
بسم الله الرحمن الرحیم

Concurrency

OS class

Virtualization

Concurrency

Persistence

OS class

Virtualization [Done so far!]

e.g., “page replacement": so many things done, but straightforward, step after step!

Concurrency

Persistence

OS class

Virtualization

Concurrency [Our focus now!]

Persistence

OS class

Virtualization

Concurrency → hardest, most secret, most on interviews!

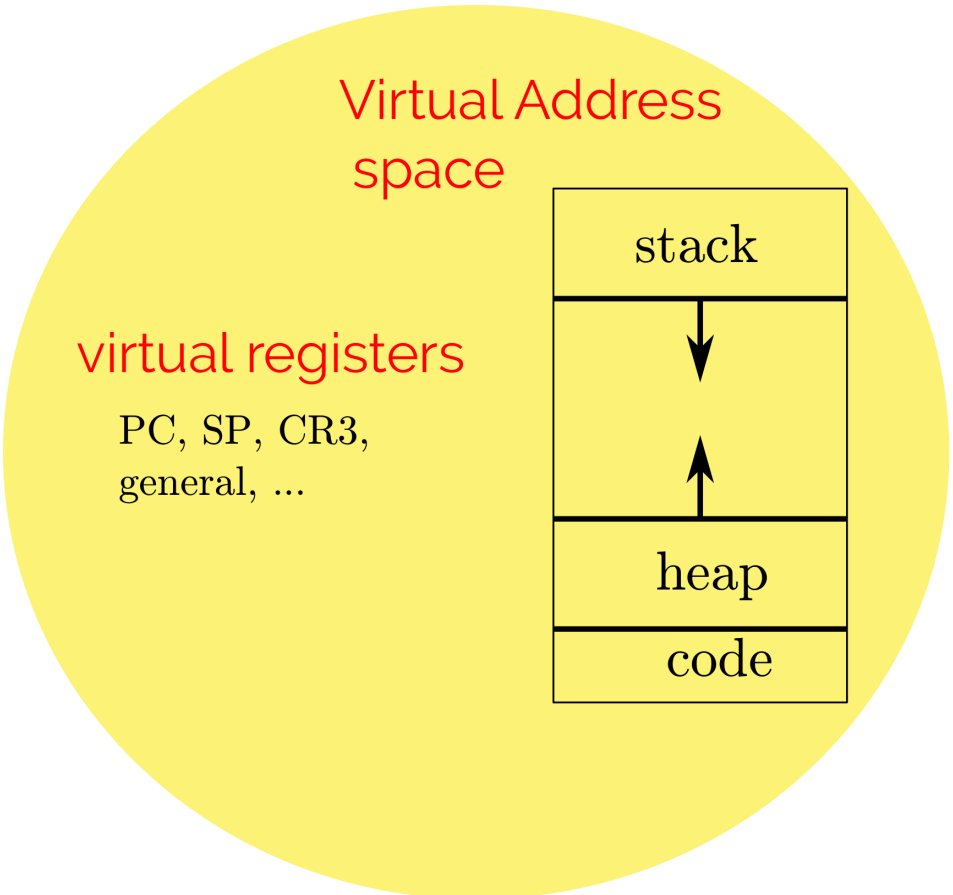
staring on 3-5 lines of code and puzzling over whether it is correct or not!

Reason: unlike other programs (single-threaded), in concurrent (multi-threaded) programs *multiple things are going on at once*→ hard to figure out full space of possibilities

Persistence

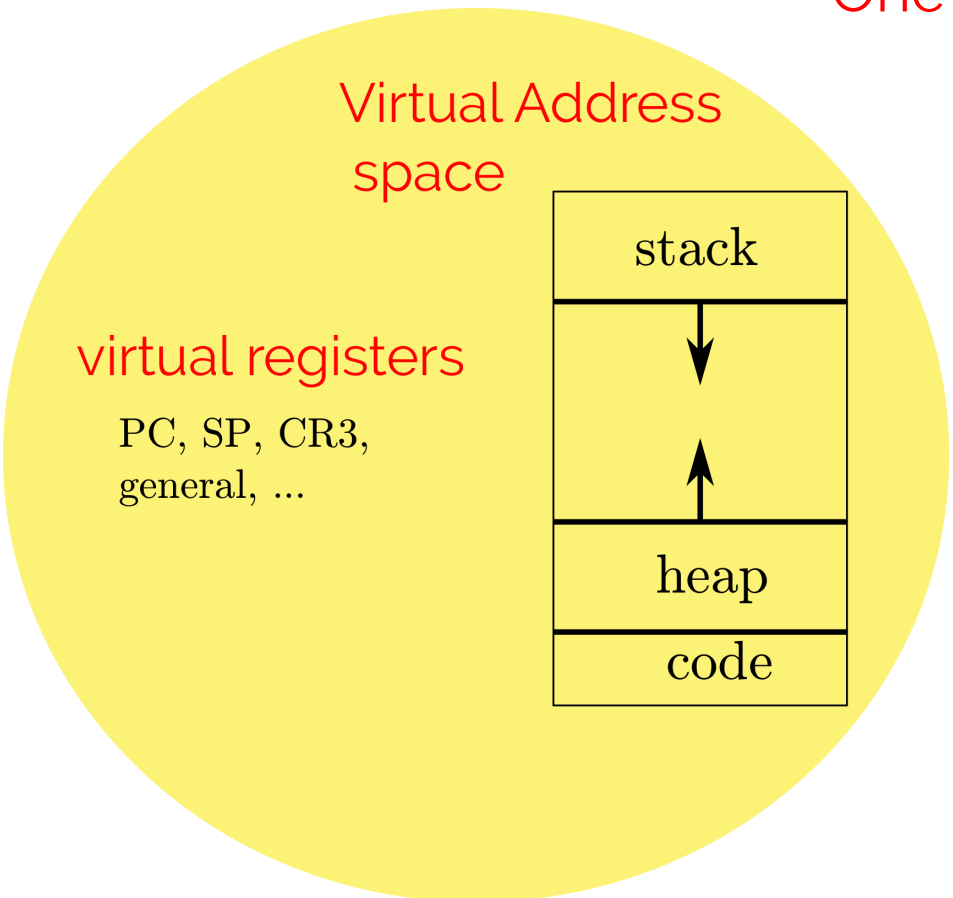
Introduction

Classic process: we call it now a **single threaded process**

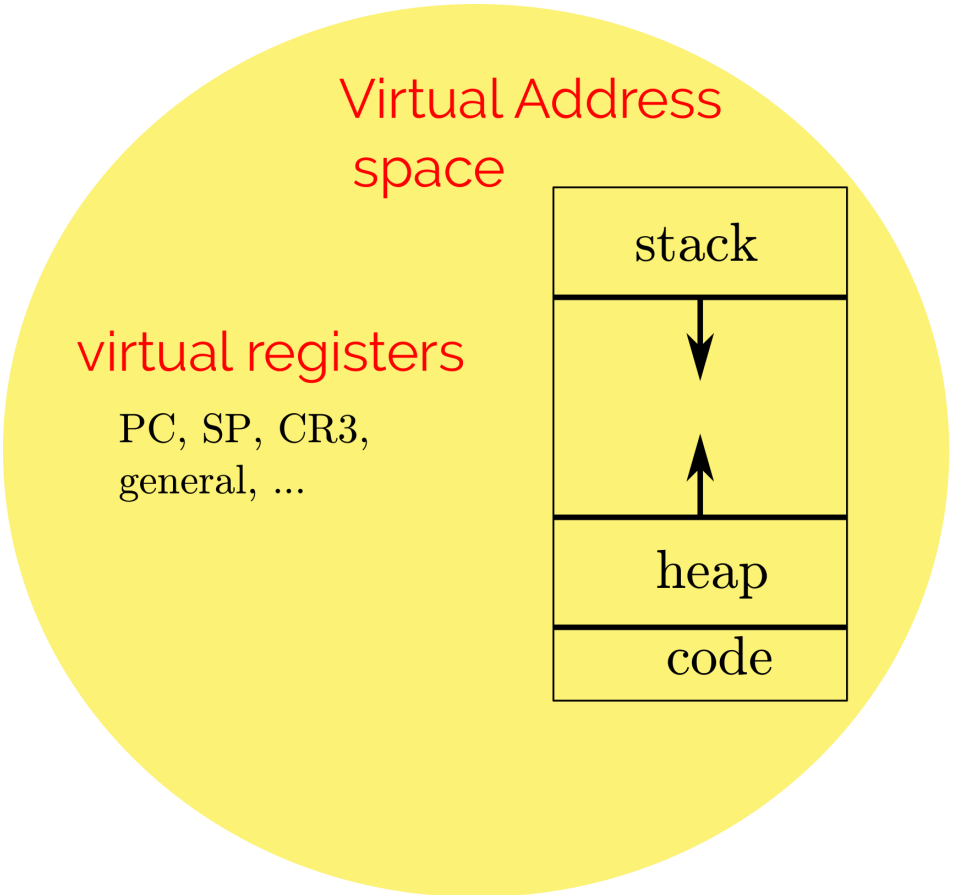


Classic process: we call it now a **single threaded** **process**

One thing at a time



Classic process: we call it now a **single threaded process**



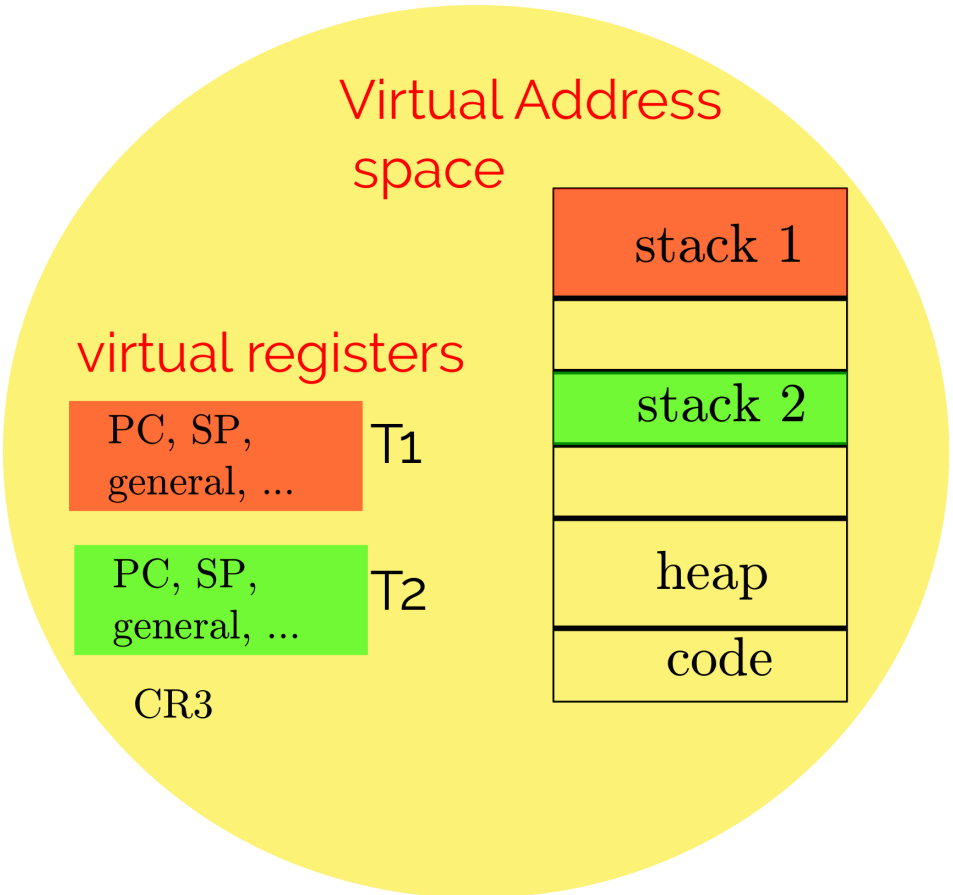
OS could switch to another process

- save (old) registers, restore (new) registers
- switch page tables

In many cases we want many “activities/computations" going on with in process (address space) at same time

$\Rightarrow$ a **multi-threaded process**

Multi threaded process: (with two threads)



Multiple threads within a single process

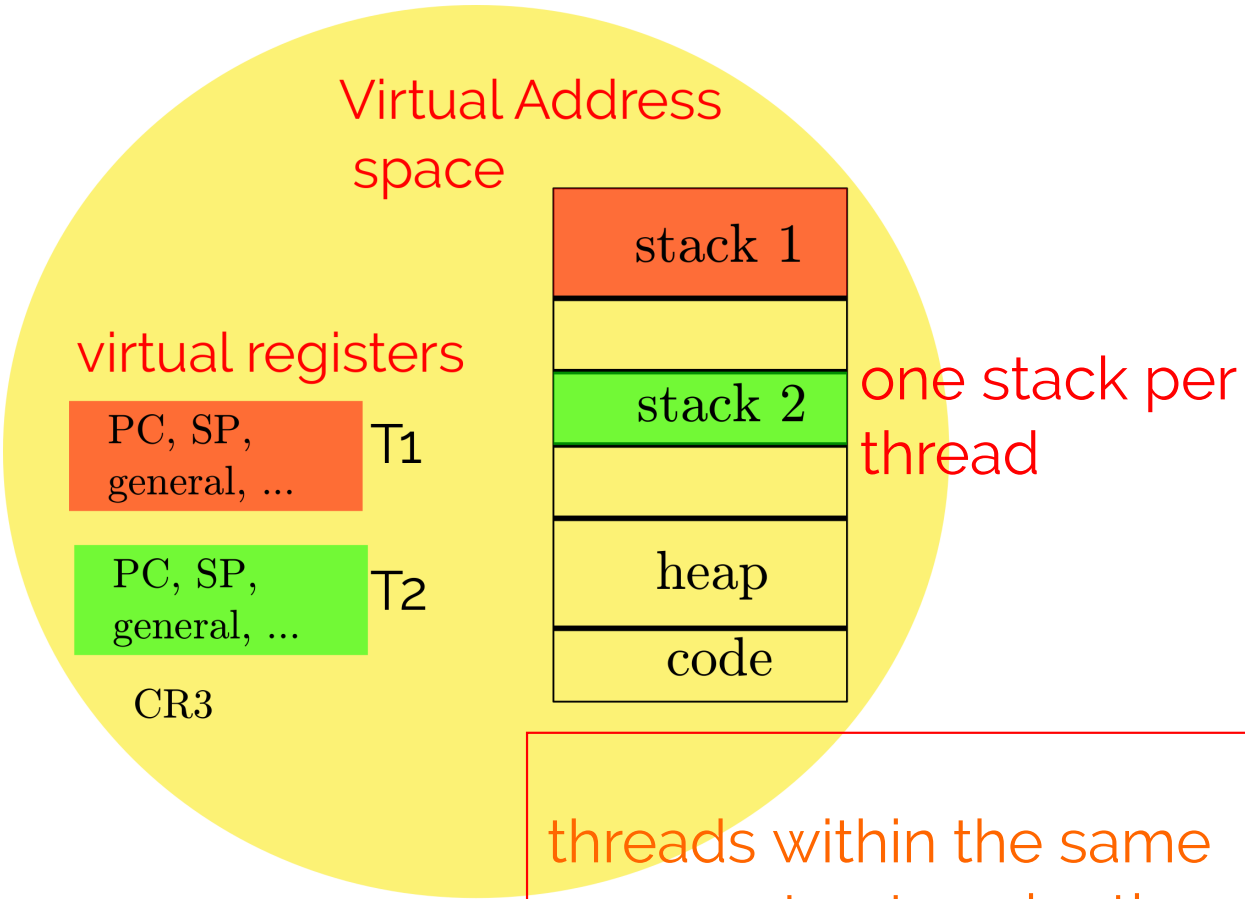
share:

- Process ID (PID)
- Address space (*normally*, code + heap)
- Open file descriptors
- Current working directory
- User and group id

do not share:

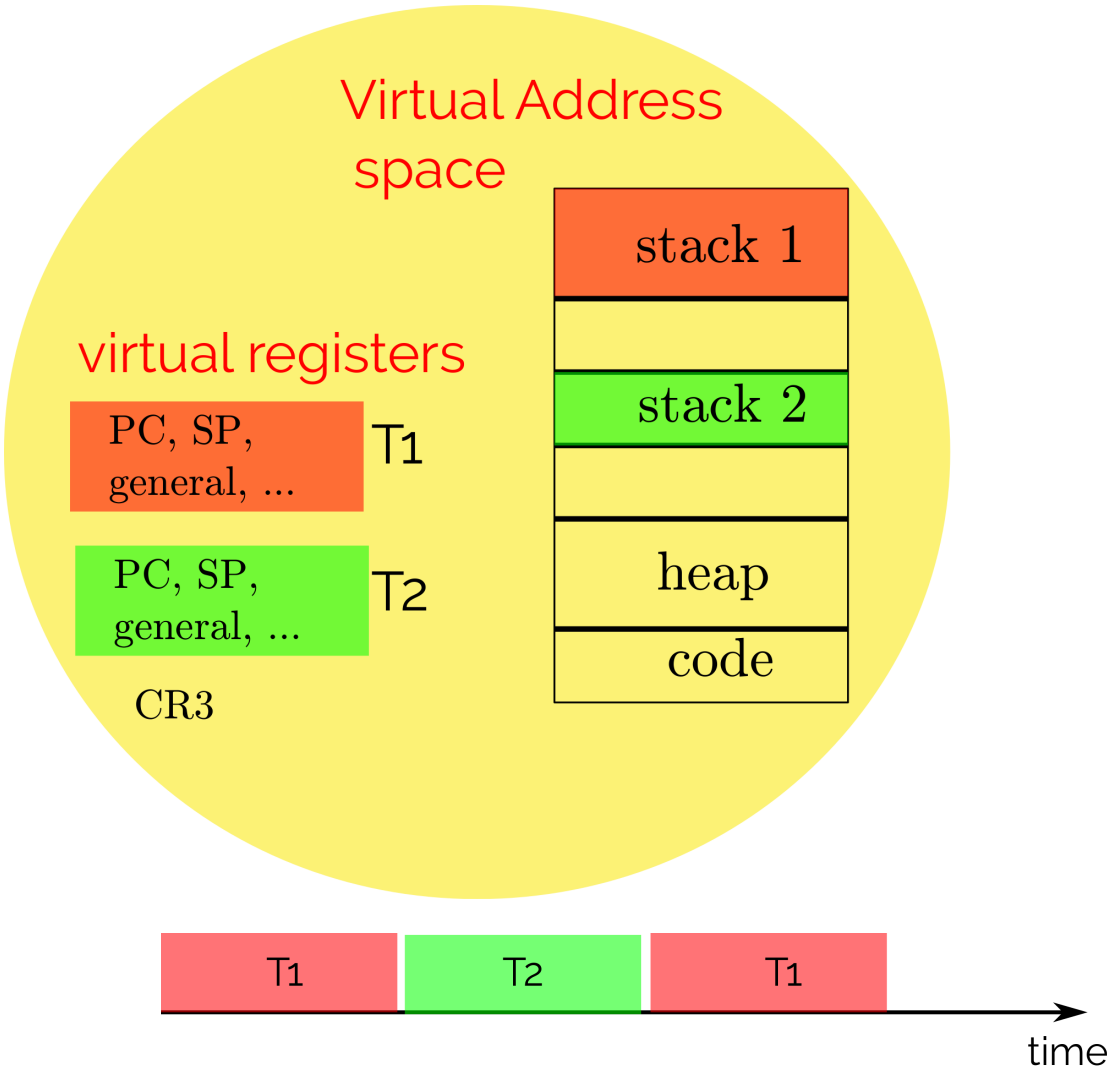
- Thread ID (TID)
- Set of registers, including PC and SP
- Stack for local variables and return addresses (although accessible by other threads)

Multi threaded process: (with two threads)



Multi threaded process: (with two threads)

A thread is a single execution sequence that represents a separately schedulable task



Context switch

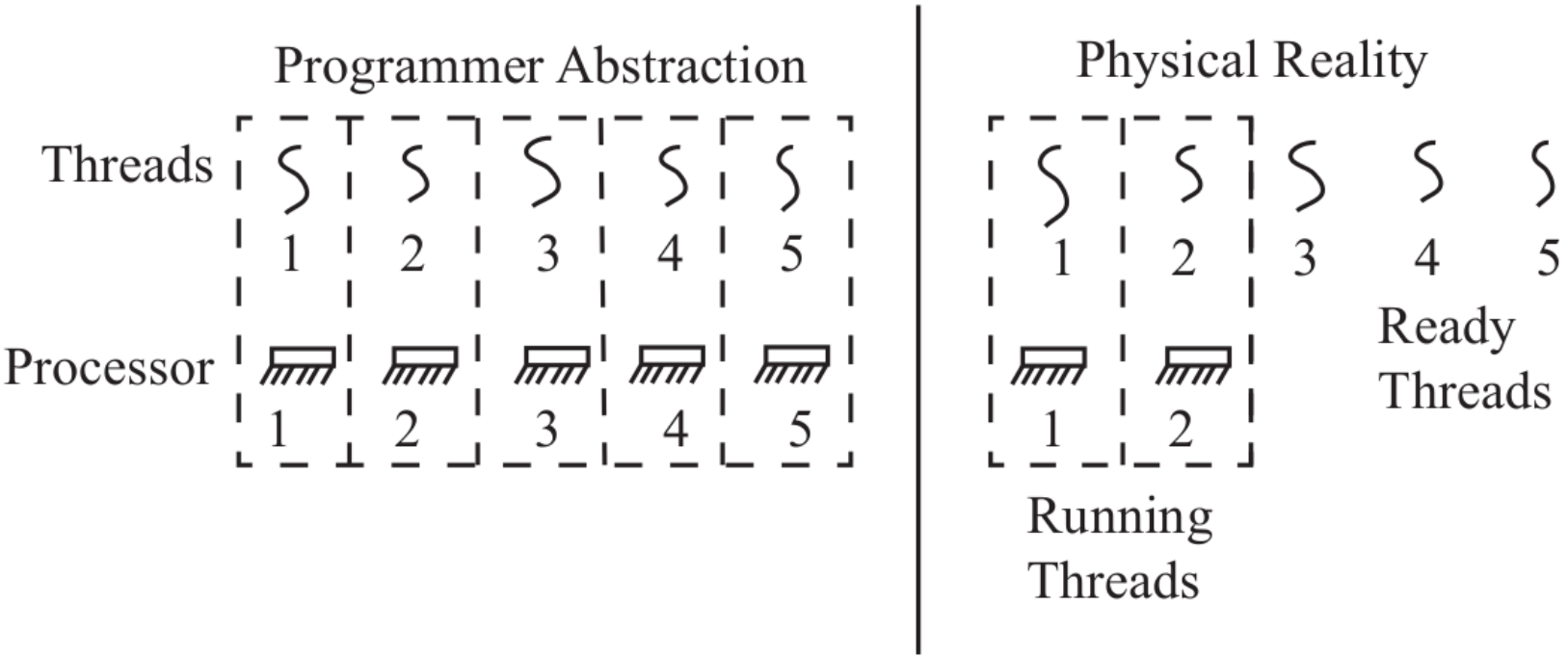
between threads

between processes

in both: saving (old) registers and restoring (new) registers

just in switching between processes: switching the address space (i.e., PTBR is changed)

Threads extend *virtualization* of the processor, providing the illusion that the system has an infinite number of processors.



