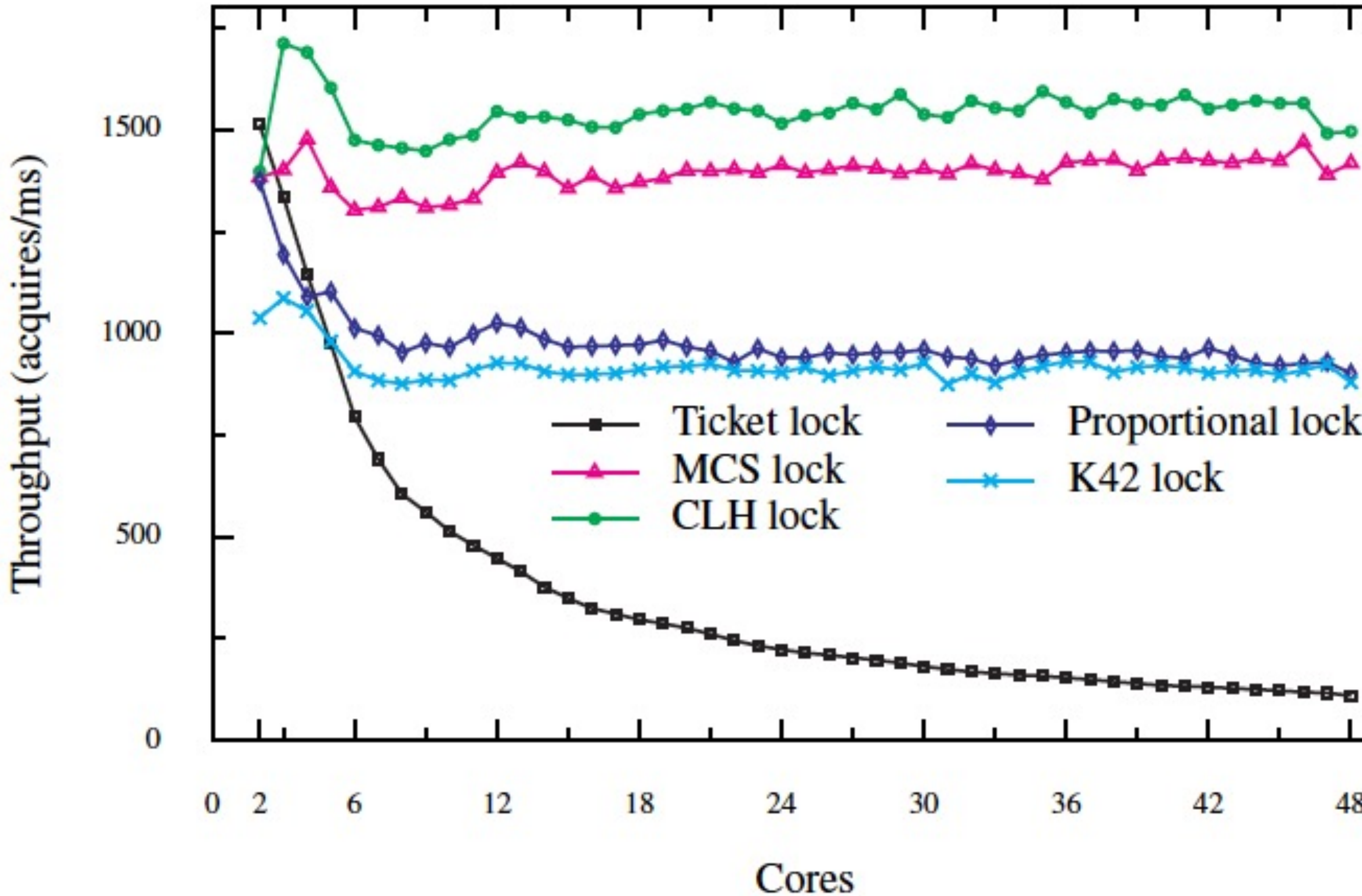


Figure 1: Pictorial example of MCS locking protocol in the presence of competition.

Performance impact

Table II. Increase in Network Latency (relative to that of an idle machine) on the Butterfly Caused by 60 Processors Competing for a Busy-Wait Lock.