Motivation

Motivation

Parallelism

Overlap

Easier programming model

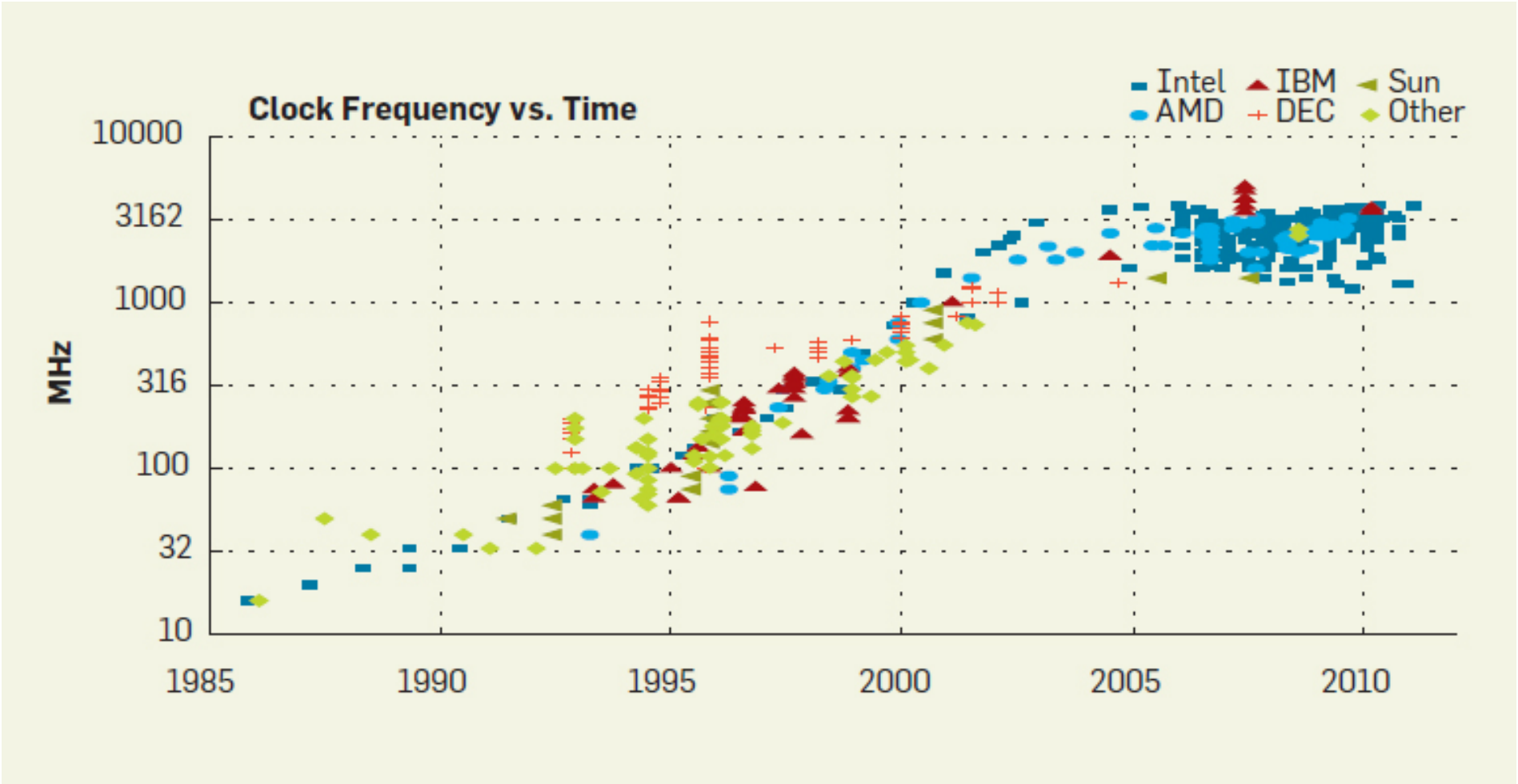
Motivation

Parallelism

Overlap

Easier programming model

Parallelism



https://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext

Parallelism

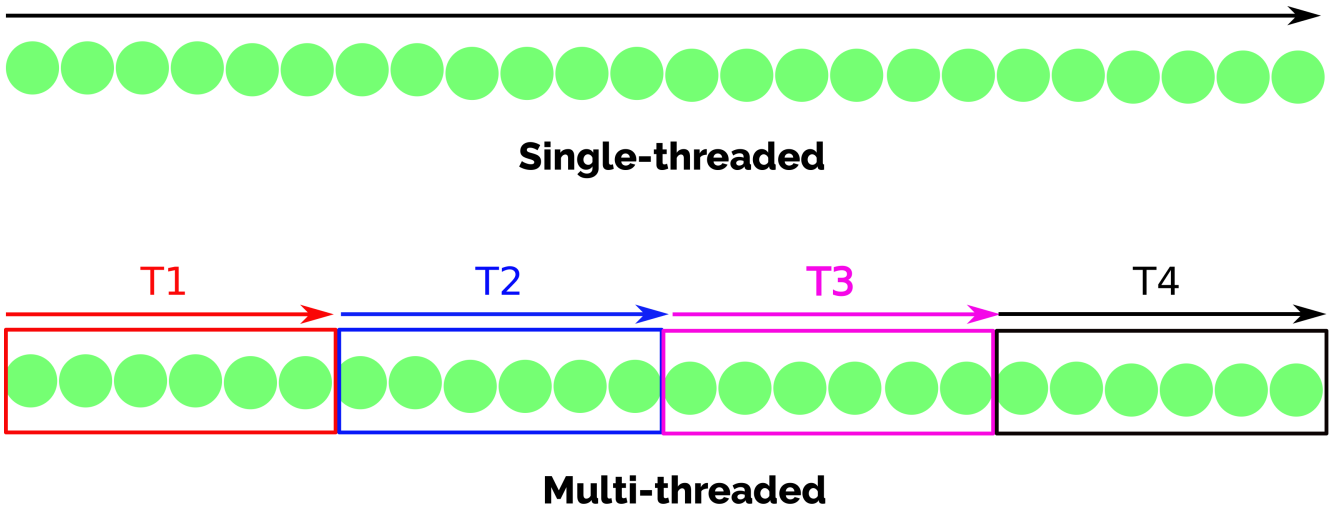
CPU Trend

- same speed
- more cores

Faster programs $\Rightarrow$ concurrent execution

Goal: Write applications that fully utilize many CPUs ...

- OS itself is a concurrent program!


Single-threaded
T1
T2
T3
T4
Multi-threaded

Aside: communicating Processes

Build apps from many communicating processes

- Example: Chrome (process per tab)
- Communicate via `pipe()` or similar

Pros/cons?

- don't need new abstractions
- sharing data among independent address spaces
- expensive context switching (why expensive?)

Motivation

Parallelism

Overlap

Easier programming model

Overlap

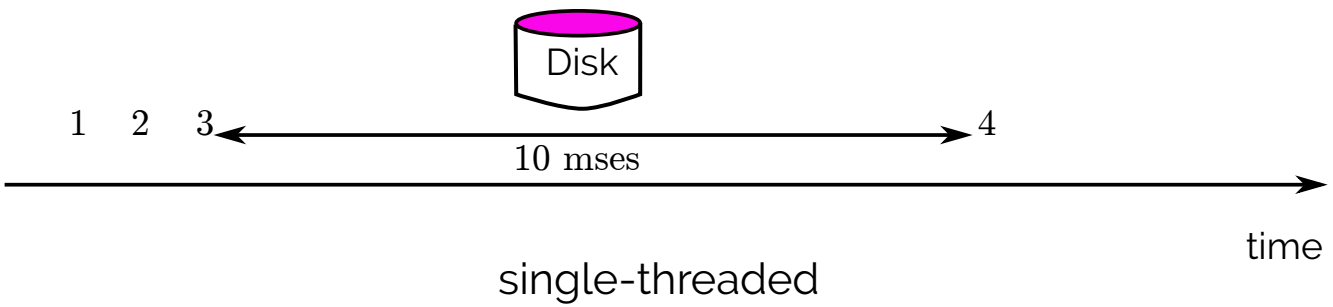
Web server

1. get request
2. parse
3. read data from disk
4. return data

Overlap

Web server

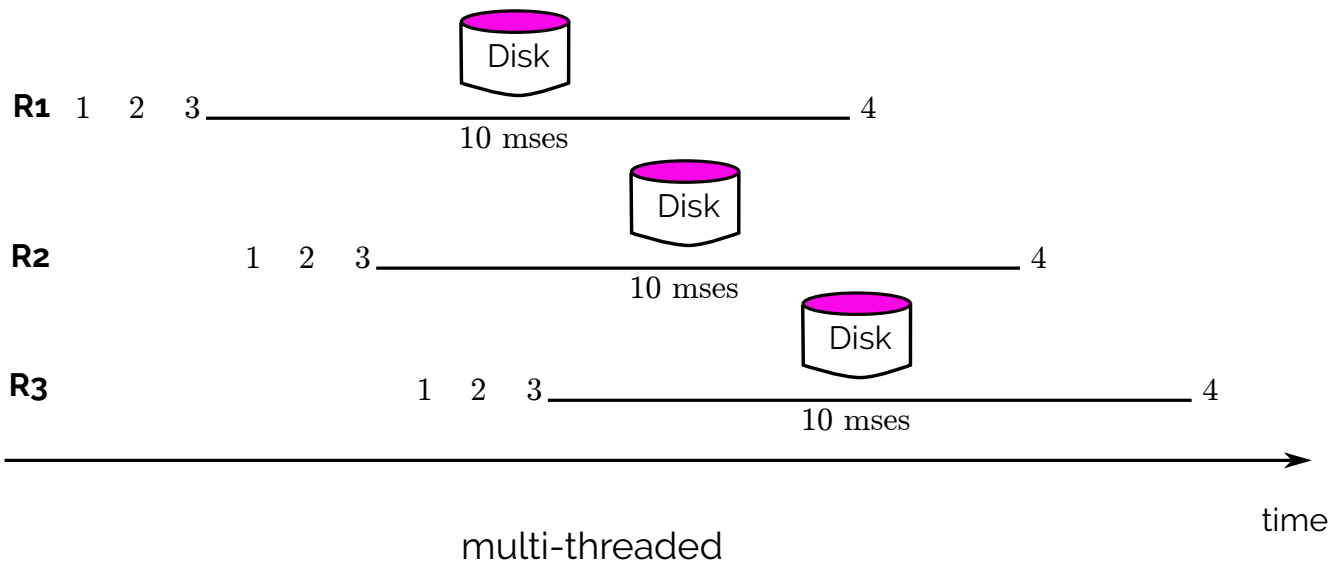
1. get request
2. parse
3. read data from disk
4. return data



Overlap

Web server

1. get request
2. parse
3. read data from disk
4. return data



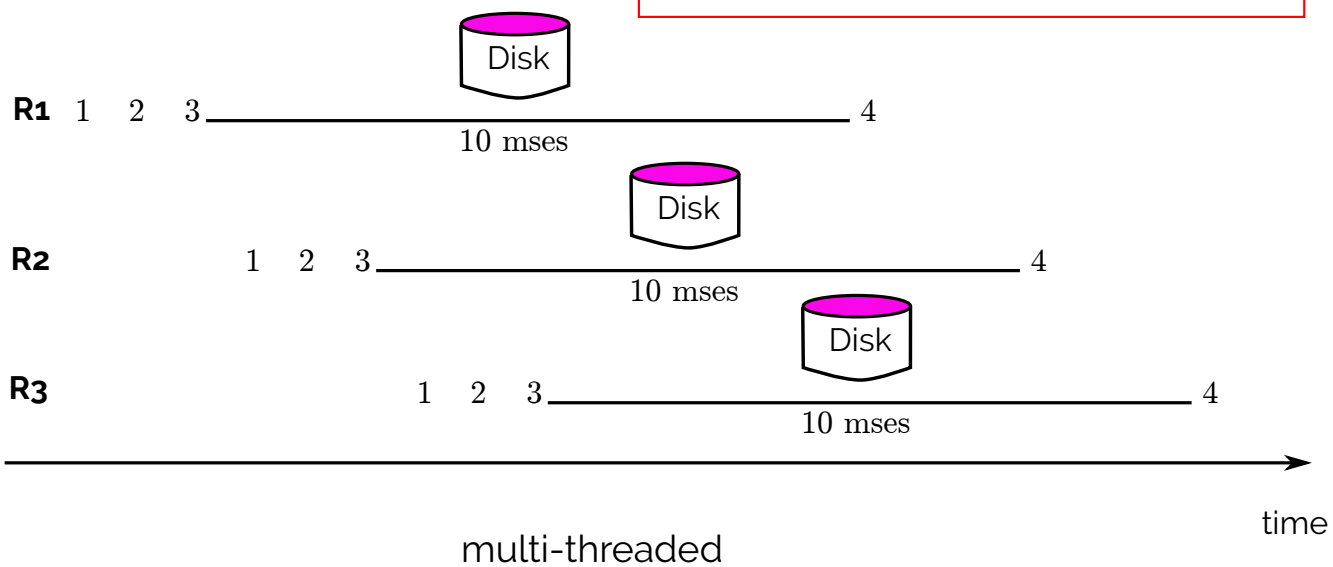
multi-threaded

Overlap

Web server

1. get request
2. parse
3. read data from disk
4. return data

multi-threading in a uni-processor setting is also important

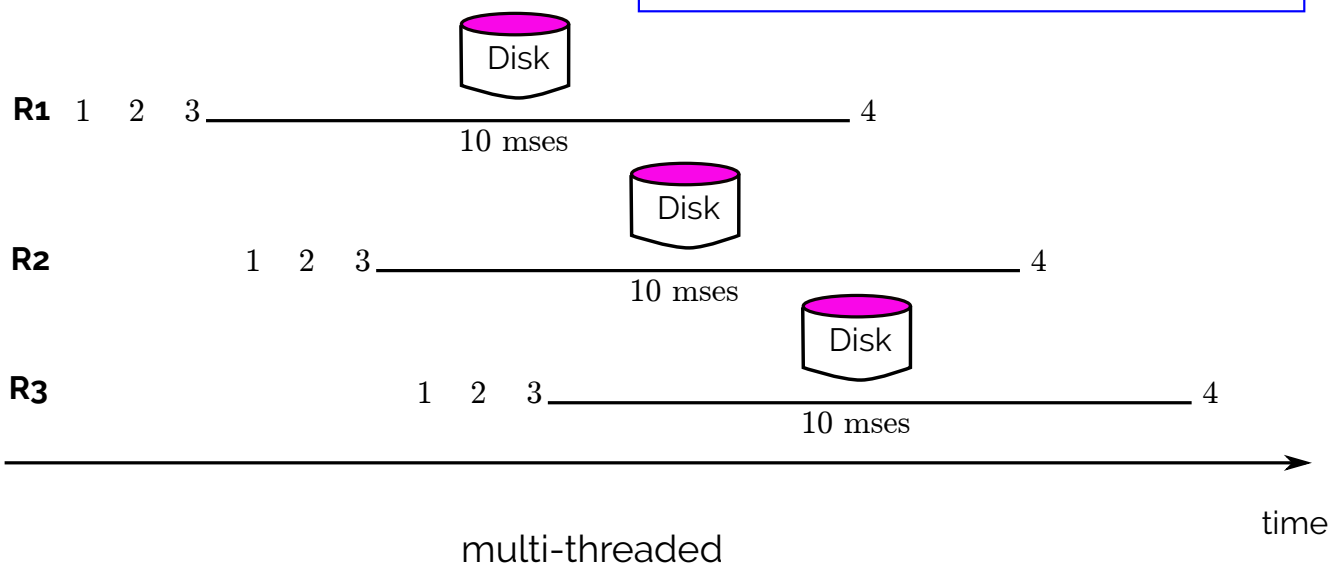


Overlap

Web server

1. get request
2. parse
3. read data from disk
4. return data

number of theards may be many more than number of cores



multi-threaded

Motivation

Parallelism

Overlap

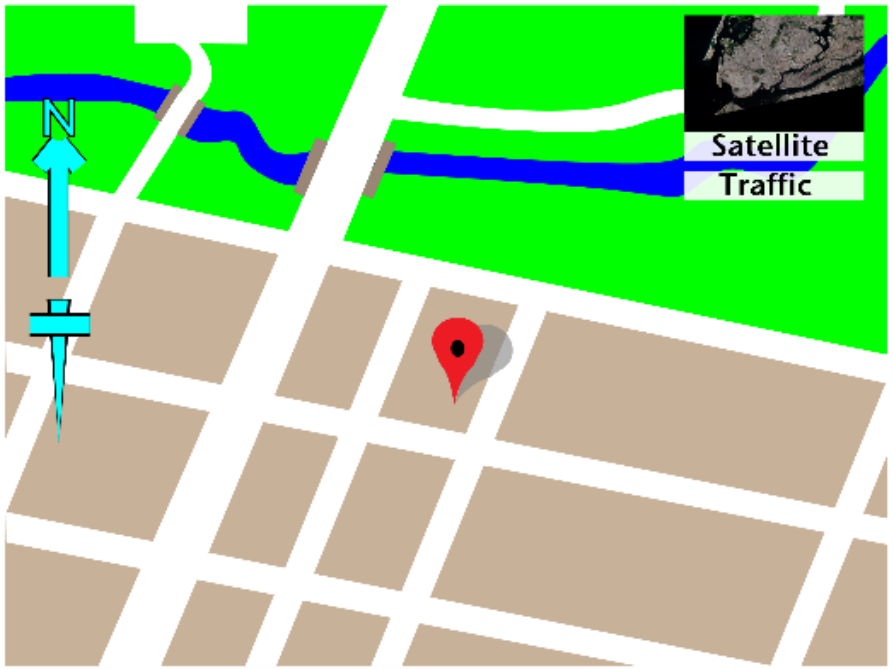
Easier programming model

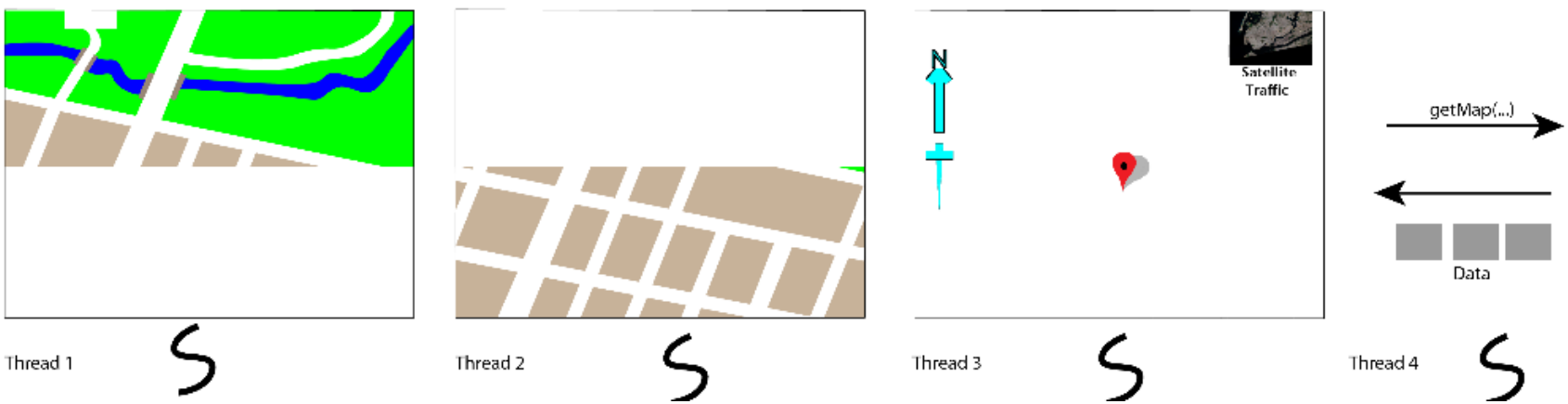
Easier Programming Model

Programs often interact with or simulate real world applications that have concurrent activities.

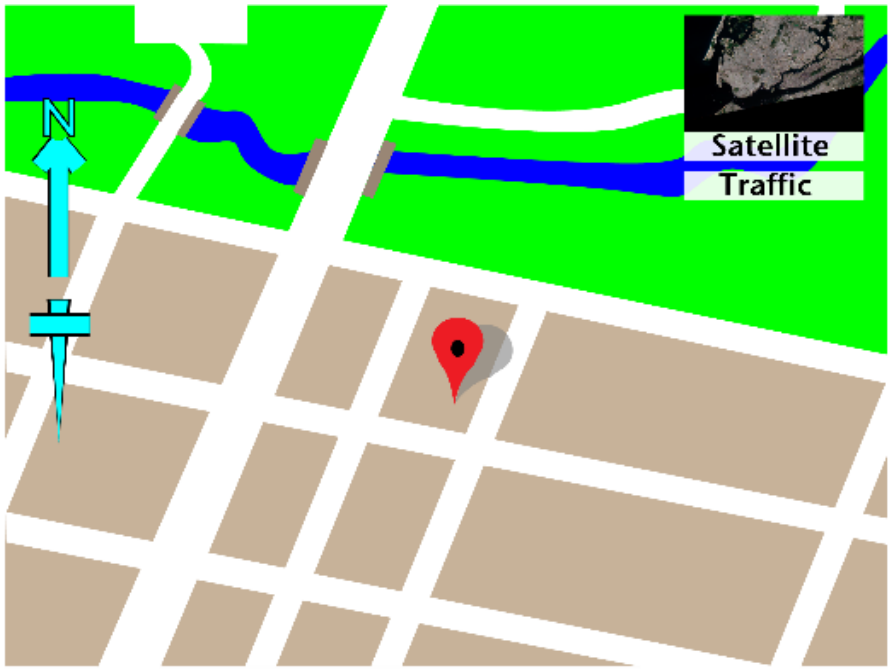
Threads allow the programmer to express an application's concurrency by writing each concurrent task as a separate thread.