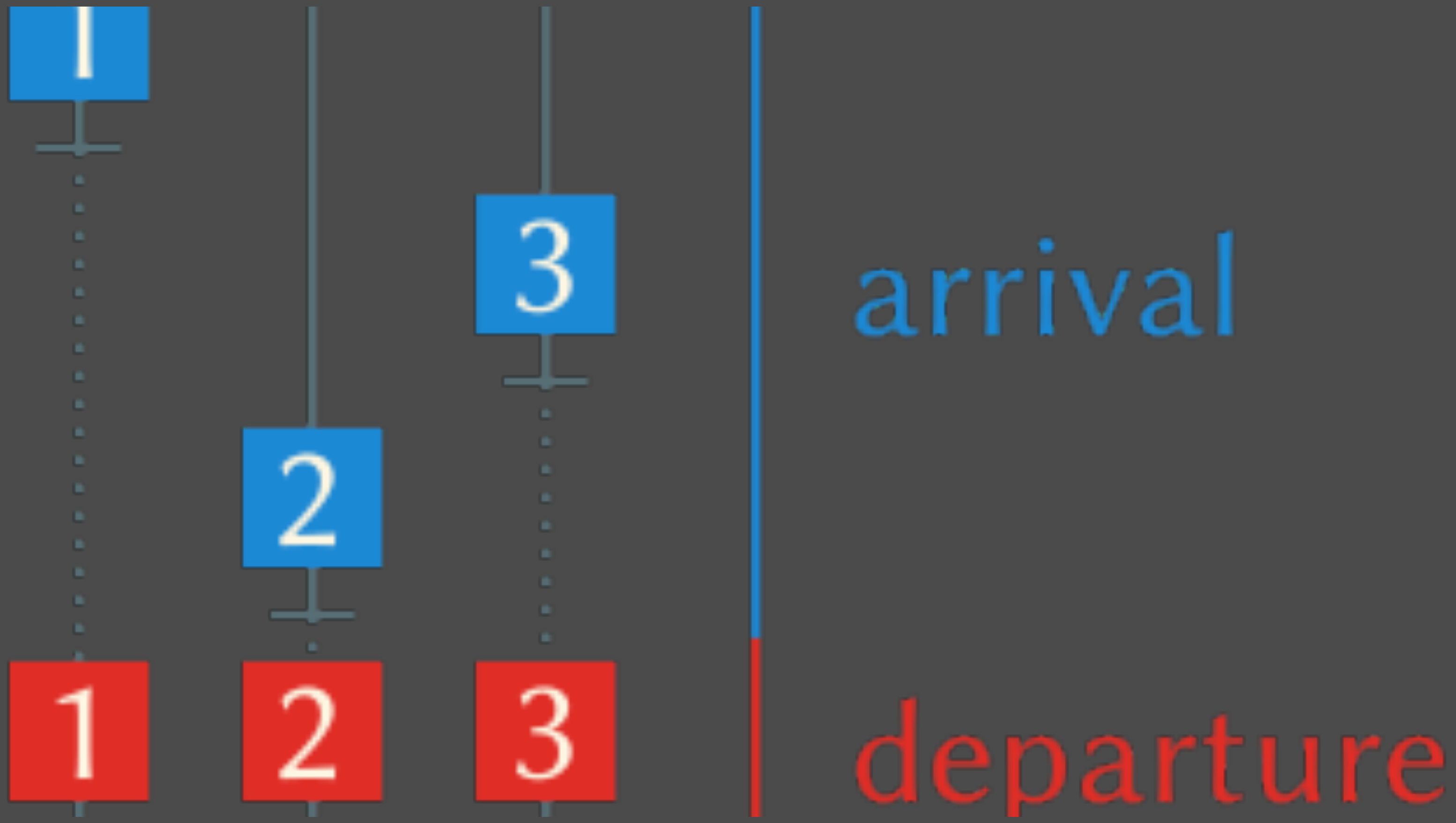
Busy-wait Lock	Increase in network latency measured from: Lock node (%)	Idle node (%)
test_and_set	1420	96
test_and_set w/linear backoff	882	67
test_and_set w/exp. backoff	32	4
ticket	992	97
ticket w/prop. backoff	53	8
Anderson	75	67
MCS	4	2