In the Earth Visualizer example, two threads each draw part of the scene, a third thread manages the user interface widgets, and a fourth thread fetches new data from a remote server.

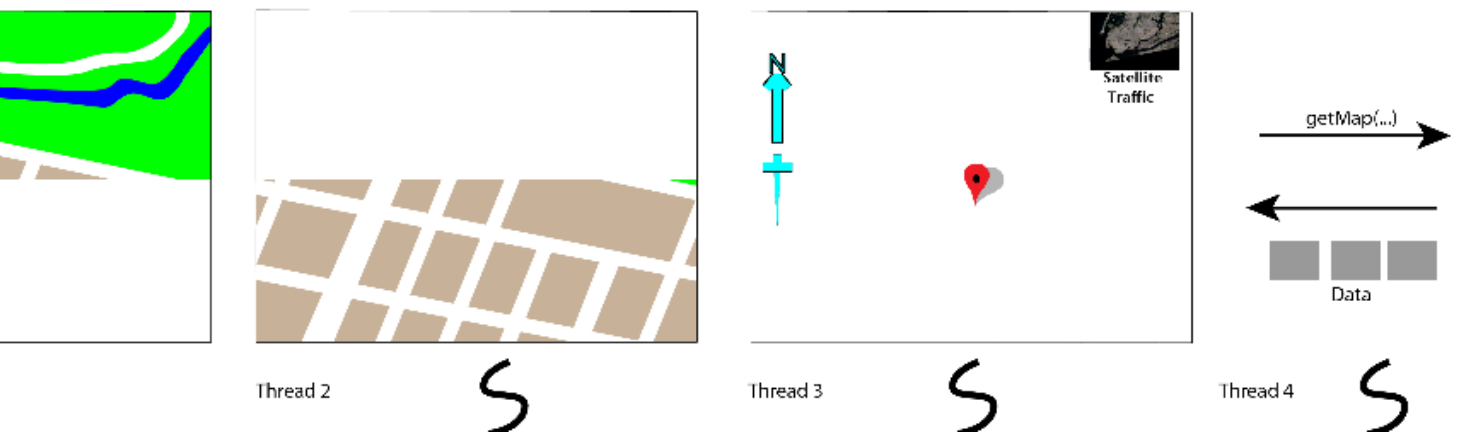




nple,
ch
e
hread
ıser
ets,
read
ata



Although one could imagine manually writing a program that interleaves these activities (e.g., draw a few pixels on the screen, then check to see if the user has moved the mouse, then check to see if new image data have arrived on the network, . . .), using threads can make things much simpler.



Note

Programmers are used to thinking sequentially, and threads allow them to write the code for each task as a separate, sequential program rather than requiring them to write one program that somehow interleaves all of these activities.

Concurrency Problems

Concurrency Problems

Reordering instructions

Interleavings of threads

Concurrency Problems

Reordering instructions

Due to H/W and compiler

Interleavings of threads

Reordering instructions

Modern compilers and hardware will reorder instructions to improve performance.

Invisible to single-threaded programs.

Dependencies within a sequence of instruction are preserved.

May become visible when multiple threads interact and observe intermediate states.

Thread 1

```
x = 2;
y = 1;
```

Thread 2

```
while (!y);
print(x)
```

Assume initially: `x=0, y=0`

What will be printed for `x`?

- ☐ 0
- ☐ 2
- ☐ both are possible

Thread 1

```
x = 2;
y = 1;
```

Thread 2

```
while (!y);
print(x)
```

Assume initially: `x=0, y=0`

What will be printed for `x`?

- ☐ 0
- ☐ 2
- ☑ both are possible

Aside: Memory Barrier

A memory barrier instruction prevents the compiler and hardware from reordering memory accesses across the barrier-no accesses before the barrier will be moved after the barrier and no accesses after the barrier will be moved before the barrier.

Concurrency Problems

Reordering instructions

Interleavings of threads

Due to the OS scheduler

Interleavings of threads

Demo ...

ostep-code/threads-intro/t1.c

pthread library

Understanding the problem

- Marking the critical section
- Looking at under-hood assembly codes
- using `objdump -d`
- Focusing on assembly of the critical section ...

```
1 mov    0x2016bc(%rip),%eax
2 add    $0x1,%eax
3 mov    %eax,0x2016b3(%rip)
```

Thread 1		Thread 2		
pc	eax	pc	eax	counter (in memory,shared)
1	1000			1000
2	1001			1000
3	1001			1001
		– – – – – –Context switch– – – – – –		
		1	1001	1001
		2	1002	1001
		3	1002	1002

Fine! (success)

```
1 mov    0x2016bc(%rip),%eax
2 add    $0x1,%eax
3 mov    %eax,0x2016b3(%rip)
```

Thread 1		Thread 2		
pc	eax	pc	eax	counter (in memory,shared)
1	1000			1000
	– – – – – –Context switch– – – – – –			
		1	1000	1000
		2	1001	1000
		3	1001	1001
	– – – – – –Context switch– – – – – –			
2	1001			1001
3	1001			1001

```
1 mov    0x2016bc(%rip),%eax
2 add    $0x1,%eax
3 mov    %eax,0x2016b3(%rip)
```

Thread 1		Thread 2		
pc	eax	pc	eax	counter (in memory,shared)
1	1000			1000
	– – – – – –Context switch– – – – – ––			
		1	1000	1000
		2	1001	1000
		3	1001	1001
	– – – – – –Context switch– – – – – ––			
2	1001			1001
3	1001			1001

Lost an update

```
1   mov     0x2016bc(%rip),%eax
2   add     $0x1,%eax
3   mov     %eax,0x2016b3(%rip)
```

Problem: arbitrary interleaving of thrreads

Thread 1 pc	Thread 1 eax	Thread 2 pc	Thread 2 eax	counter (in memory,shared)
1	1000			1000
		Context switch		
		1	1000	1000
		2	1001	1000
		3	1001	1001
		Context switch		
2	1001			1001
3	1001			1001

Lost an update

Race Condition (or Data Race)

A race condition is when the behavior of a program depends on the interleaving of operations of different threads.

In effect, the threads run a race between their operations, and the results of the program execution depends on **who wins** the race.

Debugging

Race conditions ⇒ non-deterministic bugs.

Depends on CPU schedule!

Passing tests means little.

«Heisenbugs & bohrbugs»

How to program: imagine **scheduler is malicious**.

What do we want?

Want all or none of these instructions to execute. That is, we want them to be **atomic**

```
1 mov     0x2016bc(%rip),%eax
2 add     $0x1,%eax
3 mov     %eax,0x2016b3(%rip)
```

Critical section

We want mutual exclusion for critical sections. That is, if I run, you can't (and vice versa).

سوال ۱:

فرض کنید یک برنامه دو رشته‌ای را اجرا می کنیم که در آن رشته‌ها اعمال مشخص شده زیر را انجام می‌دهند:

Thread 1	Thread 2
x = x + 1;	x = x + 2;

با فرض آنکه مقدار اولیه x برابر با صفر است، مقدار نهایی x چه مقادیری می‌تواند باشد (ذکر همه مقادیر ممکن به همراه توضیح الزامی است)

Locks

Locks

Desired: execute a series of instructions atomically

Challenges:

interrupts on a single processor
multiple threads executing on multiple processors concurrently

Solution: using locks around critical sections

Definition (Lock)

A lock is a synchronization variable that provides mutual exclusion-when one thread holds a lock, no other thread can hold the lock (other threads are excluded.)

Pthread Locks

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2  Pthread_mutex_lock(&lock); // wrapper; exits on
     failure
3  balance = balance + 1;
4  Pthread_mutex_unlock(&lock);
5
```

Evaluating Locks

Mutual exclusion

Fairness

Performance

Building a Lock

Attempt #1: Controlling Interrupts

Disable interrupts for critical sections;

invented for single-processor systems.

```
1 void lock() {
2   DisableInterrupts();
3 }
4 void unlock() {
5 EnableInterrupts();
6 }
```

Pros/cons

Simplicity

Allowing threads to do privileged operation → ...☹

Does not work on multiprocessors

Lost interrupts

inefficient

- Used only in limited cases

  e.g. in some cases, by an OS itself ...

Building a Lock

Attempt #2: Pure software

```
1  struct {
2    int lock; //lock = 0: free, lock = 1: held
3                //(acquired)
4  } mutex;
5
6  void mutex_init (struct mutex *m){
7    m->lock = 0; //free @ first time
8  }
9
10 void mutex_lock (struct mutex *m){
11   while (m->lock == 1)
12     ; //spins
13   m->lock = 1; //acquires it
14 }
15
16 void mutex_unlock (struct mutex *m){
17   m->lock = 0; //frees
18 }
```

```
1  struct {
2    int lock; //lock = 0: free, lock = 1: held (
       acquired)
3  } mutex;
4
5  void mutex_init (struct mutex *m){
6    m->lock = 0; //free @ first time
7  }
8
9  void mutex_lock (struct mutex *m){
10   while (m->lock == 1)
11     ; //spins
12   m->lock = 1; //acquires it
13 }
14
15 void mutex_unlock (struct mutex *m){
16   m->lock = 0; //frees
17 }
```

test

set

A clear data race (no need to even dive into assembly level!)

Thread 1	Thread 2
call lock()	
while (flag == 1)	
– – – – ––Context switch– – – – ––	
	call lock()
	while (flag == 1)
	flag = 1;
– – – – ––Context switch– – – – ––	
flag = 1; // set flag to 1 (too!)	

Failed attempt

Building a Lock

Attempt #3: Pure software (Peterson's Algorithm)

```
1  int flag[2];
2  int turn;
3  void init() {
4    flag[0] = flag[1] = 0; // 1 implies thread want to
5                           //grab lock
6    turn = 0; // whose turn? (thread 0 or 1)
7  }
8  void lock() {
9    flag[self] = 1; // self: thread ID of caller
10   turn = 1 - self; // make it other thread's turn
11   while(flag[1-self] && (turn == 1 - self))
12       ; // spin
13 }
14 void unlock() {
15   flag[self] = 0;
16 }
17
```

Correctness: Proof by *contradiction*.
Assume both threads are inside the critical region. Therefore,

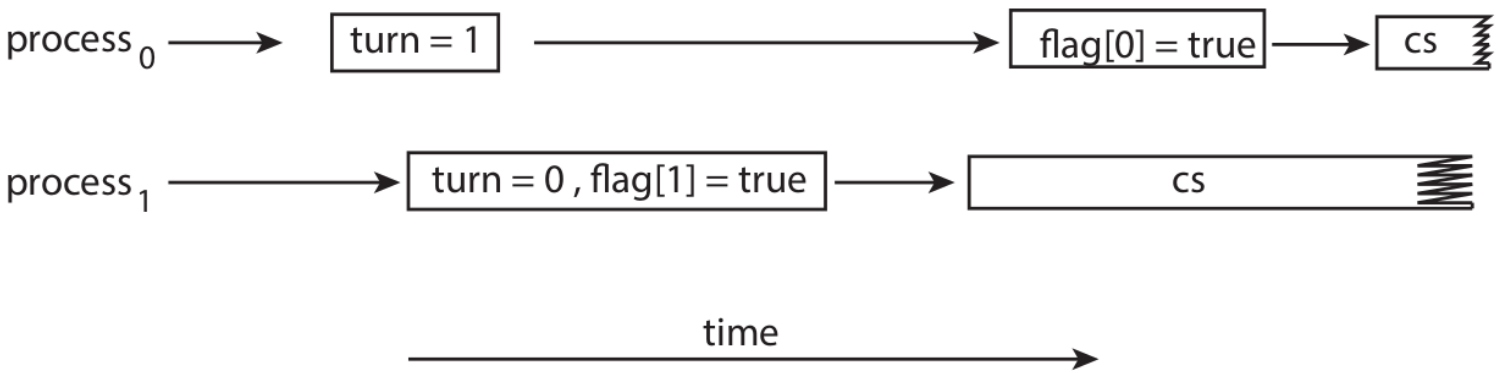
	T0	T1
Assignment	flag[0] $\leftarrow$ 1, turn $\leftarrow$ 1	flag[1] $\leftarrow$ 1, turn $\leftarrow$ 0
Condition	flag[1] = 0 or turn = 0	flag[0] = 0 or turn = 1

$$\begin{cases} \text{turn} = 0\text{: T1's condition} \rightarrow \text{flag[0]}=0 \boxtimes \\ \text{turn} = 1\text{: T0's condition} \rightarrow \text{flag[1]}=0 \boxtimes \end{cases}$$

Contradiction!

Peterson's solution is not guaranteed to work on modern computer architectures

Due to instruction reordering



Remedy: using special hardware supports (memory barriers)

a complex solution

Building a Lock

Attempt #4: Mixed H/W and software

test-and-set instruction (or *atomic exchange*)

Different names: ldstub on SPARC, xchg on X86

```
TestAndSet(int* addr, int newval)
```

returns the old value pointed to by the `addr`,
simultaneously updates said value to `newval`.

Atomic

```
1  typedef struct __lock_t {
2    int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6  // 0: lock is available, 1: lock is held
7    lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12     ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
17 }
```

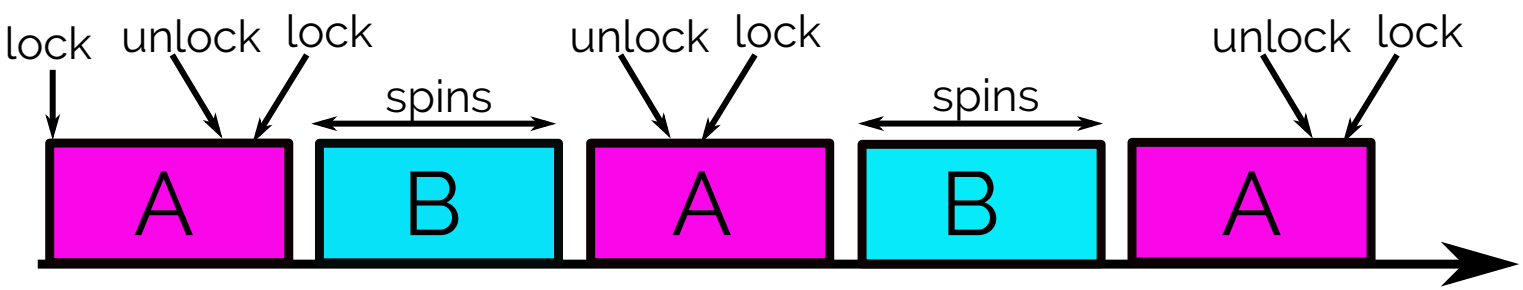
Correctness: Proof by *contradiction*.

1. Assume two threads are in the critical section $\rightarrow$ Both have got 0 from the `TestAndSet` instruction.
2. As `TestAndSet` is atomic, two executions are sequential.
3. But after the first execution the lock is 1 and so the second execution will return 1

   Contradiction!

Starvation Problem

Basic Spinlocks are Unfair



Demo

xv6 usage of `xchg`

Other hardware primitives

compare-and-swap (or *compare-and-exchange*)

load-linked and **store-conditional**

fetch-and-add

Other hardware primitives

compare-and-swap (or *compare-and-exchange*)

load-linked and **store-conditional**

fetch-and-add

Compare-and-Swap

`int CompareAndSwap(int *addr, int expected, int newval)`

- returns the original value pointed by `addr`
- update the memory location pointed to by `addr` with the `newval` if `expected` is equal to the original value

```
1 void lock(lock_t *lock) {
2   while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3     ; // spin
4 }
```

- similar to test-and-set but more powerful.

  "Wait-free Synchronization" by Maurice Herlihy. ACM TOPLAS, 1991.

Demo

Other hardware primitives

compare-and-swap (or *compare-and-exchange*)

load-linked and **store-conditional**

fetch-and-add

Load-Linked and Store-Conditional

```
int LoadLinked(int *addr)

int StoreConditional(int *addr, int value)
```

- `LoadLinked`: loads a value from memory and places it in a register
- `StoreConditional`: only succeeds (and updates the value stored at the address just load-linked from) if **no intervening store** to the address has taken place.
  - on success: updates and returns 1
  - on failure: returns 0

```
1 void lock(lock_t *lock) {
2   while (1) {
3     while (LoadLinked(&lock->flag) == 1)
4       ; // spin until it's zero
5     if (StoreConditional(&lock->flag, 1) == 1)
6       return; // if set-it-to-1 was a
7               //success: all done
8   // otherwise: try it all over again
9   }
10 }
```

Other hardware primitives

compare-and-swap (or *compare-and-exchange*)

load-linked and **store-conditional**

fetch-and-add

fetch-and-add

```
int FetchAndAdd(int *addr)
```

- atomically increments a value while returning the old value at a particular address.

Ticket Lock

```
1  typedef struct __lock_t {
2  int ticket;
3  int turn;
4  }
5  lock_t;
6
7  void lock_init(lock_t *lock) {
8  lock->ticket = 0;
9  lock->turn   = 0;
10 }
11
12 void lock(lock_t* lock) {
13 int myticket = FetchAndAdd(&lock->ticket);
14 while (lock->turn != myticket)
15 ; // spin
16 }
17
18 void unlock(lock_t *lock) {
19 lock->turn = lock->turn + 1;
20 }
21
```

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0
– – – – – – Context switch – – – – – –			
	calls lock()	1	0
	FAA: myticket ← 1	2	0
	myticket≠turn spins	2	0

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0
– – – – – –Context switch– – – – – –			
	calls lock()	1	0
	FAA: myticket ← 1	2	0
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
myticket=turn **holds lock**, <cs>		2	0

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0
– – – – – –Context switch– – – – – –			
	calls lock()	1	0
	FAA: myticket ← 1	2	0
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
myticket=turn holds lock, <cs>		2	0
– – – – – –Context switch– – – – – –			
	myticket≠turn spins	2	0

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0
– – – – – –Context switch– – – – – –			
	calls lock()	1	0
	FAA: myticket ← 1	2	0
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
myticket=turn holds lock, <cs>		2	0
– – – – – –Context switch– – – – – –			
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
<cs>, releases (calls unlock())		2	1

Thread 1	Thread 2	ticket	turn
calls lock()		0	0
FAA: myticket ← 0		1	0
– – – – – –Context switch– – – – – –			
	calls lock()	1	0
	FAA: myticket ← 1	2	0
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
myticket=turn holds lock, <cs>		2	0
– – – – – –Context switch– – – – – –			
	myticket≠turn spins	2	0
– – – – – –Context switch– – – – – –			
<cs>, releases (calls unlock())		2	1
– – – – – –Context switch– – – – – –			
	myticket=turn **holds lock**, <cs>	2	1

No starvation

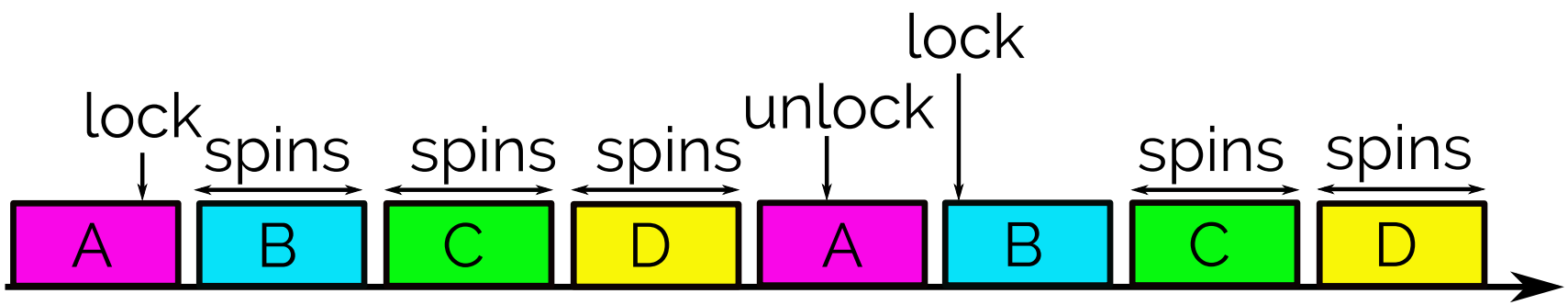
- ensures progress for all threads.
- Once a thread is assigned its ticket value, it will be scheduled at some point in the future
- In our previous attempts, no such guarantee existed

Stop (or spin ☺) & think:

Does integer overflow make a correctness problem?
(*TBD in our online session*)

Stop Spinning

CPU Wastage



CPU wastage (spinning): (#threads−1) × time_slice

Stop Spinning

we need OS support in form of some system calls

yield()

park()/unpark()

Stop Spinning

we need OS support in form of some system calls

yield()

park()/unpark()

yield

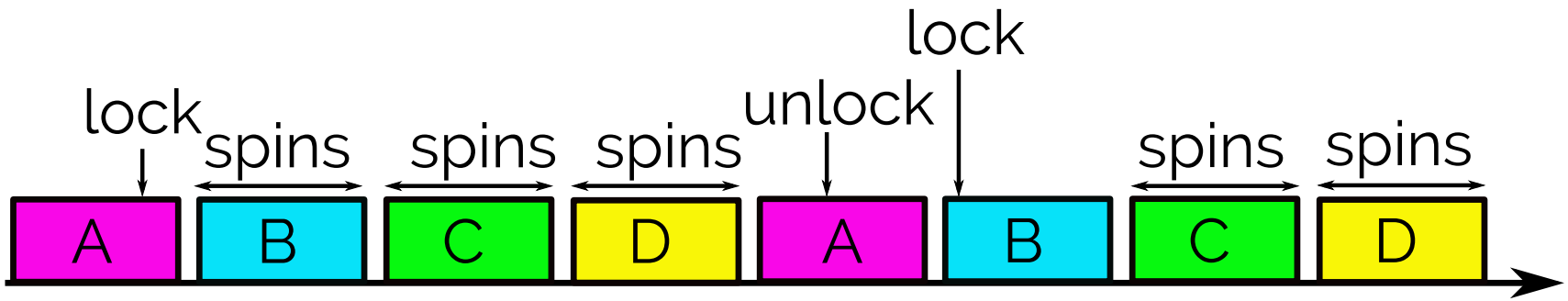
yield: **Running state** $\rightarrow$ **Runnable state**

Caller **deschedules itself**.

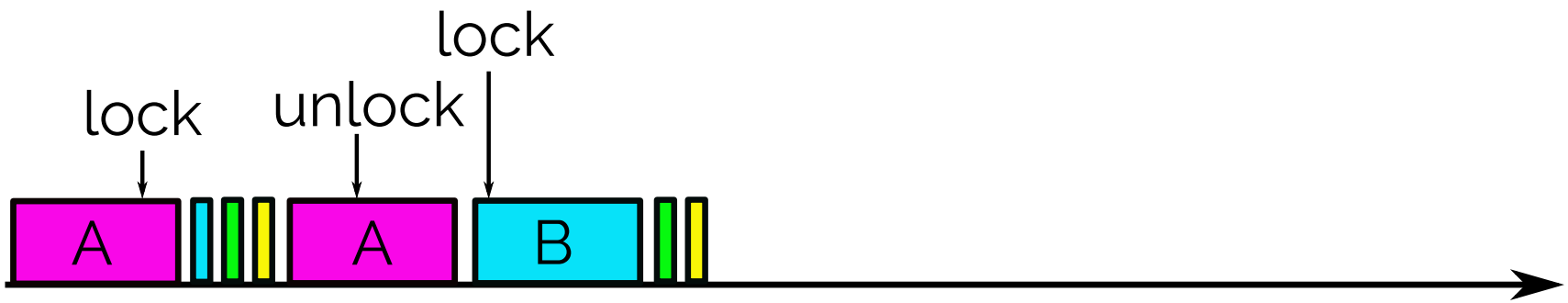
```
1 void init() {
2   flag = 0;
3 }
4
5 void lock() {
6   while (TestAndSet(&flag, 1) == 1)
7     yield(); // give up the CPU
8 }
9
10 void unlock() {
11   flag = 0;
12 }
```

CPU Wastage

no yield:



with yield:



CPU wastage (with yield): (#threads−1) × context switch

Stop Spinning

we need OS support in form of some system calls

yield()

park()/unpark()

park/unpark

Introduced in Solaris

park(): **Running state** $\rightarrow$ **Blocked state sleep**

unpark(threadID): **wakes (makes runnable)** a particular thread as designated by threadID.

```
1 typedef struct __lock_t {
2   int flag;
3   int guard;
4   queue_t *q;
5 } lock_t;
6
7 void lock_init(lock_t *m) {
8   m->flag = 0;
9   m->guard = 0;
10  queue_init(m->q);
11 }
12 void lock(lock_t *m) {
13  while (TestAndSet(&m->guard, 1) == 1)
14    ; //acquire guard lock by spinning
15  if (m->flag == 0) {
16    m->flag = 1; // lock is acquired
17    m->guard = 0;
18  } else {
19    queue_add(m->q, gettid());
20    m->guard = 0;
21    park();
22  }
23 }
24
```

```
1  void unlock(lock_t *m) {
2    while (TestAndSet(&m->guard, 1) == 1)
3      ; //acquire guard lock by spinning
4    if (queue_empty(m->q))
5      m->flag = 0; // let go of lock; no one wants it
6    else
7      unpark(queue_remove(m->q)); // hold lock
8                                  // (for next thread!)
9    m->guard = 0;
10 }
11
```

```
1  void lock(lock_t *m) {
2    while (TestAndSet(&m->guard, 1) == 1)
3      ; //acquire guard lock by spinning
4    if (m->flag == 0) {
5      m->flag = 1; // lock is acquired
6      m->guard = 0;
7    } else {
8      queue_add(m->q, gettid());
9      m->guard = 0;
10     park();
11   }
12 }
```

race condition

```
1  void lock(lock_t *m) {
2    while (TestAndSet(&m->guard, 1) == 1)
3      ; //acquire guard lock by spinning
4    if (m->flag == 0) {
5      m->flag = 1; // lock is acquired
6      m->guard = 0;
7    } else {
8      queue_add(m->q, gettid());
9      setpark(); // about to park
10     m->guard = 0;
11     park();
12   }
13 }
```

using setpark() to avoid race

Different OS, Different Support

futex

```
man futex, man 7 futex
```

Summery on Building Locks

Using H/W supports for mutual exclusive access to shared data structures

Mainly instructions to **atomically** read and then write a single memory location

Multi-cores guarantee (in hardware) **serialization** of simultaneous such atomic instructions

A lock is needed for critical sections of more than one instruction.

- Atomic instructions are used to arbitrate among simultaneous attempts to acquire the lock
- When a lock is busy: 1)spin-wait, 2)relinquishing CPU, 3) block, or 4) **hybrid**

We will return to locks after introducing *semaphores*!

سوال ۲:

فرض کنید دو نفر آشپز A و B برای پختن یک آش با همدیگر همکاری می‌کنند. هر یک از آشپزها موقعی که به دیگ سر می‌زند آش را می‌چشد و اگر بی‌نمک بود، به آن نمک اضافه می‌کند. هدف آن است که رویه انجام کار به نحوی بین دو آشپز تعیین شود که آش نه شور شود و نه بی نمک.

فرض کنید هر دو آشپز نامرئی هستند و همدیگر را نمی‌بینند. همچنین نمک داشتن و یا نداشتن آش توسط متغیر salt مشخص می‌شود (اگر salt برابر با صفر باشد آش نمک ندارد و اگر برابر با یک باشد آش نمک دارد). همچنین عملیات اضافه کردن نمک با اضافه کردن یک واحد به salt انجام می‌شود. در ضمن آشپزها می‌توانند حضور و یا عدم حضور خود را با یادداشت گذاری و یادداشت برداری به یکدیگر اعلام نمایند (متعیرهای باینری noteA و noteB به ترتیب توسط آشپزهای A و B استفاده می‌شوند).

آیا رویه زیر می‌تواند روشی کارآمد برای پختن آشی خوش نمک باشد؟ توضیح دهید.

```
1  //CHEF A
2  noteA = 1;          // leave note
3  if (noteB==0) {     // if no note
4    if (salt==0) { // if no salt
5    salt++;     // add salt
6    }
7  }
8  noteA = 0;          // remove note A
9
```

```
1  //CHEF B
2  noteB = 1;          // leave note
3  if (noteA==0) {     // if no note
4    if (salt==0) { // if no salt
5      salt++;     // add salt
6    }
7  }
8  noteB = 0;          // remove note A
9
```

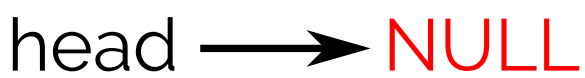
Using locks

Using Locks

What about problems more complex than "counter"?

Linked-List Race

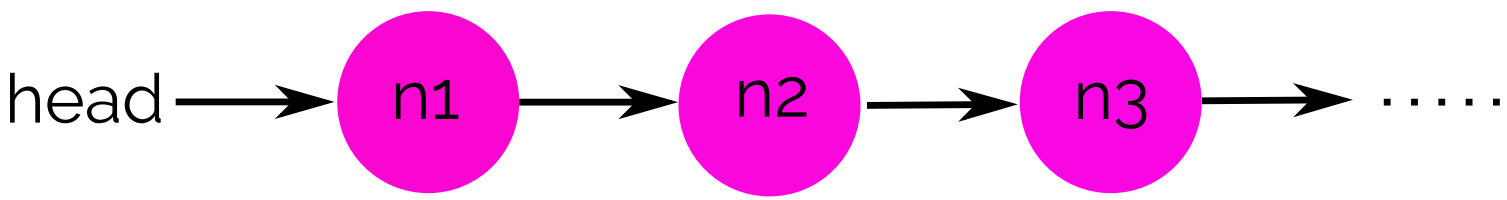
```
1  // basic node structure
2  typedef struct __node_t {
3    int                 key;
4    struct __node_t *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9    node_t *head;
10 } list_t;
11
12 void List_Init(list_t *L) {
13   L->head = NULL;
14 }
```



focusing on insert

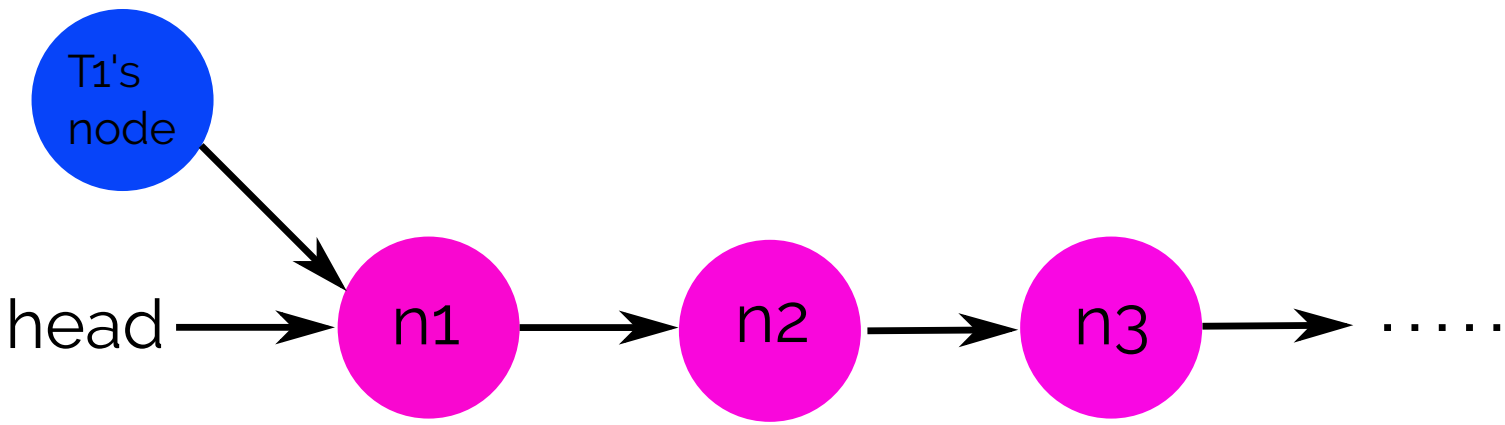
```
1  int List_Insert(list_t*L, int key) {
2    node_t*new = malloc(sizeof(node_t));
3    if (new == NULL) {
4      perror("malloc");
5      return -1; // fail
6    }
7    new->key  = key;
8    new->next = L->head;
9    L->head   = new;
10   return 0; // success
11 }
12
```

Without data race:

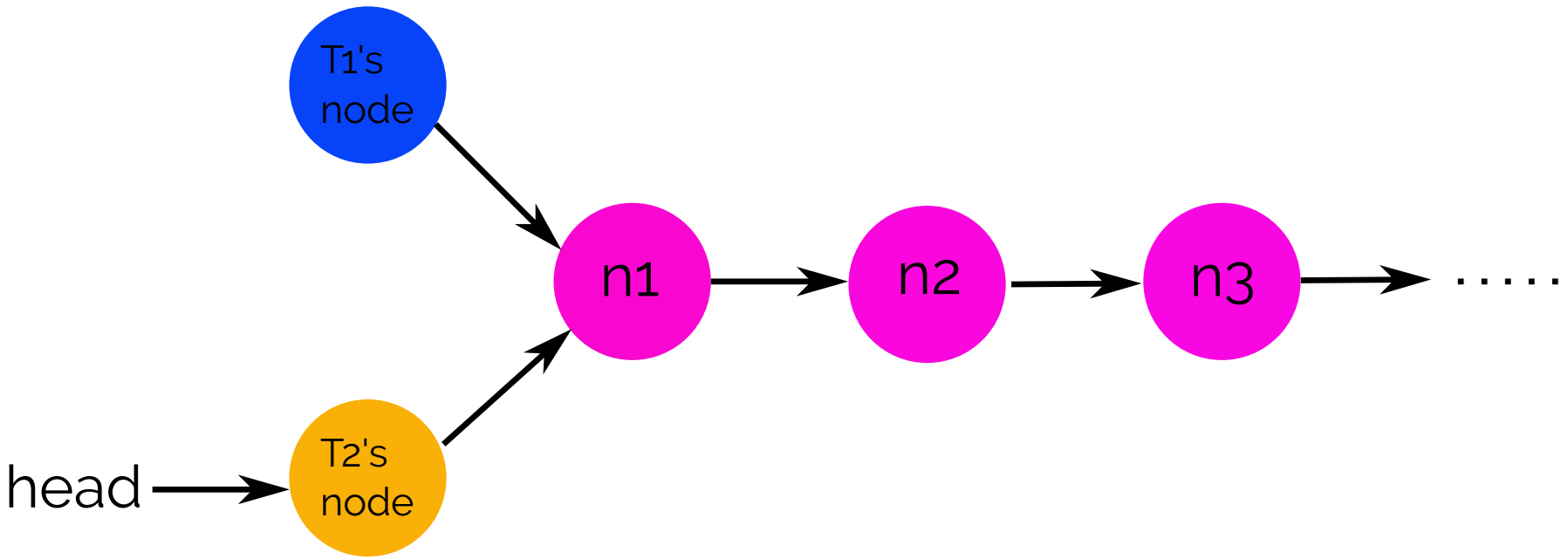


Is there any race condition?

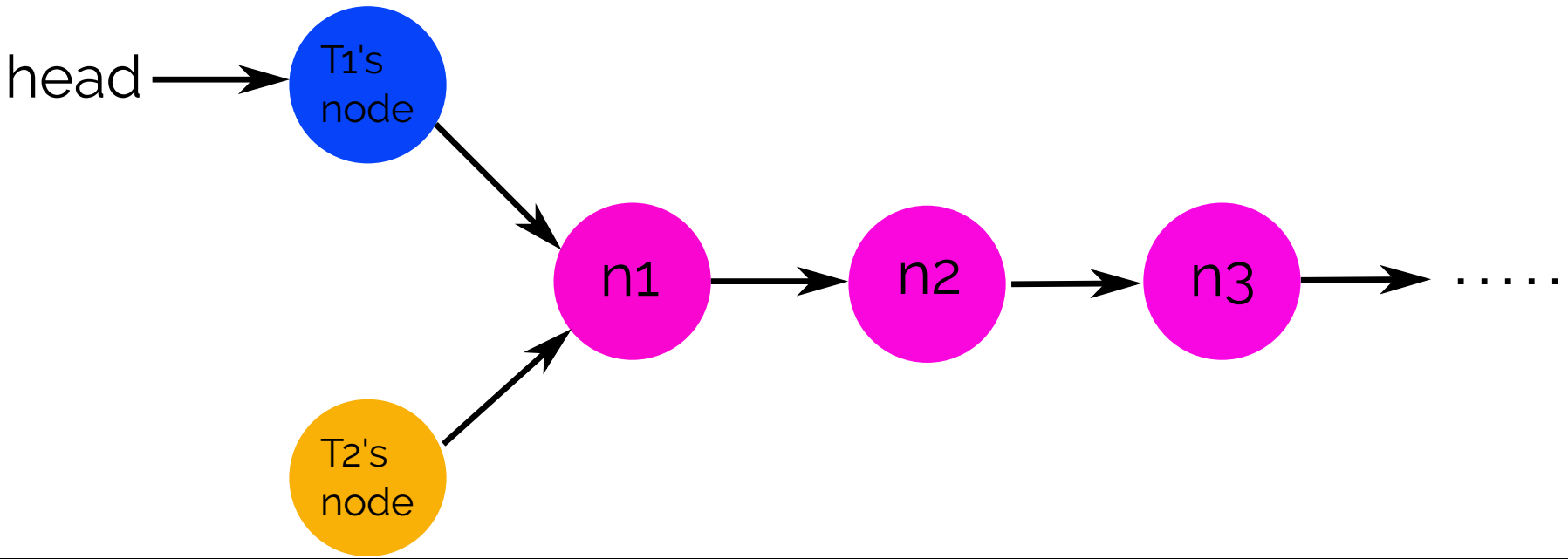
Thread 1	Thread 2
new->key = key	
new->next = L->head	



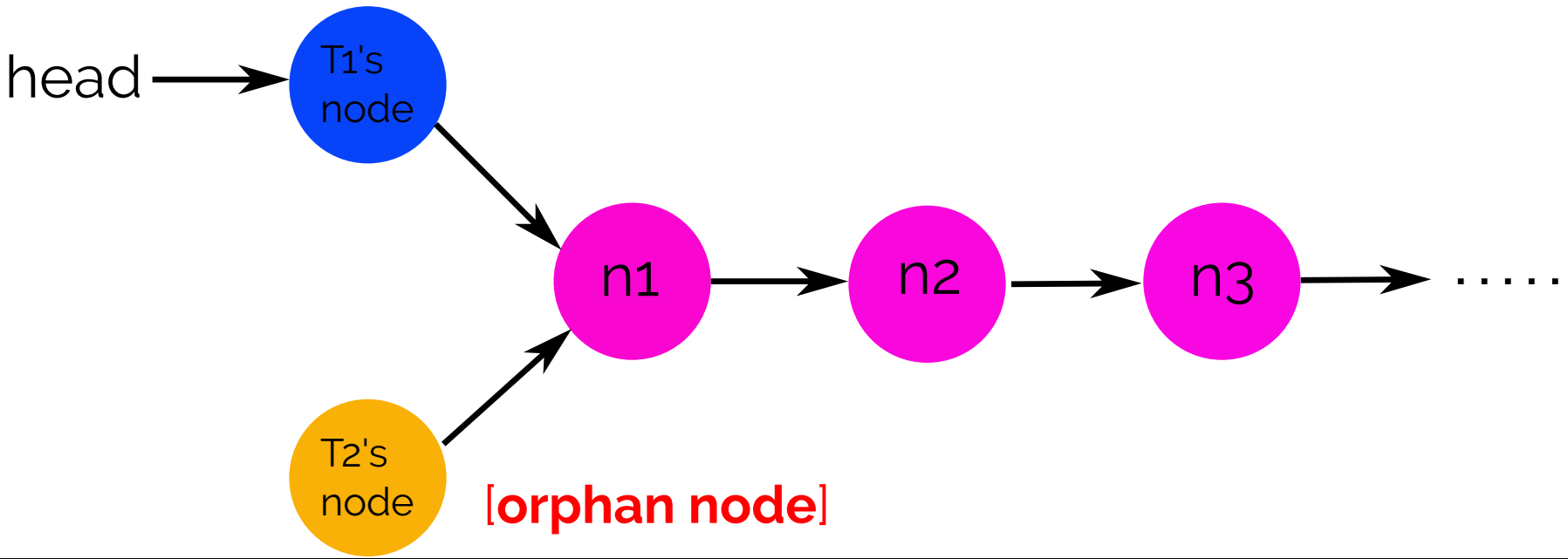
Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new



Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	



Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	



Is there any race condition? Yes

It is **not** thread-safe

Solution?

Using locks

```
1  // basic node structure
2  typedef struct __node_t {
3  int                 key;
4  struct __node_t *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9  node_t *head;
10 pthread_mutex_t      lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14 L->head = NULL;
15 pthread_mutex_init(&L->lock, NULL);
16 }
17
```

Method 1

```
1  int List_Insert(list_t*L, int key) {
2  pthread_mutex_lock(&L->lock);
3  node_t*new = malloc(sizeof(node_t));
4  if (new == NULL) {
5  perror("malloc");
6  pthread_mutex_unlock(&L->lock);
7  return -1; // fail
8  }
9  new->key  = key;
10 new->next = L->head;
11 L->head   = new;
12 pthread_mutex_unlock(&L->lock);
13 return 0; // success
14 }
15
```

Method2: a better usage of locks

```
1   int List_Insert(list_t*L, int key) {
2   node_t*new = malloc(sizeof(node_t));
3   if (new == NULL) {
4   perror("malloc");
5   return -1; // fail
6   }
7   new->key  = key;
8   pthread_mutex_lock(&L->lock);
9   new->next = L->head;
10  L->head   = new;
11  pthread_mutex_unlock(&L->lock);
12  return 0; // success
13  }
14
```

Makes critical section smaller → Better performance

Discussion Question

What is the other benefit?

Exercise: what about delete and lookup?

A single lock problem

acquired when calling a routine that manipulates the data structure, and is released when returning from the call

similar to a data structure built with **monitors [Deprecated]**

if the data structure is not too slow, you are done!

But if it is too slow?

How to speed up?

One lock per node

Overheads of acquiring and releasing locks for each node of a list traversal is prohibitive

what about hybrid? (a new lock every so many nodes)

Condition Variables

Condition Variables

Locks: used to prevent data race

Condition variables (CV): Waiting for a change

used when a thread wishes to wait for something else to change the **state** of the system so that it can make progress.

Examples:

- Web server
- Word processor
- ...

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // declare/init a lock
pthread_cond_t c = PTHREAD_COND_INITIALIZER; // declare/init a CV
```

A **condition variable** (CV) is a synchronization object that lets a thread efficiently wait for a change to shared state that is protected by a lock.

`wait` `(cond_t *cv, mutex_t *lock)`

- atomically releases the lock and suspends execution of the calling thread, placing the calling thread on the condition variable;s waiting list. Later, when the calling thread is re-enabled, it re-acquires the lock before returning from the wait call.

`signal` `(cond_t *cv)`

- wakes a single waiting thread (if ≥ 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

`broadcast` `(cond_t *cv)`

- wakes all threads (if ≥ 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

A CV is usually **PAIRED** with some kind *state variable* - e.g., an integer (which indicates the state of the program)

```
int done = 0; // example of related "state" variable
```

Classic problems

Classic Problem I
«Join»

```
1 void *child(void *arg) {
2 printf("child\n");
3 // XXX how to indicate we are done?
4 return NULL;
5 }
6 int main(int argc, char *argv[]) {
7 pthread_t p;
8 printf("parent: begin\n");
9 Pthread_create(&p, 0, child, 0);
10 // XXX how to wait for child?
11 printf("parent: end\n");
12 return 0;
13 }
```

Expected output:

```
1 parent: begin
2 child
3 parent: end
```

Solution 1: Spinning Approach

```
1 void *child(void *arg) {
2   printf("child\n");
3   done = 1;
4   return NULL;
5 }
6 int main(int argc, char *argv[]) {
7   pthread_t p;
8   printf("parent: begin\n");
9   Pthread_create(&p, 0, child, 0);
10  while (done == 0)
11    ; // spin (inefficient)
12  printf("parent: end\n");
13  return 0;
14 }
```

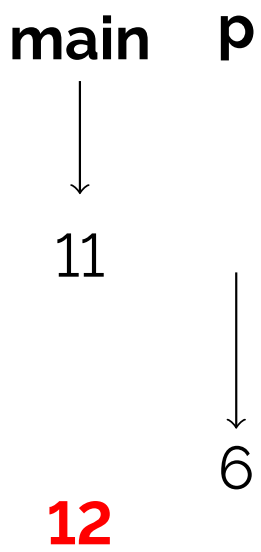
wastes CPU time!