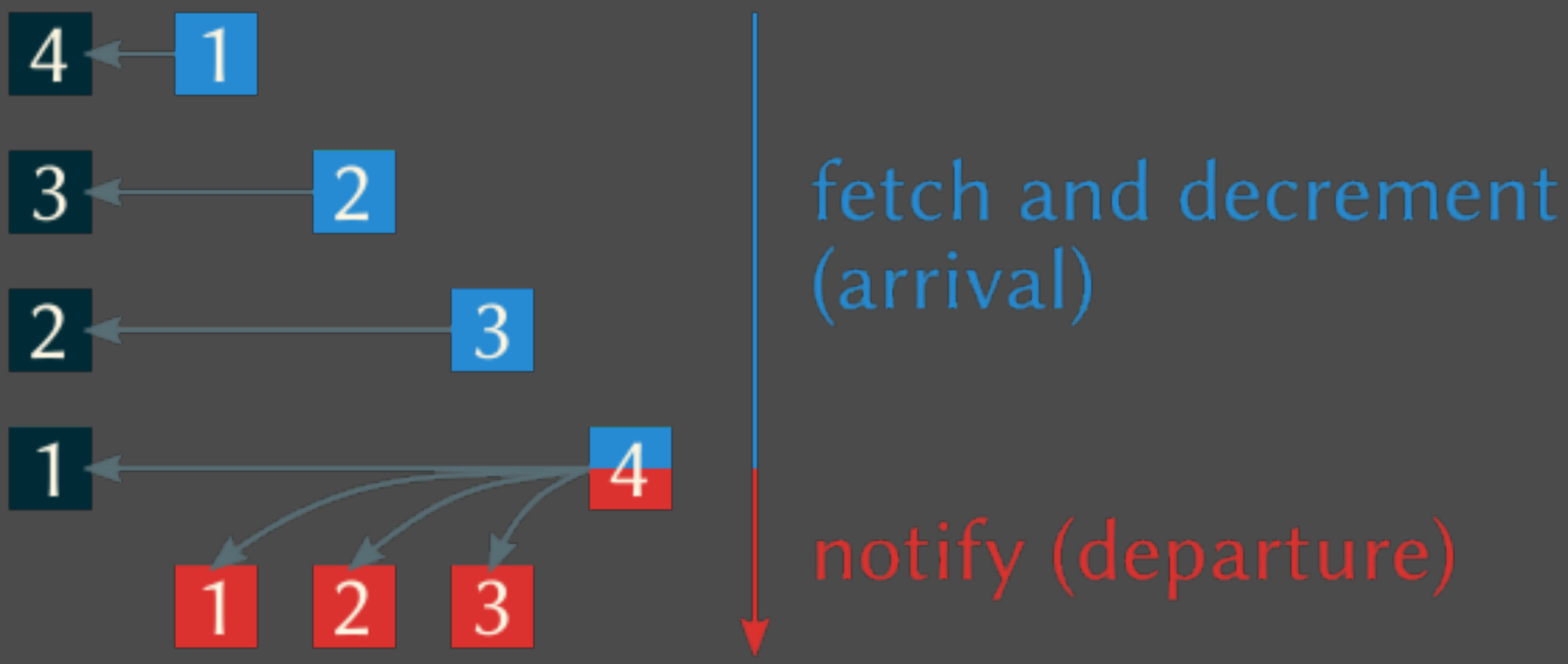
From the Boyd-Wickizer et al paper, “Non-scalable locks are dangerous”
CLH and K42 are MCS variants

BARRIERS/FYI

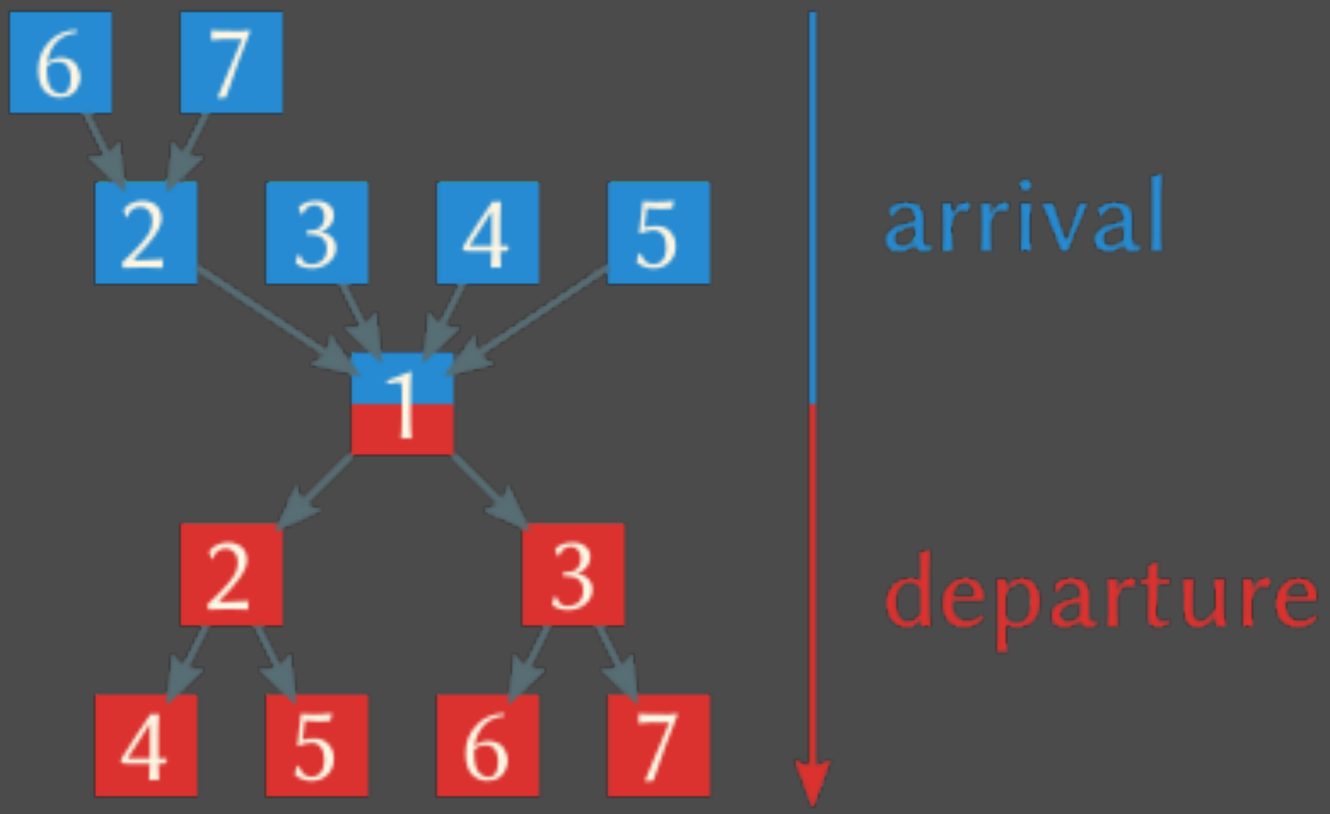
Barriers



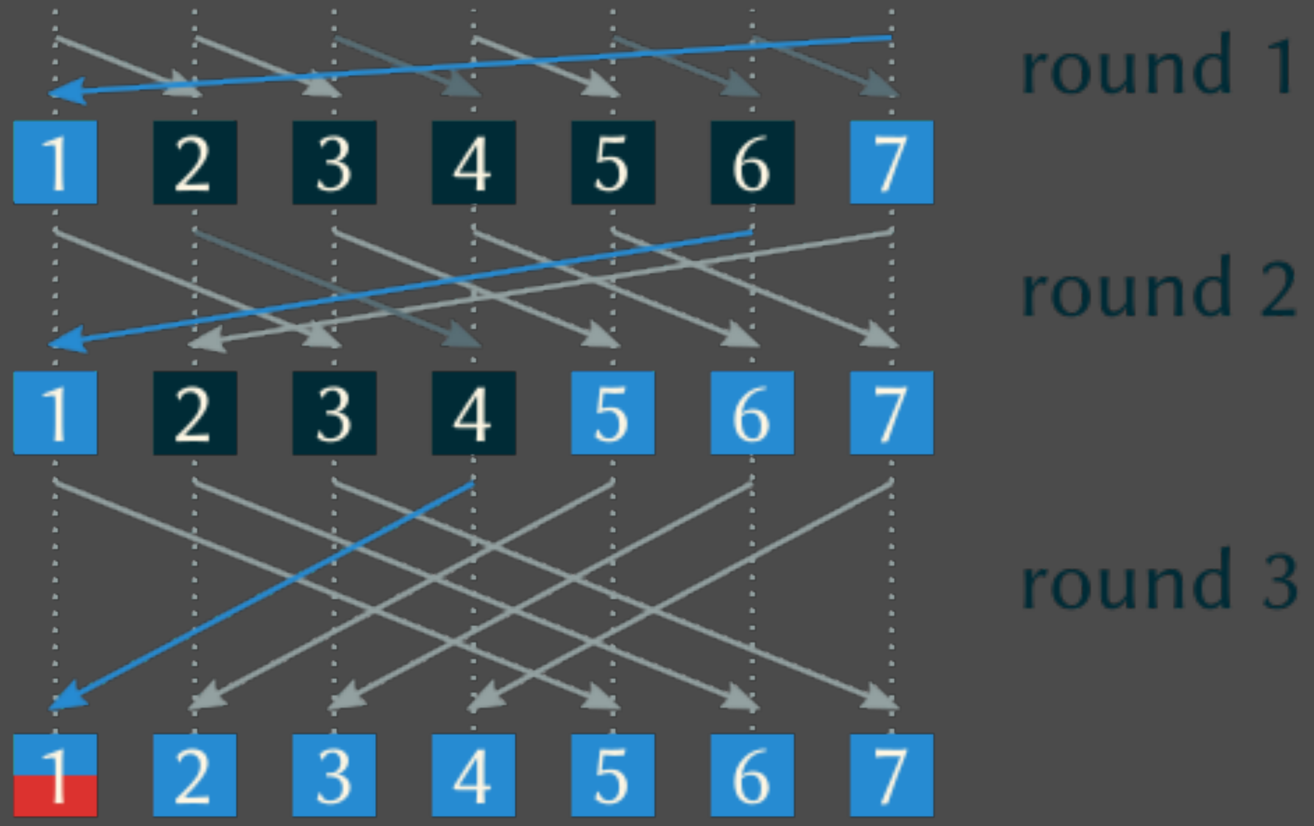
Linear barriers



Tree barrier (MCS paper)



Dissemination Barrier (Hensgen/Finkel)



Counter based performs best!