Solution 2: No Lock

```
1 void *child(void *arg) {
2   printf("child\n");
3   done = 1;
4   Pthread_cond_signal(&c);
5   return NULL;
6 }
7 int main(int argc, char *argv[]) {
8   pthread_t p;
9   printf("parent: begin\n");
10  Pthread_create(&p, 0, child, 0);
11  while (done == 0) {
12    Pthread_cond_wait(&c, &m);
13  }
14  printf("parent: end\n");
15  return 0;
16 }
```

Solution 2: No Lock

```
1 void *child(void *arg) {
2   printf("child\n");
3   done = 1;
4   Pthread_cond_signal(&c);
5   return NULL;
6 }
7 int main(int argc, char *argv[]) {
8   pthread_t p;
9   printf("parent: begin\n");
10  Pthread_create(&p, 0, child, 0);
11  while (done == 0) {
12    Pthread_cond_wait(&c, &m);
13  }
14  printf("parent: end\n");
15  return 0;
16 }
```



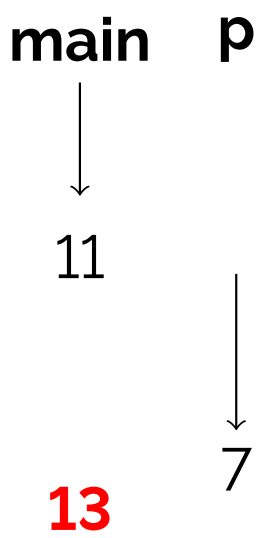
Race! (sleeping forever)

Solution 3: No State Variable

```
1 void *child(void *arg) {
2   printf("child\n");
3   Pthread_mutex_lock(&m);
4   Pthread_cond_signal(&c);
5   Pthread_mutex_unlock(&m);
6   return NULL;
7 }
8 int main(int argc, char *argv[]) {
9   pthread_t p;
10  printf("parent: begin\n");
11  Pthread_create(&p, 0, child, 0);
12  Pthread_mutex_lock(&m);
13  Pthread_cond_wait(&c, &m);
14  Pthread_mutex_unlock(&m);
15  printf("parent: end\n");
16  return 0;
17 }
```

Solution 3: No State Variable

```
1 void *child(void *arg) {
2   printf("child\n");
3   Pthread_mutex_lock(&m);
4   Pthread_cond_signal(&c);
5   Pthread_mutex_unlock(&m);
6   return NULL;
7 }
8 int main(int argc, char *argv[]) {
9   pthread_t p;
10  printf("parent: begin\n");
11  Pthread_create(&p, 0, child, 0);
12  Pthread_mutex_lock(&m);
13  Pthread_cond_wait(&c, &m);
14  Pthread_mutex_unlock(&m);
15  printf("parent: end\n");
16  return 0;
17 }
18
```



Sleeping forever

Solution 4: Actually Works

```
1 void *child(void *arg) {
2   printf("child\n");
3   Pthread_mutex_lock(&m);
4   done = 1;
5   Pthread_cond_signal(&c);
6   Pthread_mutex_unlock(&m);
7   return NULL;
8 }
9 int main(int argc, char *argv[]) {
10  pthread_t p;
11  printf("parent: begin\n");
12  Pthread_create(&p, 0, child, 0);
13  Mutex_lock(&m);
14  while (done == 0)
15    Cond_wait(&c, &m);
16  Mutex_unlock(&m);
17  printf("parent: end\n");
18  return 0;
19 }
```

Tip: Always use `while` even though `if` also works sometimes (e.g., here)!

Class Question

Does changing the order of lines 4 and 5 make any problem?

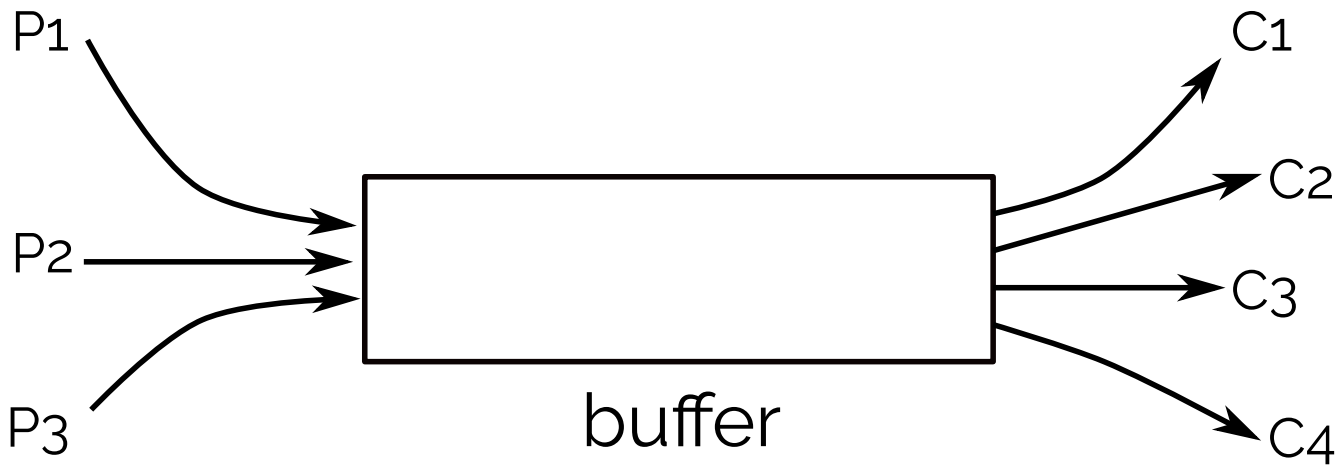
Condition Variable: **Important Points**

memoryless

wait atomically releases the lock

When a waiting thread is re-enabled via signal, it may not run immediately $\rightarrow$ *wait must always be called from within a loop*

Classic Problem II
«Producer-Consumer»



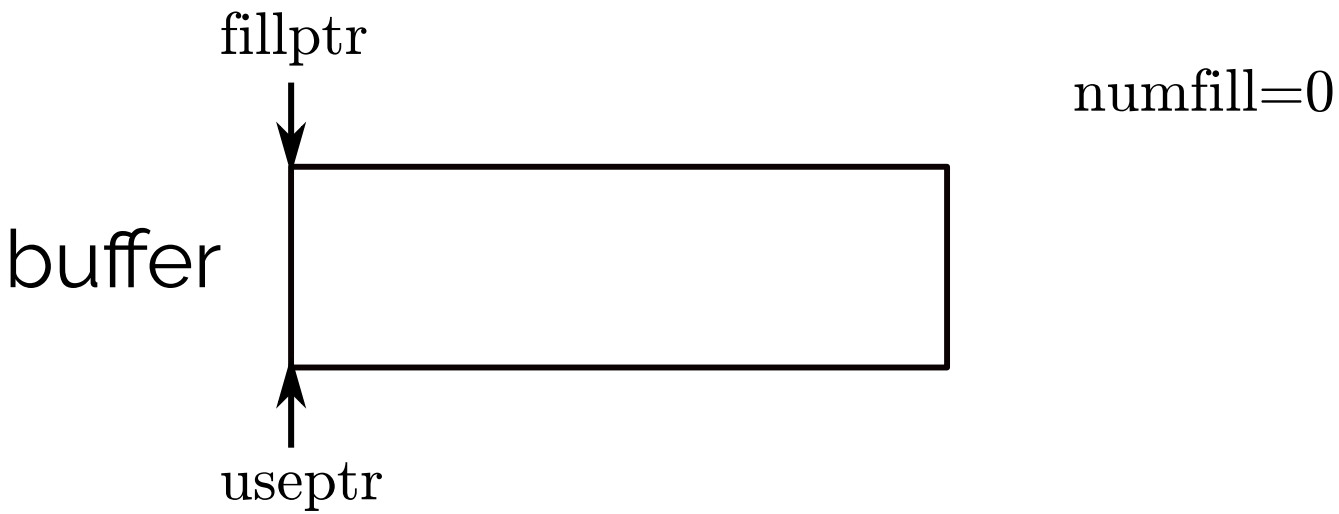
Introduced by **Dijkstra**

led to invention of *semaphores* (more later ...)

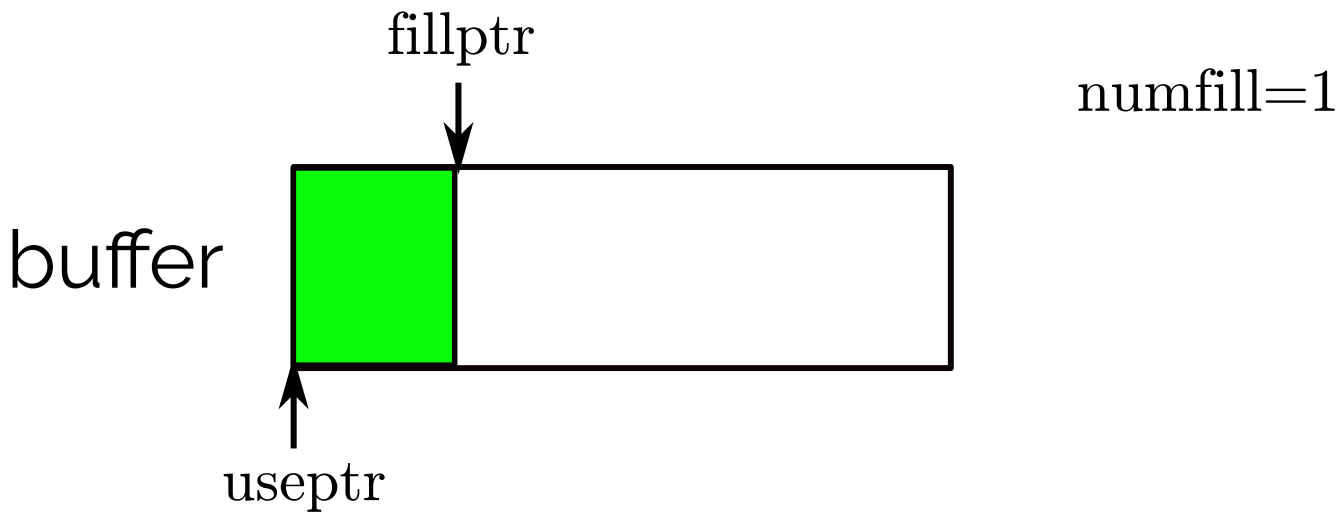
Occurs in many real systems

web servers

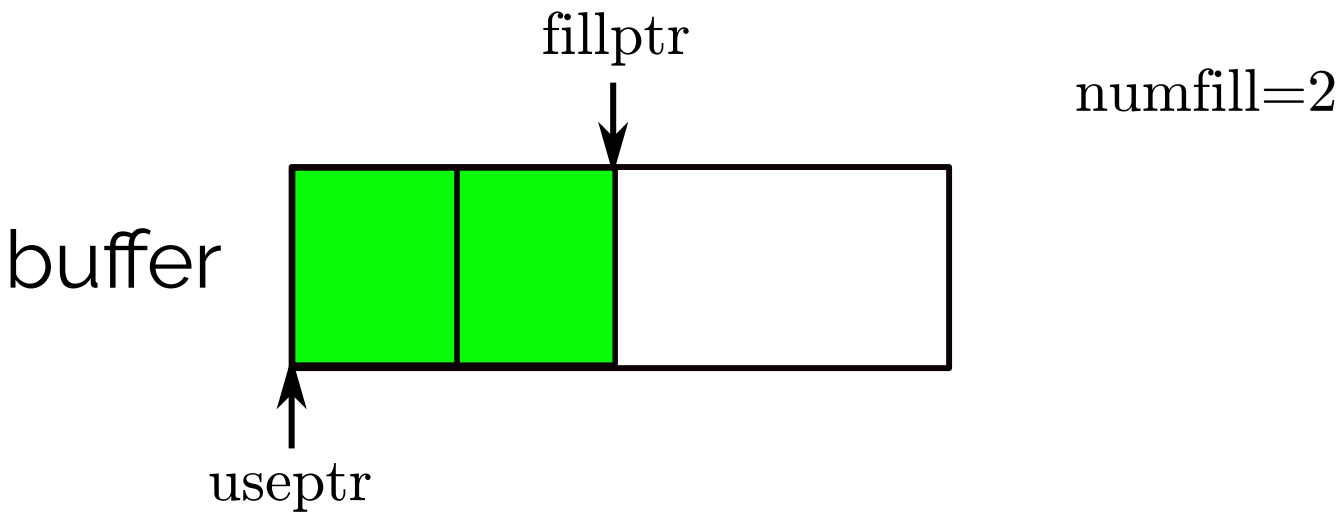
Pipe: `grep foo file.txt | wc -l`


fillptr
numfill=0
buffer
useptr

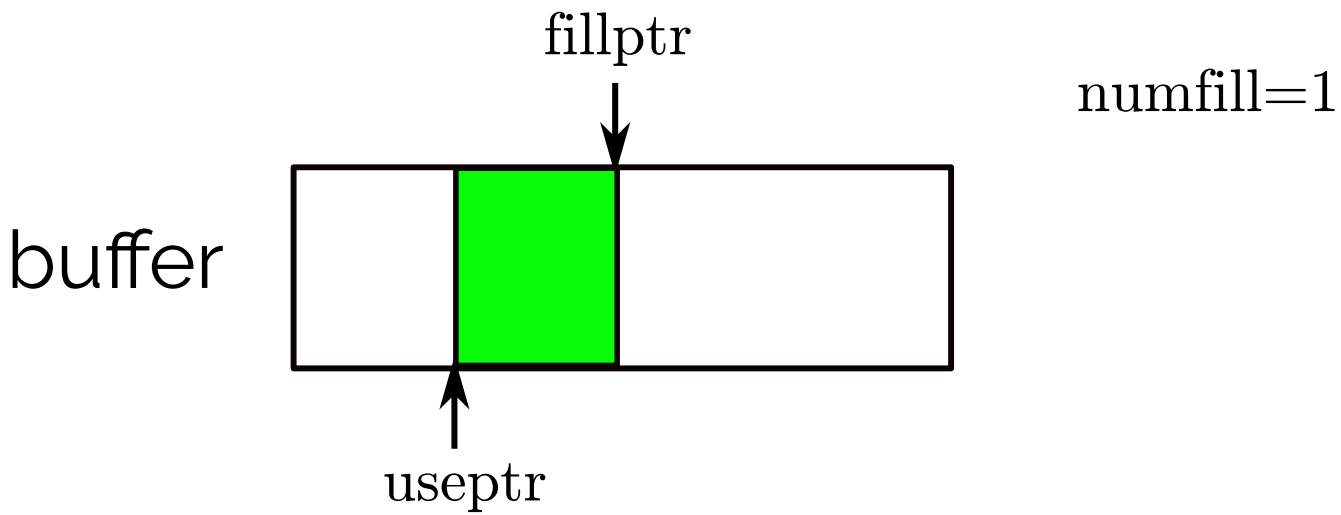
FILL



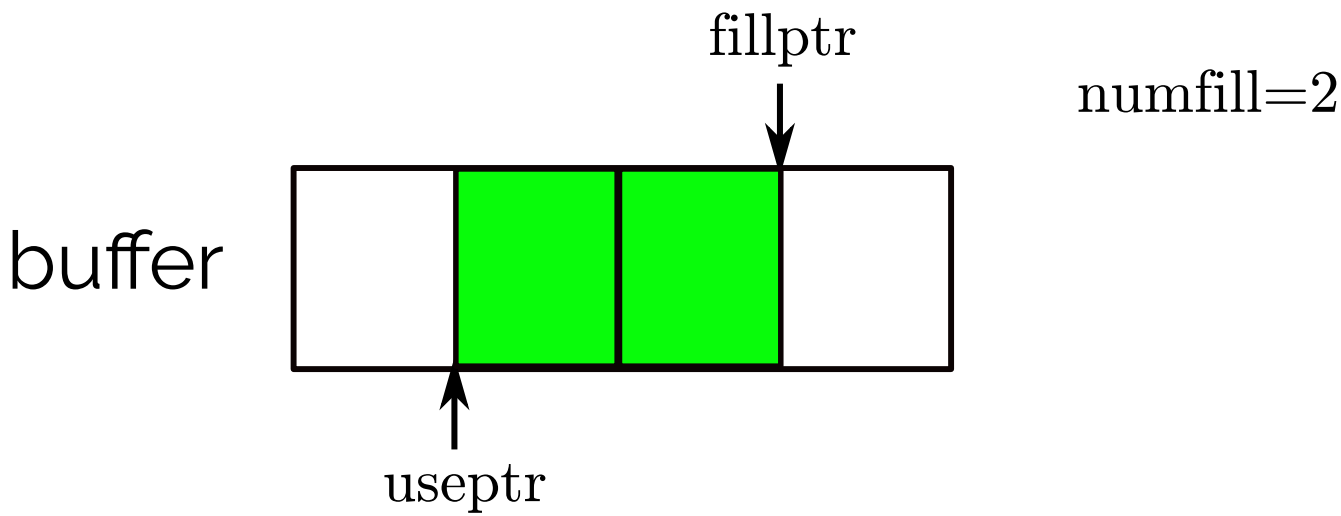
FILL



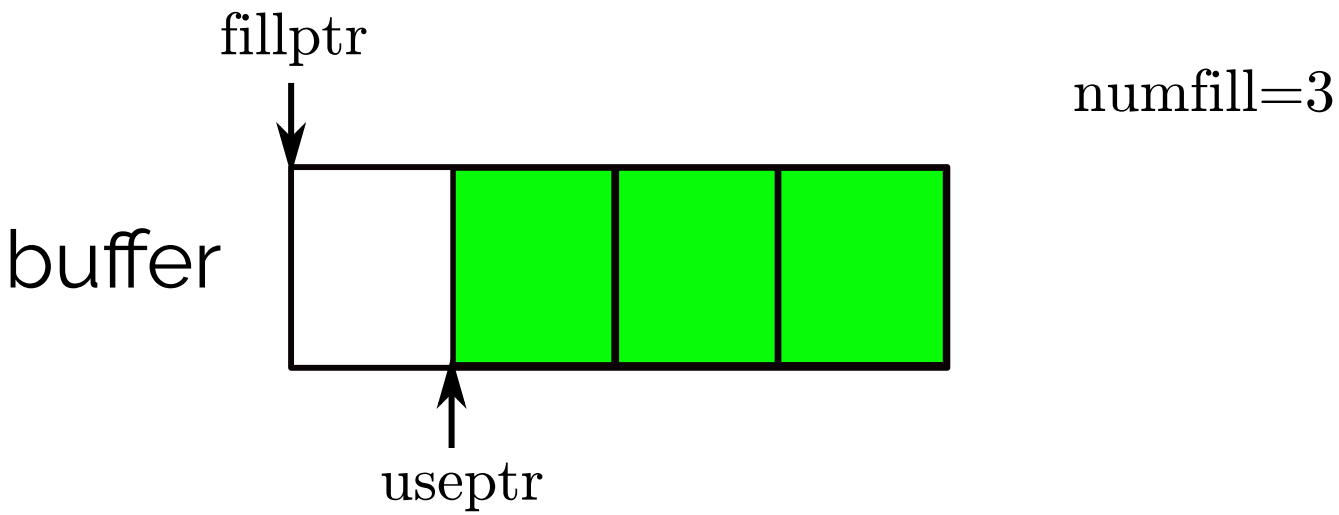
GET



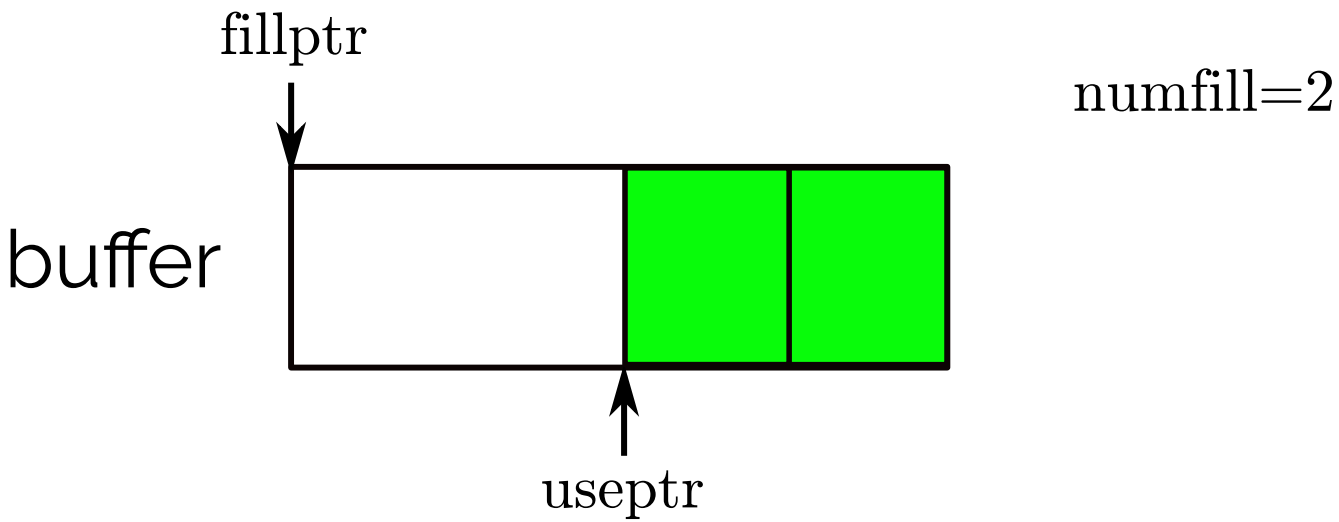
FILL



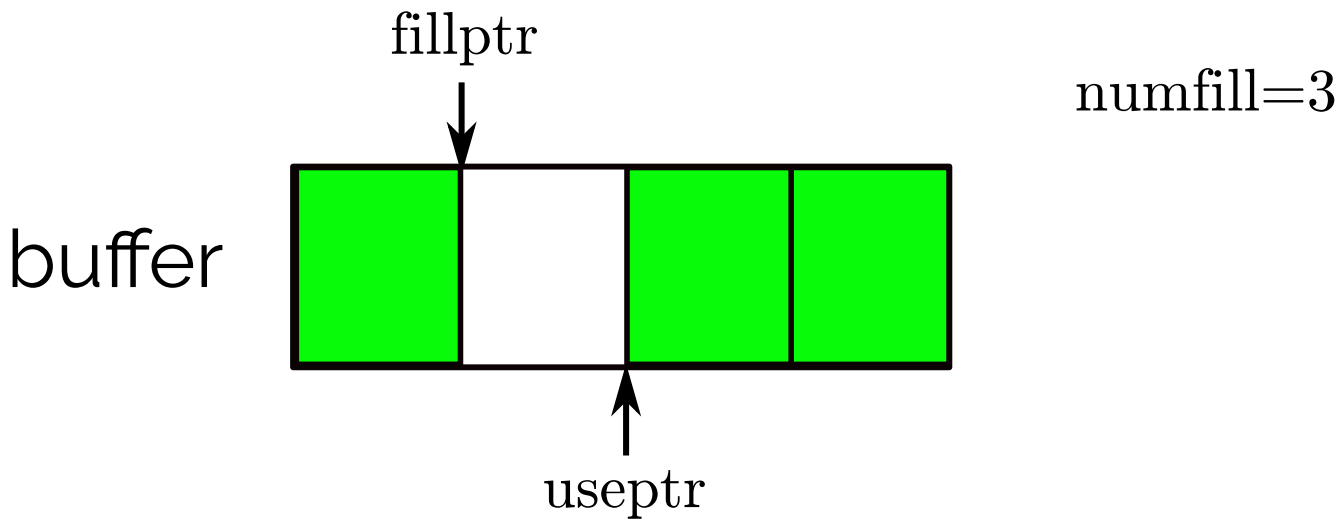
FILL

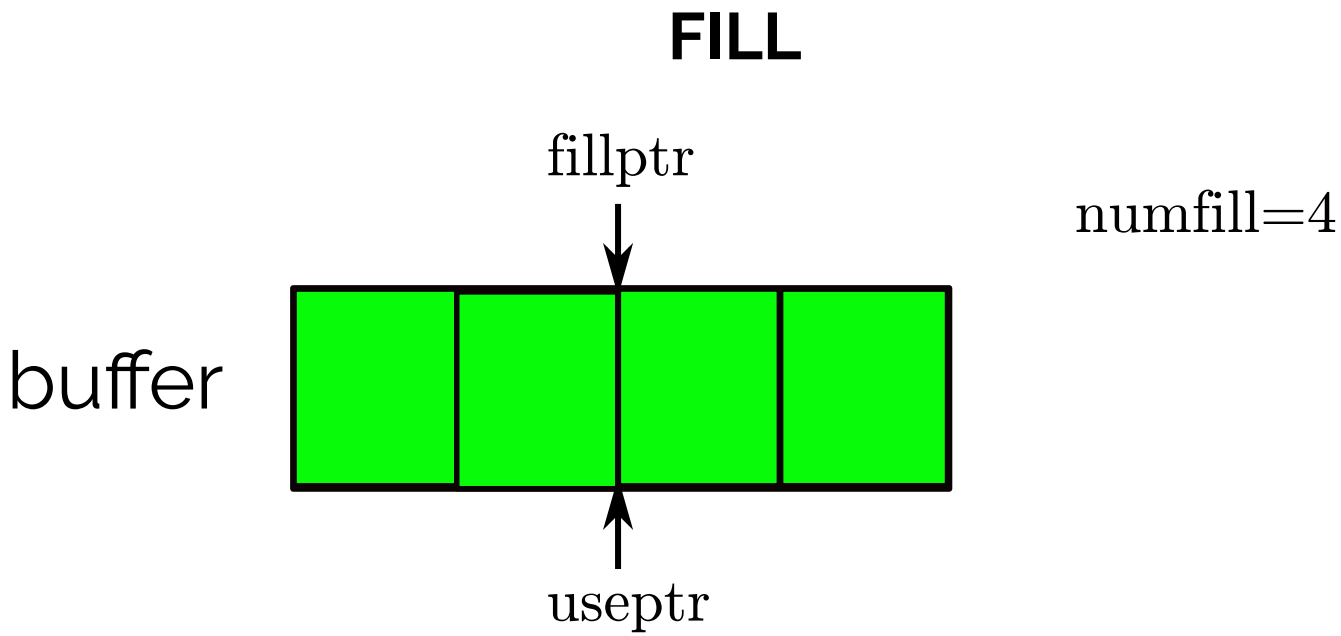


GET

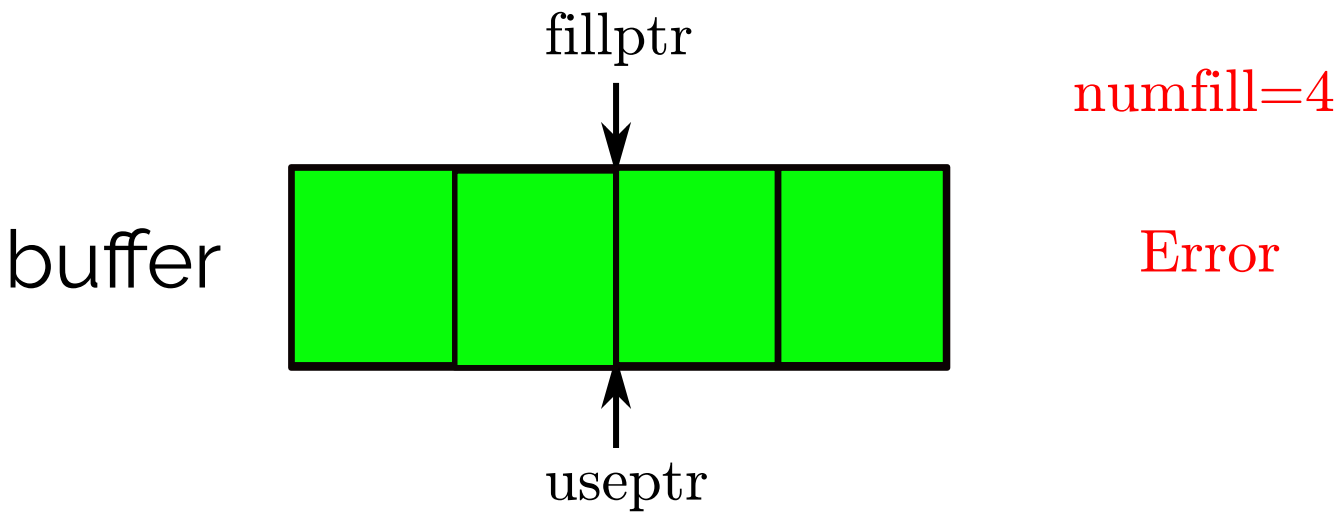


FILL




FILL
fillptr
numfill=4
buffer
useptr

FILL



MAIN PROGRAM

```
1 int main(int argc, char *argv[]) {
2   max = atoi(argv[1]);
3   loops = atoi(argv[2]);
4   consumers = atoi(argv[3]);
5   buffer = (int *) Malloc(max * sizeof(int));
6   pthread_t pid, cid[CMAX];
7   Pthread_create(&pid, NULL, producer, NULL);
8   for (int i = 0; i < consumers; i++)
9     Pthread_create(&cid[i], NULL, consumer, NULL);
10  Pthread_join(pid, NULL);
11  for (i = 0; i < consumers; i++)
12    Pthread_join(cid[i], NULL);
13 }
```

QUEUE GET/PUT

```
1 void do_fill(int value) {
2   assert(numfill < max);
3   buffer[fillptr] = value;
4   fillptr = (fillptr + 1) % max;
5   numfull++;
6 }
7 int do_get() {
8   assert(numfill > 0);
9   int tmp = buffer[useptr];
10  useptr = (useptr + 1) % max;
11  numfull--;
12  return tmp;
13 }
```

Solution v1 (Single CV)

```
1 void *producer(void *arg) {
2   for (int i = 0; i < loops; i++) {
3     Mutex_lock(&m);
4     while (numfull == max)
5       Cond_wait(&cond, &m);
6     do_fill(i);
7     Cond_signal(&cond);
8     Mutex_unlock(&m);
9   }
10 }
11 void *consumer(void *arg) {
12   for (i = 0; i < loops; i++) {
13     Mutex_lock(&m);
14     while (numfull == 0)
15       Cond_wait(&cond, &m);
16     int tmp = do_get();
17     Cond_signal(&cond);
18     Mutex_unlock(&m);
19     printf("%d\n", tmp);
20   }
21 }
```

Stop & think:

Find the problem!

Big Bang

Runnable

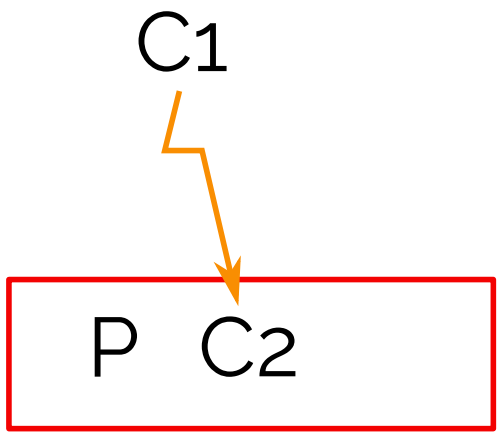
Running C1

CV P C2

Runnable

Running

CV



Runnable C2

Running C1

CV P

Runnable C2

Running C1 --> wait

CV P

Runnable

Running C2

CV P C1

Runnable

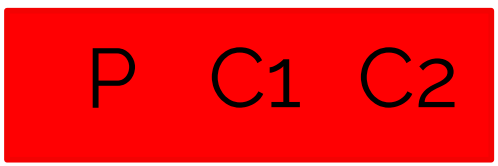
Running C2 --> wait

CV P C1

Runnable

Running

CV



Sleeping forever ...

Big Bang

Runnable

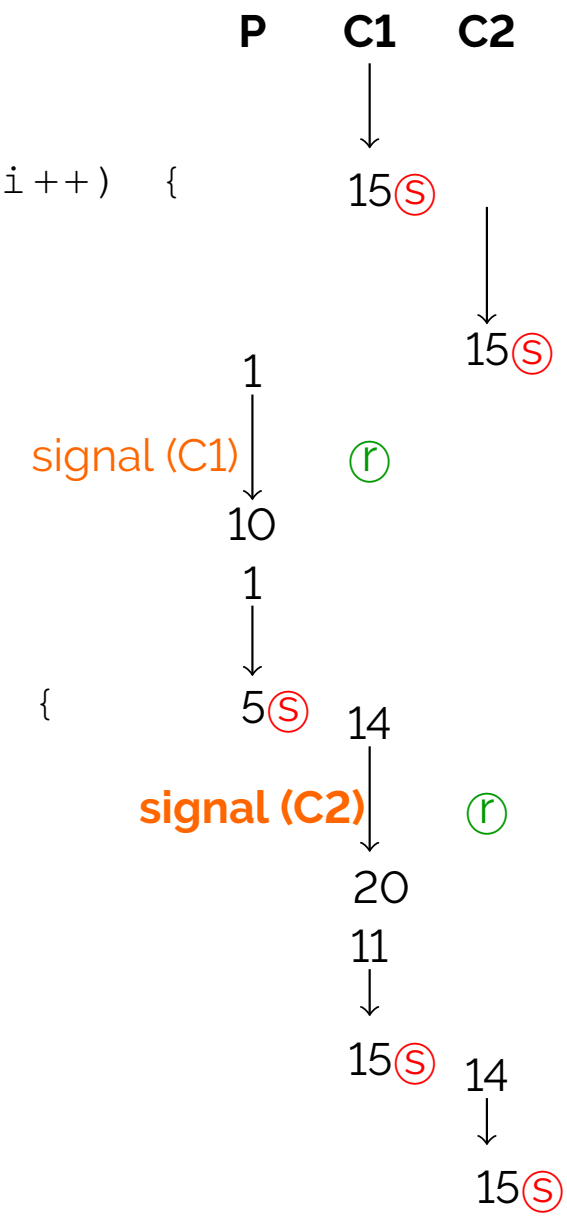
Running C1

CV P C2

But how to create one?

Solution v1 (Single CV)

```
1  void *producer(void *arg) {
2    for (int i = 0; i < loops; i++) {
3      Mutex_lock(&m);
4      while (numfull == max)
5        Cond_wait(&cond, &m);
6      do_fill(i);
7      Cond_signal(&cond);
8      Mutex_unlock(&m);
9    }
10 }
11 void *consumer(void *arg) {
12   for (i = 0; i < loops; i++) {
13     Mutex_lock(&m);
14     while (numfull == 0)
15       Cond_wait(&cond, &m);
16     int tmp = do_get();
17     Cond_signal(&cond);
18     Mutex_unlock(&m);
19     printf("%d\n", tmp);
20   }
21 }
```



Solution v2 (2 CVs, "if")

```
1 void *producer(void *arg) {
2   for (int i = 0; i < loops; i++) {
3     Mutex_lock(&m);
4     if (numfull == max)
5       Cond_wait(&empty, &m);
6     do_fill(i);
7     Cond_signal(&full);
8     Mutex_unlock(&m);
9   }
10 }
11 void *consumer(void *arg) {
12   for (i = 0; i < loops; i++) {
13     Mutex_lock(&m);
14     if (numfull == 0)
15       Cond_wait(&full, &m);
16     int tmp = do_get();
17     Cond_signal(&empty);
18     Mutex_unlock(&m);
19     printf("%d\n", tmp);
20   }
21 }
```

Stop & think:

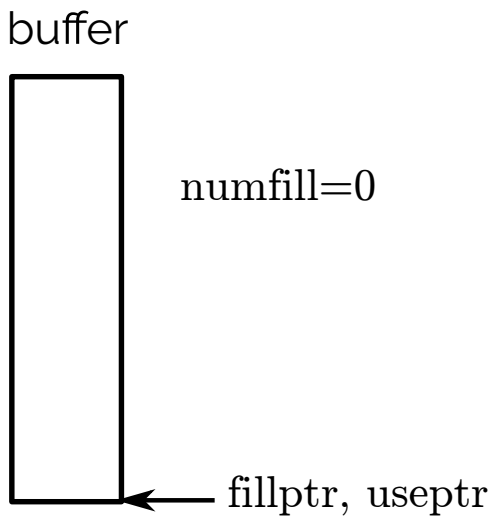
Find the problem!

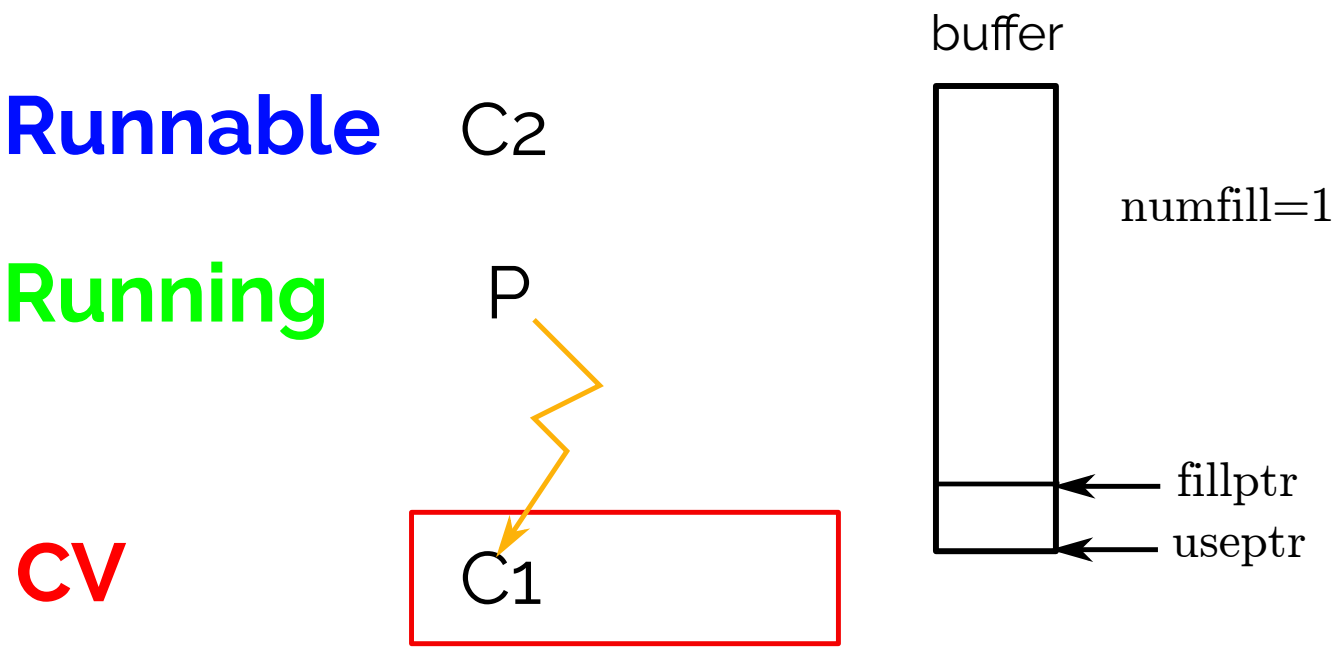
Big Bang

Runnable C2

Running P

CV C1

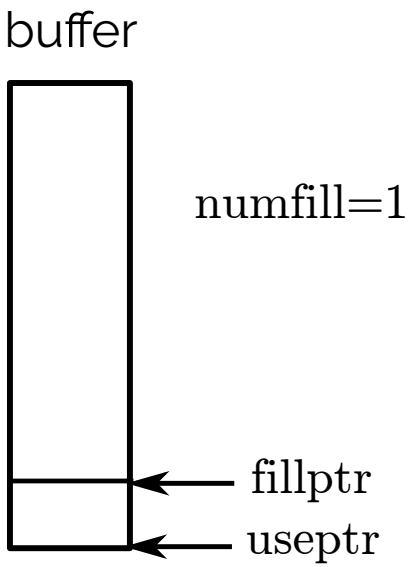


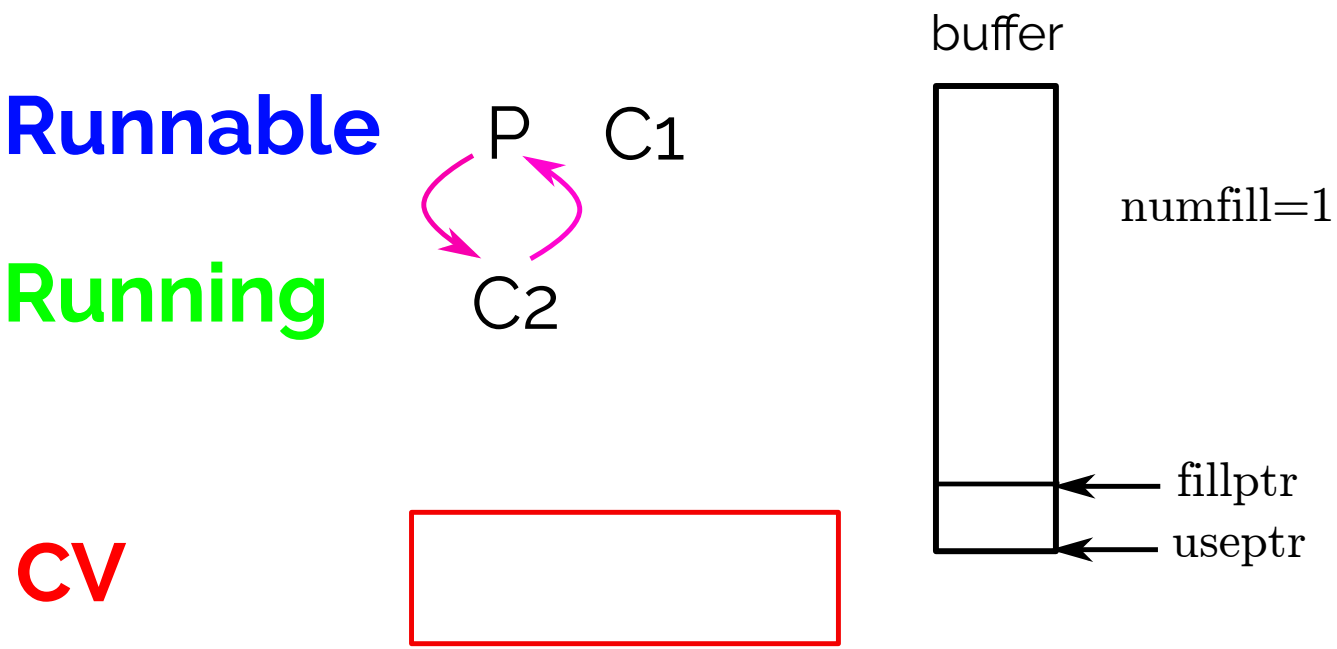

Runnable
C2
Running
P
CV
C1
buffer
numfill=1
fillptr
useptr

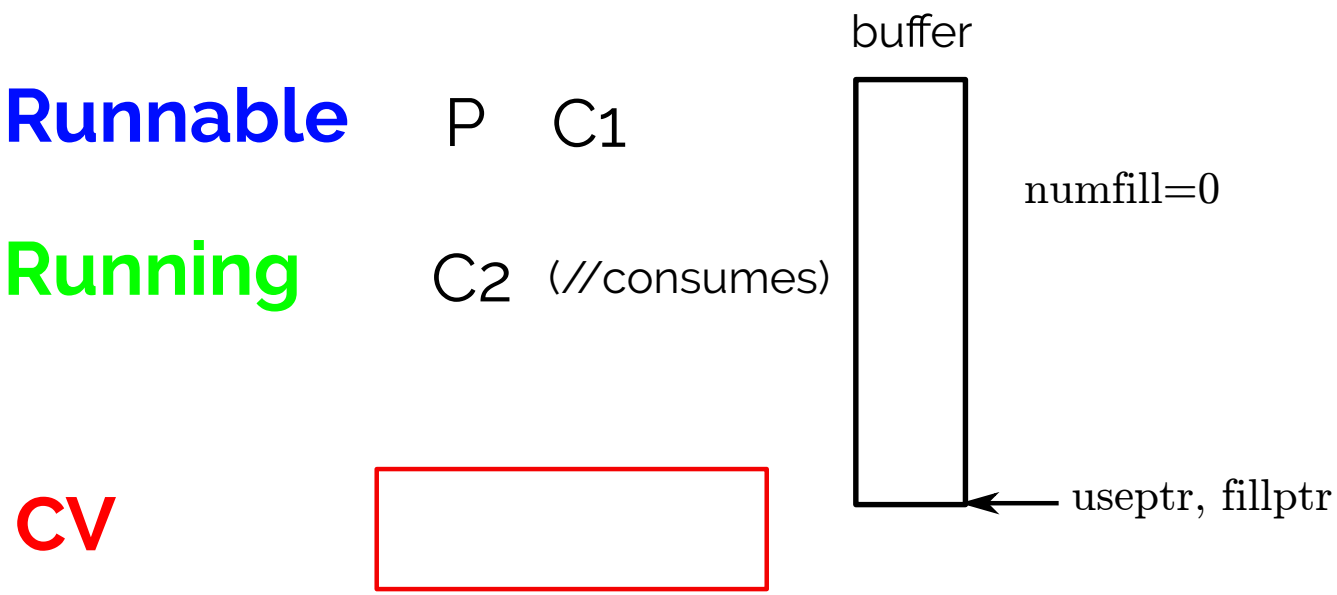
Runnable C2 C1

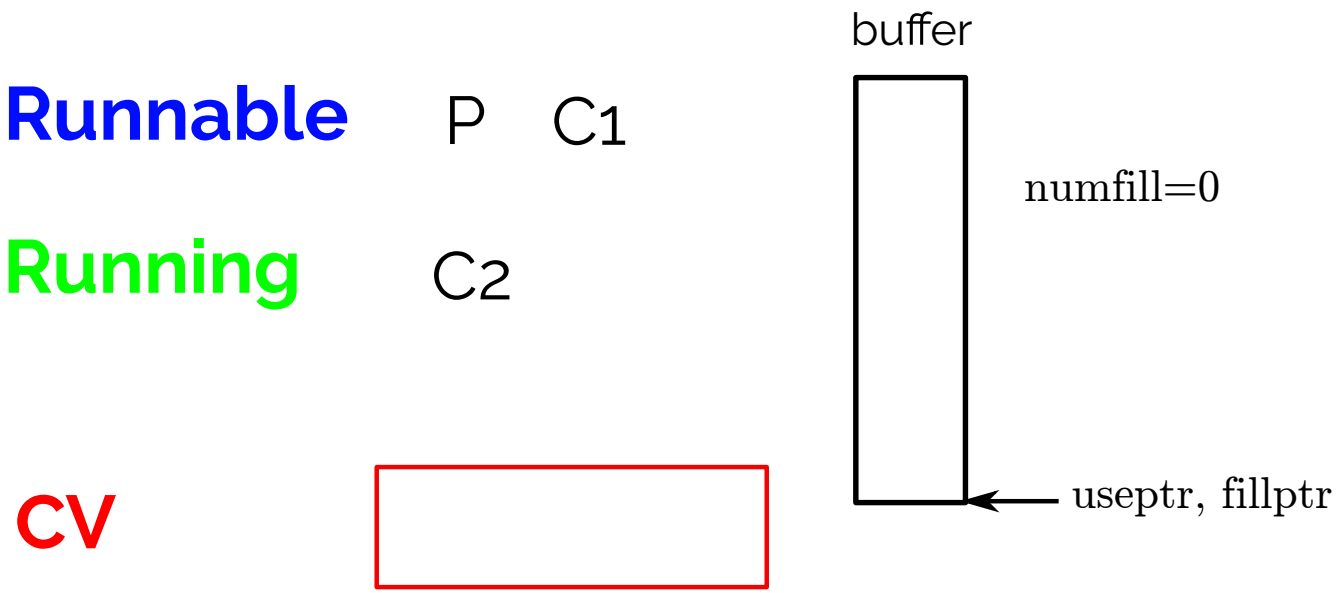
Running P

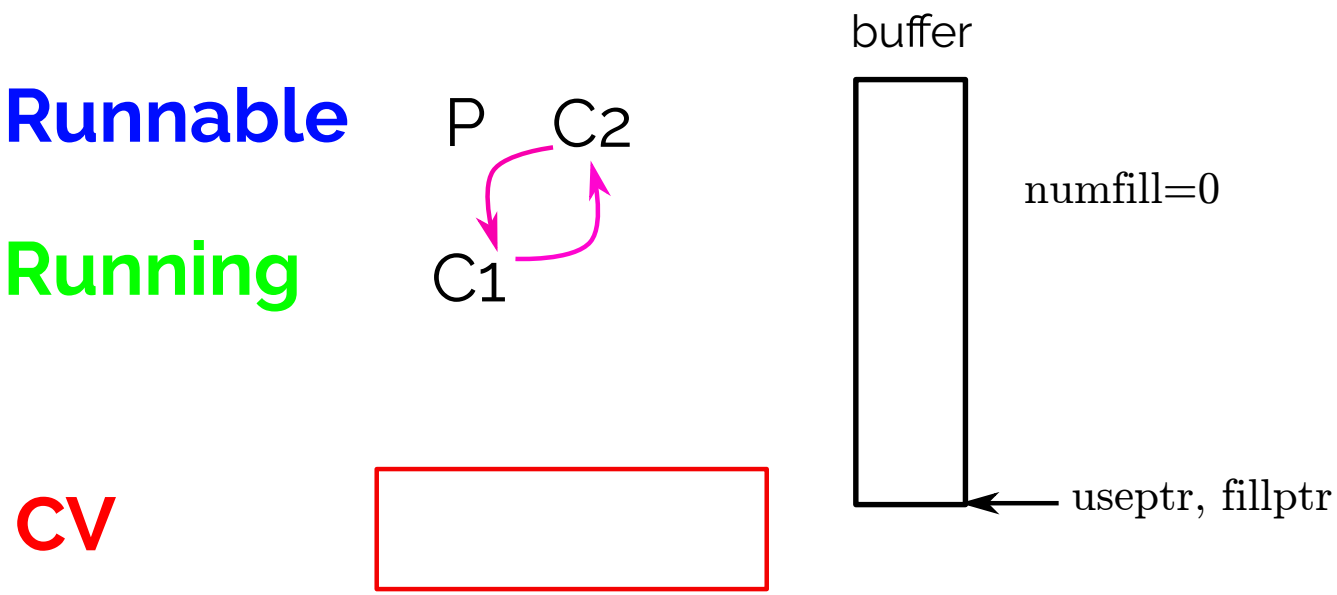
CV

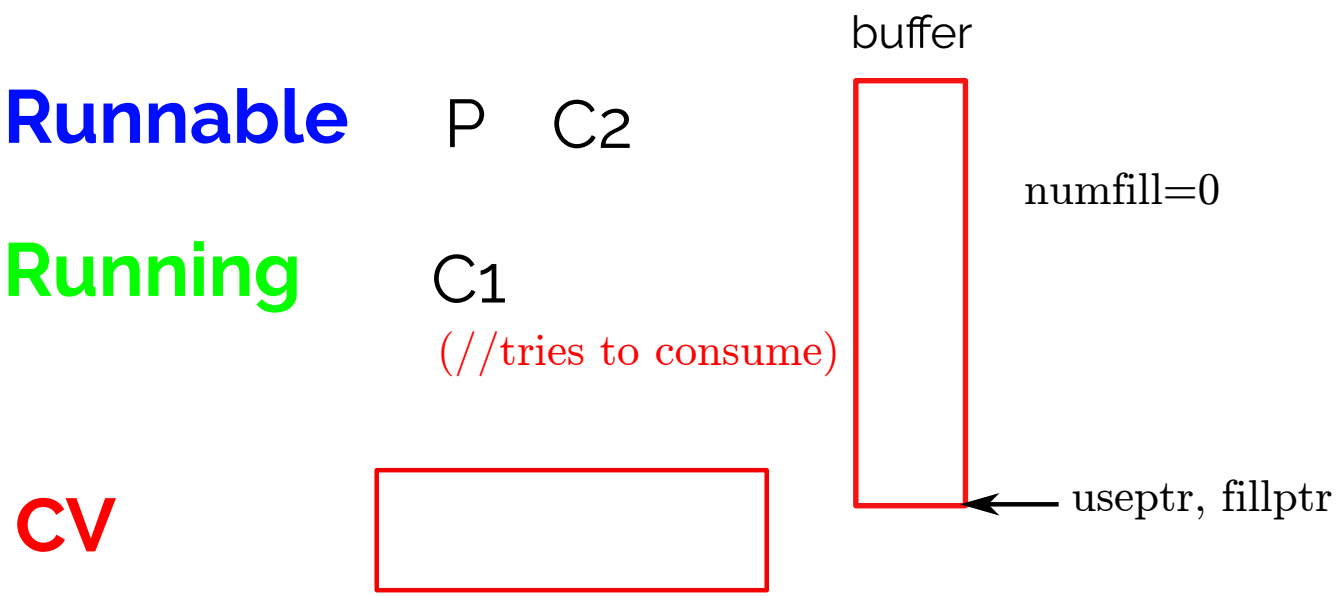



Runnable
Running
CV
P
C1
C2
buffer
numfill=1
fillptr
useptr


Runnable
P
C1
Running
C2
(//consumes)
CV
buffer
numfill=0
useptr, fillptr


Runnable
P
C1
Running
C2
CV
buffer
numfill=0
useptr, fillptr


Runnable
Running
CV
P
C2
C1
buffer
numfill=0
useptr, fillptr



assertion fires ...

Note

Mesa semantics

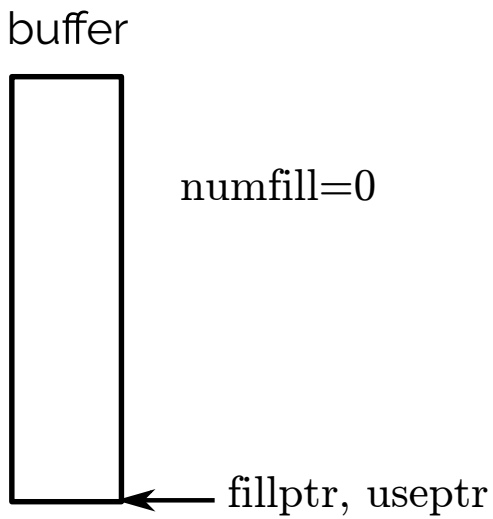
Signaling a thread only wakes them up; it is thus a hint that the state of the world has changed (in this case, that a value has been placed in the buffer), but there is no guarantee that when the woken thread runs, the state will still be as desired. This interpretation of what a signal means is often referred to as **Mesa semantic.** The contrast, referred to as **Hoare semantics**, is harder to build but provides a stronger guarantee that the woken thread will run immediately upon being woken. *Virtually every system ever built employs Mesa semantics*

Big Bang

Runnable C2

Running P

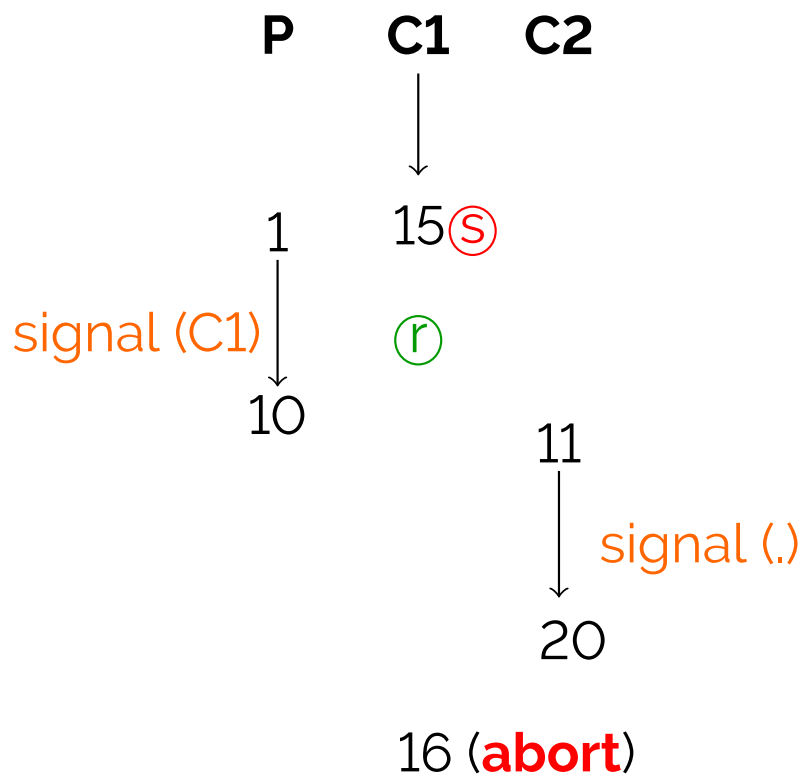
CV C1



But how to create one?

Solution v2 (2 CVs, "if")

```
1 void *producer(void *arg) {
2   for (int i = 0; i < loops; i++) {
3     Mutex_lock(&m);
4     if (numfull == max)
5       Cond_wait(&empty, &m);
6     do_fill(i);
7     Cond_signal(&full);
8     Mutex_unlock(&m);
9   }
10 }
11 void *consumer(void *arg) {
12   for (i = 0; i < loops; i++) {
13     Mutex_lock(&m);
14     if (numfull == 0)
15       Cond_wait(&full, &m);
16     int tmp = do_get();
17     Cond_signal(&empty);
18     Mutex_unlock(&m);
19     printf("%d\n", tmp);
20   }
21 }
```



Solution v3 (2 CVs, "while")

```
1 void *producer(void *arg) {
2   for (int i = 0; i < loops; i++) {
3     Mutex_lock(&m);
4     while (numfull == max)
5       Cond_wait(&empty, &m);
6     do_fill(i);
7     Cond_signal(&full);
8     Mutex_unlock(&m);
9   }
10 }
11 void *consumer(void *arg) {
12   for (i = 0; i < loops; i++) {
13     Mutex_lock(&m);
14     while (numfull == 0)
15       Cond_wait(&full, &m);
16     int tmp = do_get();
17     Cond_signal(&empty);
18     Mutex_unlock(&m);
19     printf("%d\n", tmp);
20   }
21 }
```

Summery of the Best Practice

- Define state(s), one lock, and CV(s)
- Within each public method using the shared data:

1. Hold the lock
2. Read/write state(s) (not always necessary)
3. `while (<waiting_condition_ statement>)`
   `cond_wait(&ĈV);`
   - number of CVs = number of different waiting conditions
4. read/write state(s) & the shared data
5. `cond_signal(&C̃V)`
   - `C̃V` may be `ĈV` or may not
6. Release the lock

Also we can use one CV and `cond_broadcast` accepting a performance loss

سوال ۳:

یک شرکت دانش بنیان قصد دارد یک برنامه کاربردی (application) به نام «کلینیک آنلاین» بنویسد که در آن پزشکان بتوانند به صورت مجازی بیماران را ویزیت کنند. در قسمت سرور برنامه کلینیک آنلاین برای هر پزشکی که برای ویزیت آماده است یک ترد (thread) از نوع پزشک و برای هر بیماری که برای ویزیت آماده است یک ترد از نوع بیمار ایجاد می شود. بنابراین تردها از دو نوع پزشک و یا بیمار هستند. در قسمتی از برنامه سرور، تابعی به نام visit قرار دارد که باید برای استفاده از آن ملاحظاتی صورت بگیرد:

در هر زمان دو ترد غیر هم جنس به ترتیب تابع visit را فراخوانی کنند و هر موقع کار هر دو با تابع visit به پایان رسید، دو ترد دیگر (مجدد یکی بیمار و یکی پزشک) می‌توانند از تابع visit استفاده نمایند. در غیر این صورت خطایی در برنامه رخ خواهد داد.

سؤال: میخواهیم دو تابع به نام های doctor و patient بنویسیم به نحوی که تردهای دکتر و بیمار پس از ایجاد شدن به ترتیب این توابع را شروع به اجرا کنند و هر ترد پس از استفاده مناسب از تابع visit به اجرای خود خاتمه دهد.

(برای حل این سوال از قفل و متغیرهای شرطی استفاده کنید)

Semaphore

In real life a semaphore is a system of signals used to communicate visually, usually with flags, lights, or some other mechanism.

In software, a semaphore is a data structure that is useful for solving a variety of synchronization problems.

Introduced by *Dijkstra* to provide synchronization in the **THE** operating system.

Semaphore: Definition

```
sem_init(sem_t *s, int value) {
  s->value = value;
}
sem_wait(sem_t *s) {
  if (s->value <= 0)
    put_self_to_sleep();
  s->value--;
}
sem_post(sem_t *s) {
  s->value++;
  wake_one_waiting_thread();
}
```

// Each routine executes **ATOMICALLY**

// If sem_post() enables a thread in sem_wait(), then sem_wait()'s increment and sem_post()'s decrement of value are atomic–no other thread can observe the incremented value, and the thread in sem_wait() is guaranteed to decrement the value and return.

A semaphore is like an **integer**, with three differences:

When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are **increment** (increase by one) and **decrement** (decrease by one). You cannot read the current value of the semaphore.

When a thread calls wait, it first observes the semaphore's value and decrements if it is **positive**, otherwise thread blocks itself (all atomically).

if signal occurs when there is waiting thread(s), then one is guaranteed to decrement the just incremented semaphore's value and return.

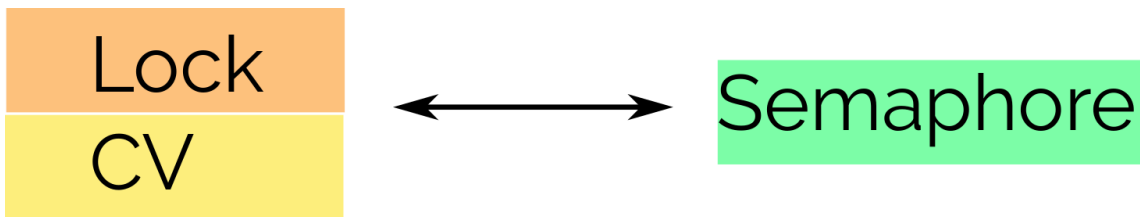
Note

Be careful about misleading names

It may be surprising that there are so many names, but there is a reason for the plurality. **increment** and **decrement** describe what the operations do. **signal (post)** and **wait** describe what they are often used for. And **V** and **P** were the original names proposed by *Dijkstra*, who wisely realized that a meaningless name is better than a misleading name

(Semaphore) vs. (lock + CV)

We can build them from each other



Some programmers like lock + CV

Some prefer working with semaphores

You? ☺

Important Differences

wait

`cond_wait()` releases its paired lock (urges its usage with "while"), but `sem_wait()` is not even paired with a lock. Moreover, after receiving a signal (or post), `cond_wait()` returns only with a lock held. However, `sem_wait()` returns immediately.

signal

Signals may be lost in CV

e.g., because of an empty CV's queue when a signal arrives

but they are not lost in Sem

even when the Sem's queue is empty, a signal is implicitly stored in **Sem's value**

The Sem's value plays the role of queue for signals

Semaphores considered harmful: programming with locks and condition variables is superior to programming with semaphores.

Code based on semaphores is not uncommon, especially in operating systems.

Semaphores in interrupt handlers are superior to condition variables and locks.

Mutual Exclusion (Lock)

```
1 sem_t lock;
2 void *worker(void *arg) {
3   int i;
4   // What goes here?
5   for (i = 0; i < 1e6; i++)
6     counter++;
7   // What goes here?
8   return NULL;
9 }
10 int main(int argc, char *argv[]) {
11   int num = atoi(argv[1]);
12   pthread_t pid[PMAX];
13   sem_init(&lock, /* What goes here? */);
14   for (int i = 0; i < num; i++)
15     Pthread_create(&pid[i], 0, worker, 0);
16   for (int i = 0; i < num; i++)
17     Pthread_join(pid[i], NULL);
18   printf("counter: %d\n", counter);
19   return 0;
20 }
```

Stop & think:

Find the solution!

Mutual Exclusion (Lock)

```
1 sem_t lock;
2 void *worker(void *arg) {
3   int i;
4   sem_wait(&lock);
5   for (i = 0; i < 1e6; i++)
6     counter++;
7   sem_post(&lock);
8   return NULL;
9 }
10 int main(int argc, char *argv[]) {
11   int num = atoi(argv[1]);
12   pthread_t pid[PMAX];
13   sem_init(&lock, 1);
14   for (int i = 0; i < num; i++)
15     Pthread_create(&pid[i], 0, worker, 0);
16   for (int i = 0; i < num; i++)
17     Pthread_join(pid[i], NULL);
18   printf("counter: %d\n", counter);
19   return 0;
20 }
```

Multiplex

Puzzle: Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads. In other words, no more than n threads can run in the critical section at the same time.

(*TBD in our online session*)

Join (serialization)

```
1  sem_t s;
2  void *child(void *arg) {
3    printf("child\n");
4    // What goes here?
5    return NULL;
6  }
7  int main(int argc, char *argv[]) {
8    pthread_t p;
9    printf("parent: begin\n");
10   sem_init(&s, /* What goes here? */);
11   Pthread_create(&p, 0, child, 0);
12   // What goes here?
13   printf("parent: end\n");
14   return 0;
15 }
```

Stop & think:

Find the solution!

Join (serialization)

```
1  sem_t s;
2  void *child(void *arg) {
3    printf("child\n");
4    sem_post(&s);
5    return NULL;
6  }
7  int main(int argc, char *argv[]) {
8    pthread_t p;
9    printf("parent: begin\n");
10   sem_init(&s, 0);
11   Pthread_create(&p, 0, child, 0);
12   sem_wait(&s);
13   printf("parent: end\n");
14   return 0;
15 }
```

Barrier

Every thread should run the following code:

```
1 rendezvous
2 critical point
```

The synchronization requirement is that no thread executes **critical point** until after all threads have executed **rendezvous**.

Stop & think:

Find the solution!

```
1 /*rendezvous ends here*/
2
3 // your code
4
5 /*critical point starts here*/
```

You can assume that there are n threads and that this value is stored in a variable, n, that is accessible from all threads.

When the first n - 1 threads arrive they should block until the nth thread arrives, at which point all the threads may proceed.

Barrier: a broken solution

```
1 /*rendezvous ends here*/
2 sem_wait(mutex);
3 count = count + 1;
4 sem_post(mutex)
5 if (count == n)
6   sem_post(barrier);
7 sem_wait(barrier);
8 /*critical point starts here*/
```

where `n`, `count`, `mutex` and `barrier` are globally defined as

```
1 int n; // the number of threads
2 int count = 0;
3 sem_t mutex;
4 sem_init(&mutex,1);
5 sem_t barrier;
6 sem_init(&barrier,0);
```

Stop & think:

What is wrong with this solution?

Barrier: a broken solution

```
1 /*rendezvous ends here*/
2 sem_wait(mutex);
3 count = count + 1;
4 sem_post(mutex)
5 if (count == n)
6   sem_post(barrier);
7 sem_wait(barrier);
8 /*critical point starts here*/
```

Why?

Barrier: a broken solution

```
1 /*rendezvous ends here*/
2 sem_wait(mutex);
3 count = count + 1;
4 sem_post(mutex)
5 if (count == n)
6   sem_post(barrier);
7 sem_wait(barrier);
8 /*critical point starts here*/
```

Deadly sleep

Puzzle: Does this code always create a deadly sleep? Can you find an execution path through this code that does not cause a deadly sleep?

Stop & think:

Fix the Problem!

Barrier: a working solution

```
1 /*rendezvous ends here*/
2 sem_wait(mutex);
3 count = count + 1;
4 sem_post(mutex)
5 if (count == n)
6   sem_post(barrier);
7 sem_wait(barrier);
8 sem_post(barrier);
9 /*critical point starts here*/
```

The only change is another post after waiting at the barrier. Now as each thread passes, it signals the semaphore so that the next thread can pass.

Barrier: a working solution

```
1 /*rendezvous ends here*/
2 sem_wait(mutex);
3 count = count + 1;
4 sem_post(mutex)
5 if (count == n)
6   sem_post(barrier);
7 sem_wait(barrier);
8 sem_post(barrier);
9 /*critical point starts here*/
10
```

turnstile

The only change is another post after waiting at the barrier. Now as each thread passes, it signals the semaphore so that the next thread can pass.

Barrier with lock and CV

```
1  /*rendezvous ends here*/
2  Mutex_lock(&m);
3  count = count + 1;
4  while(count<n)
5    Cond_wait(&barrier);
6  Cond_signal(&barrier);
7  Mutex_unlock(&m);
8  /*critical point starts here*/
9
```

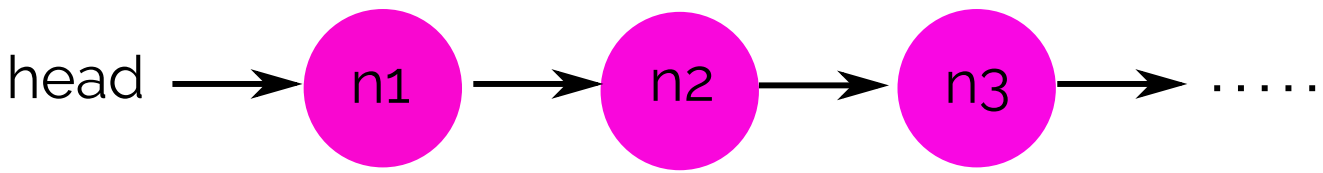
Readers-writers problem

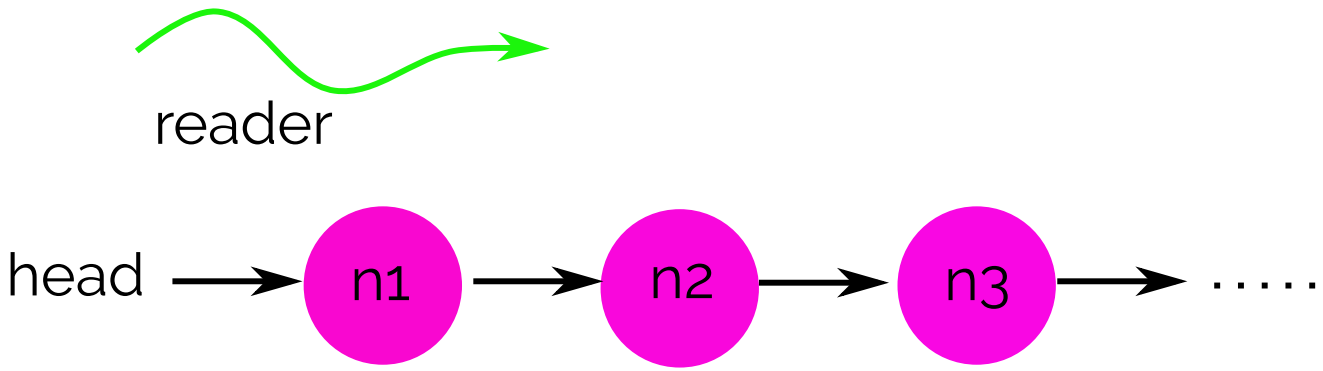
A classic problem (1971)

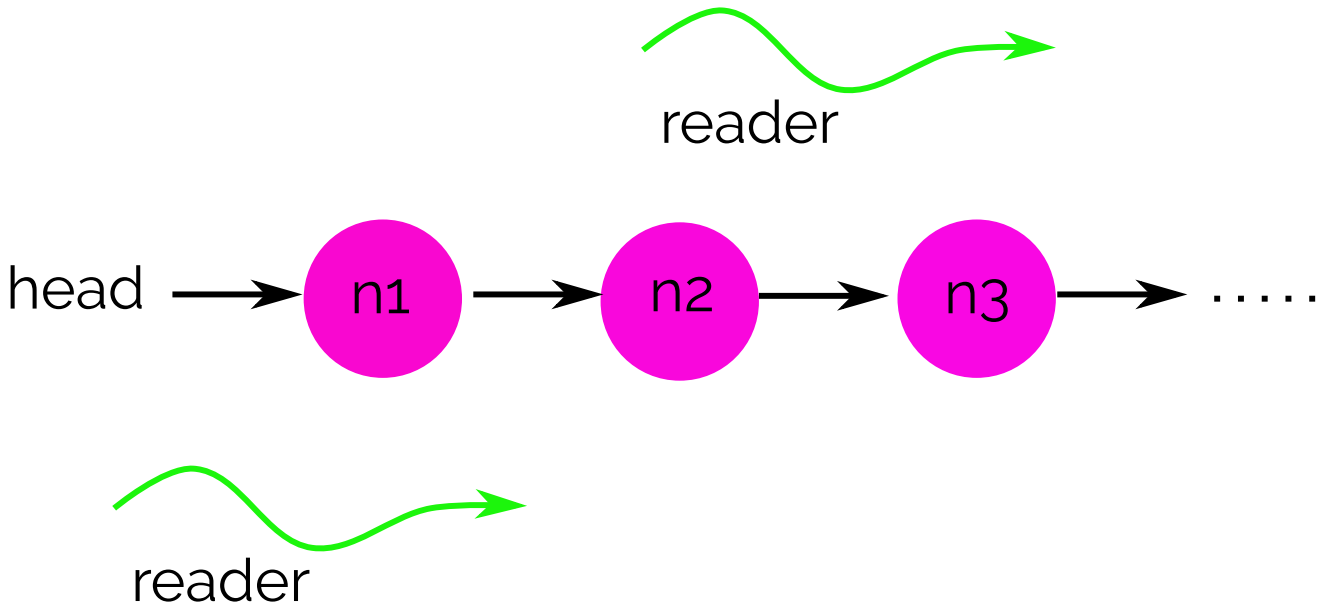
Consider list operations: `insert` and `lookup` .

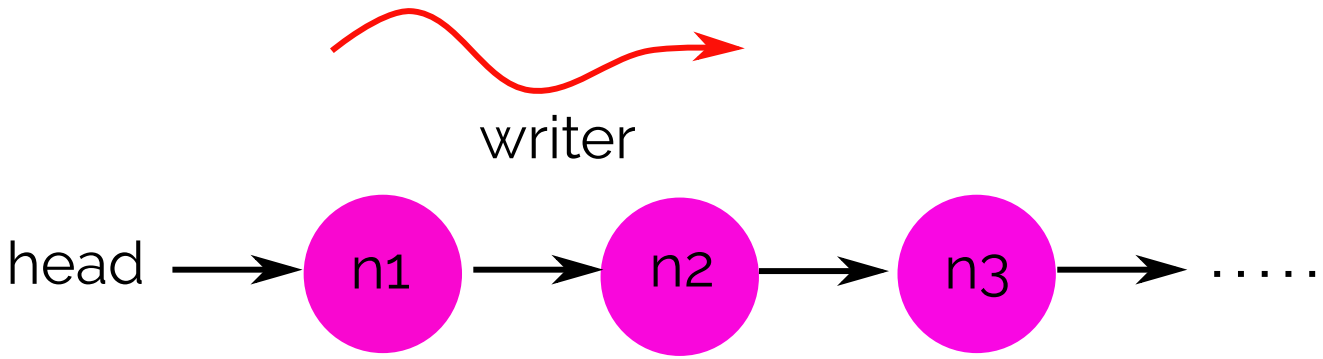
- inserts change the state of the list (*critical section* ...)
- lookups simply read the data structure → as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

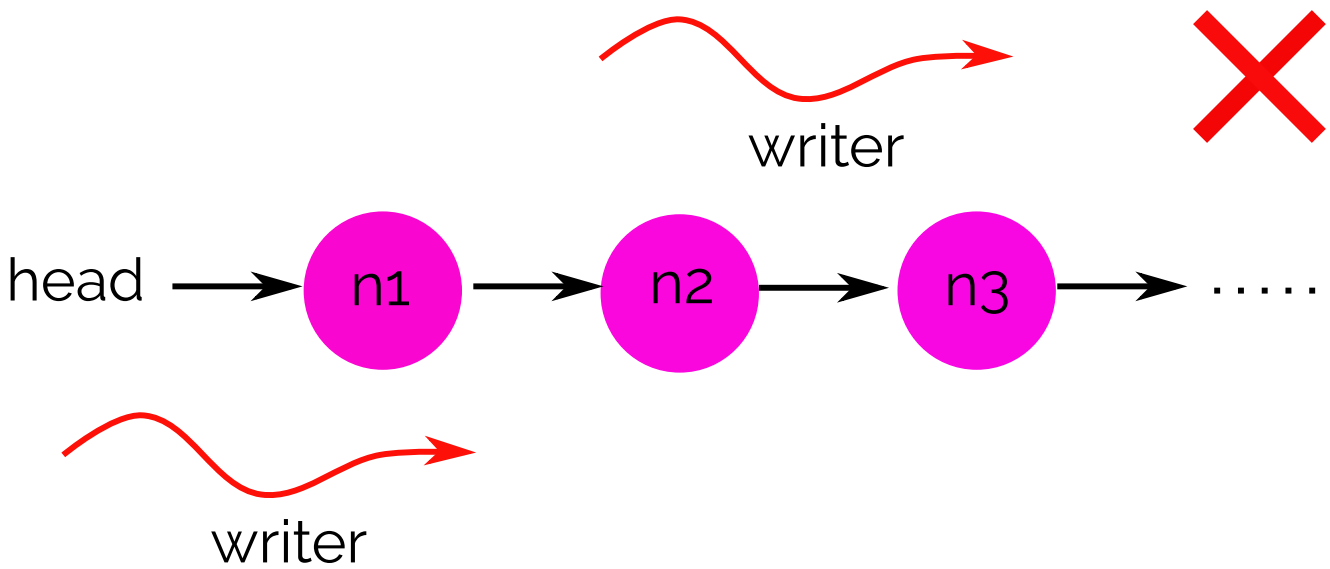
Different data structure accesses might require different kinds of locking

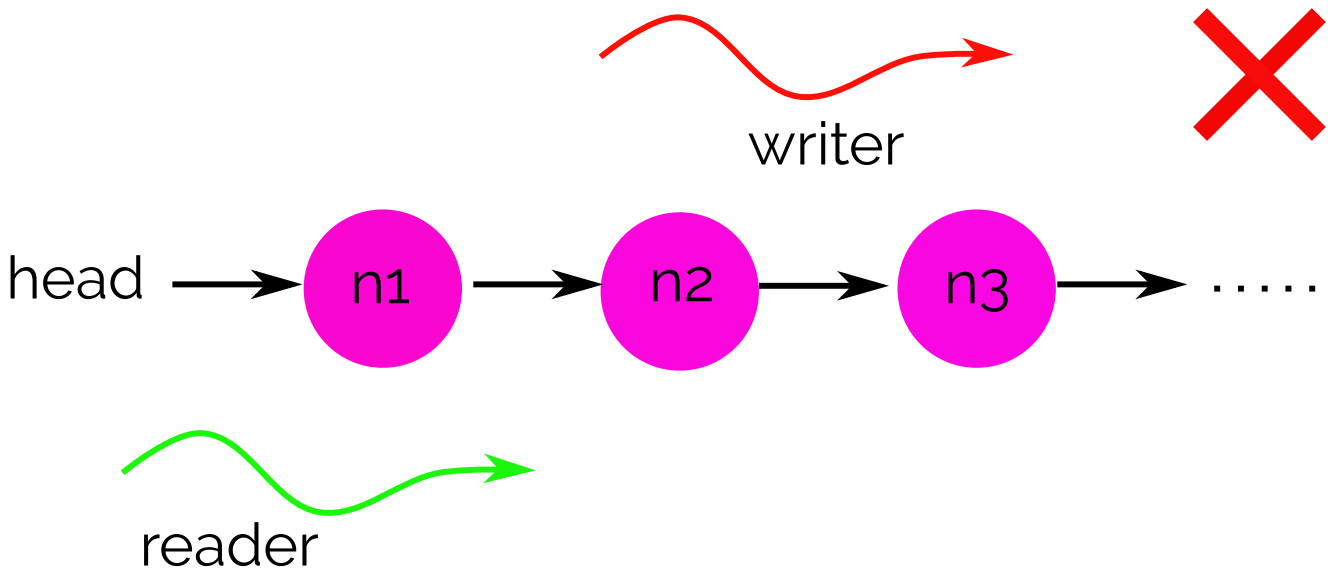

head
n1
n2
n3
.....


reader
head
n1
n2
n3
.....

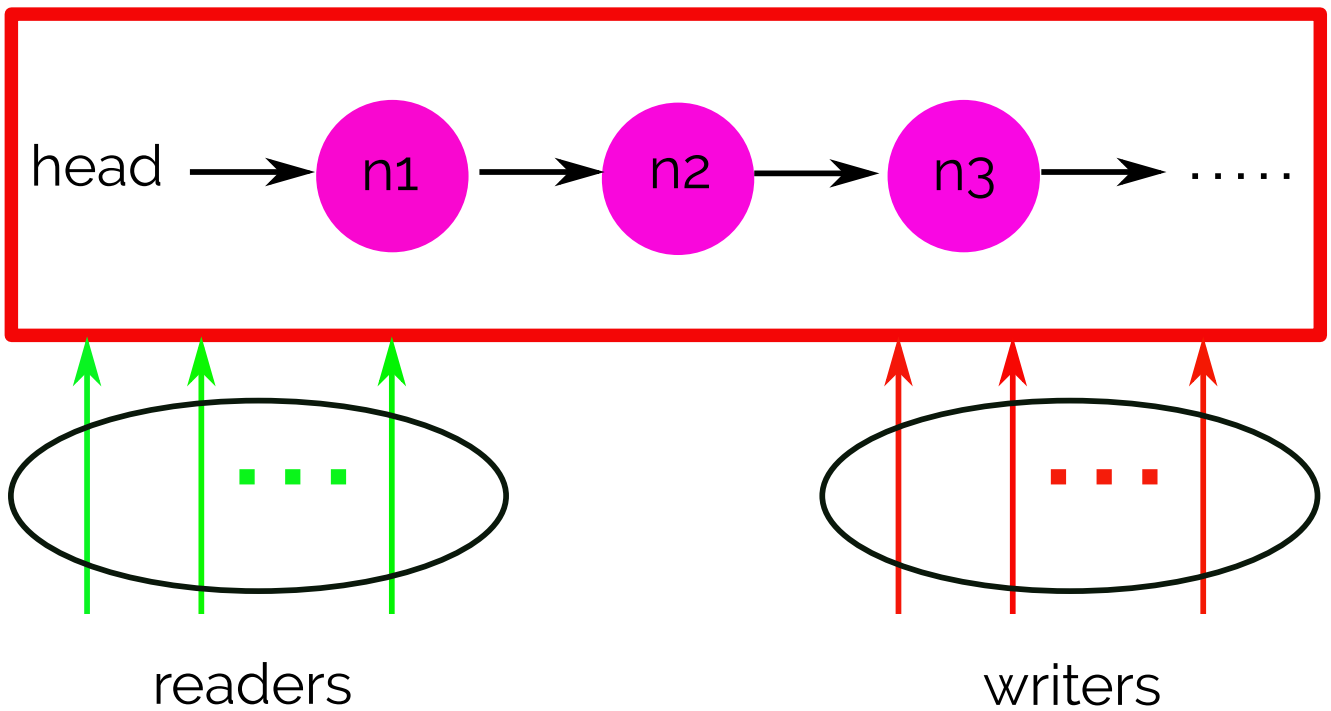

reader
head
n1
n2
n3
.....
reader


writer
head
n1
n2
n3
.....


writer
head
n1
n2
n3
.....
writer


writer
head
n1
n2
n3
.....
reader

Writers' Mutual Exclusion
Categorical Mutual Exclusion



Stop & think:

Find the solution!

Readers-writers with lock and CV

```
1  typedef struct _rwlock_t {
2    Mutex_t m;
3    Cond_t writerCV;
4    Cond_t readerCV;
5    int readers;
6    int writers;
7  } rwlock_t;
8  void rw_init(rwlock_t *L) {
9    L->readers = 0;
10   L->writers = 0;
11   L->m = PTHREAD_MUTEX_INITIALIZER;
12   L->writerCV = PTHREAD_COND_INITIALIZER;
13   L->readerCV = PTHREAD_COND_INITIALIZER;
14 }
15
```

```
1  void acquire_readlock(rwlock_t *L){
2  Mutex_lock(&L->m);
3  while (L->writers != 0)
4    Cond_wait(L->readerCV);
5  L->readers++;
6  Cond_signal(&L->readerCV);
7  Mutex_unlock(L->m)
8  }
9  void release_readlock(rwlock_t *L){
10 Mutex_lock(&L->m);
11 L->readers--;
12 if (L->readers == 0)
13   Cond_signal(&L->writerCV);
14 mutex_unlock(L->m);
15 }
16
```

```
1  void acquire_writelock(rwlock_t *L){
2  Mutex_lock(&L->m);
3  while (L->readers != 0 && L->writers !=0)
4    Cond_wait(L->writerCV);
5  L->writer++;
6  Mutex_unlock(L->m)
7  }
8  void release_writelock(rwlock_t *L){
9  Mutex_lock(&L->m);
10 L->writers--;
11 Cond_signal(&L->readerCV);
12 Cond_signal(&L->writerCV);
13 Mutex_unlock(L->m)
14 }
15
```

Readers-writers with lock and CV

```
1  typedef struct _rwlock_t {
2  Mutex_t m;
3  Cond_t writerCV;
4  Cond_t readerCV;
5  int readers;
6  int writers;
7  } rwlock_t;
8  void rw_init(rwlock_t *L) {
9  L->readers = 0;
10 L->writers = 0;
11 L->m = PTHREAD_MUTEX_INITIALIZER;
12 L->writerCV = PTHREAD_COND_INITIALIZER;
13 L->readerCV = PTHREAD_COND_INITIALIZER;
14 }
15
```

Readers-writers: Solution

```
1  typedef struct _rwlock_t {
2    sem_t write_lock;
3    sem_t lock;
4    int readers;
5  } rwlock_t;
6  void rw_init(rwlock_t *L) {
7    L->readers = 0;
8    sem_init(&L->lock, 1);
9    sem_init(&L->write_lock, 1);
10 }
11
```

```
1  void acquire_readlock(rwlock_t *L){
2    sem_wait(&L->lock);
3    L->readers++;
4    if (L->readers == 1)
5      sem_wait(&L->write_lock);
6    sem_post(&L->lock);
7  }
8  void release_readlock(rwlock_t *L){
9    sem_wait(&L->lock);
10   L->readers--;
11   if (L->readers == 0)
12     sem_post(&L->write_lock);
13   sem_post(&L->lock);
14 }
15 void acquire_writelock(rwlock_t *L){
16   sem_wait(&L->write_lock);
17 }
18 void release_writelock(rwlock_t *L){
19   sem_post(&L->write_lock);
20 }
```

Puzzle: Is it possible for a writer to starve?

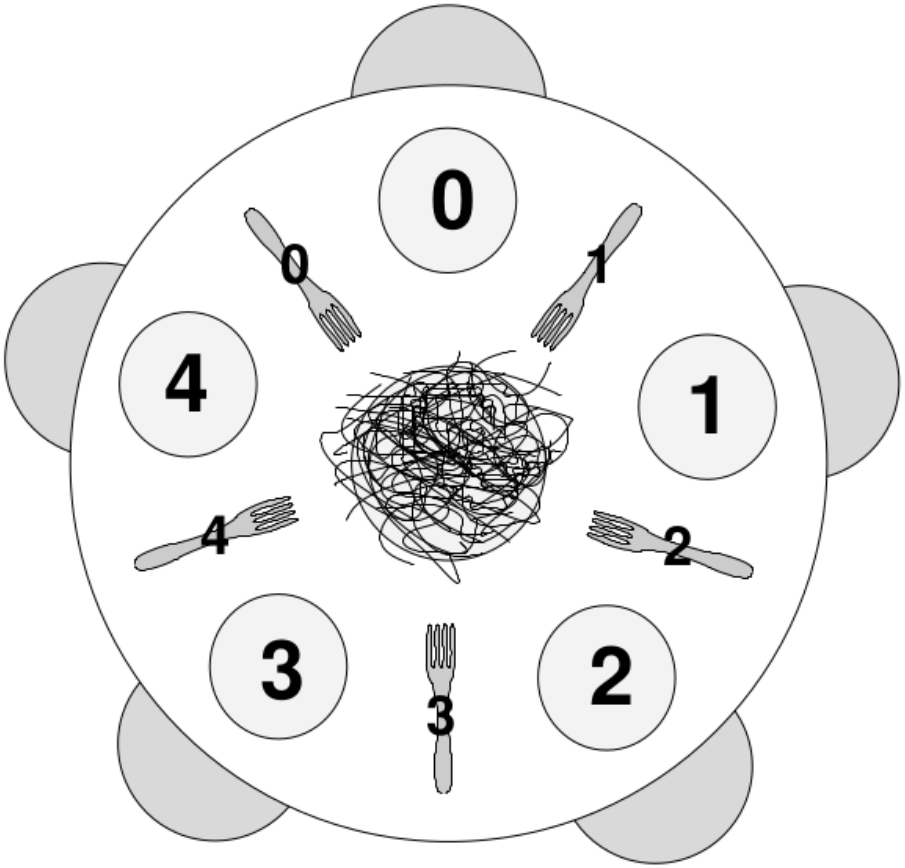
Dining Philosopher's Problem

One of the most famous concurrency problems posed, and solved, by Dijkstra (1965)

interesting, however, its practical utility is low

What each prof does:

```
while (1) {
   think();
   get_forks(p);
   eat();
   put_forks(p);
}
```



Unreal part: must have two forks even if starving!

Stop & think:

How to write a version of `get_forks` and `put_forks` such that there is no deadlock, no philosopher starves, and concurrency is high.

```
1 /* helper functions */
2 int right (int i){
3   return i;
4 }
5 int left (int i){
6   return (i+1)%5;
7 }
```

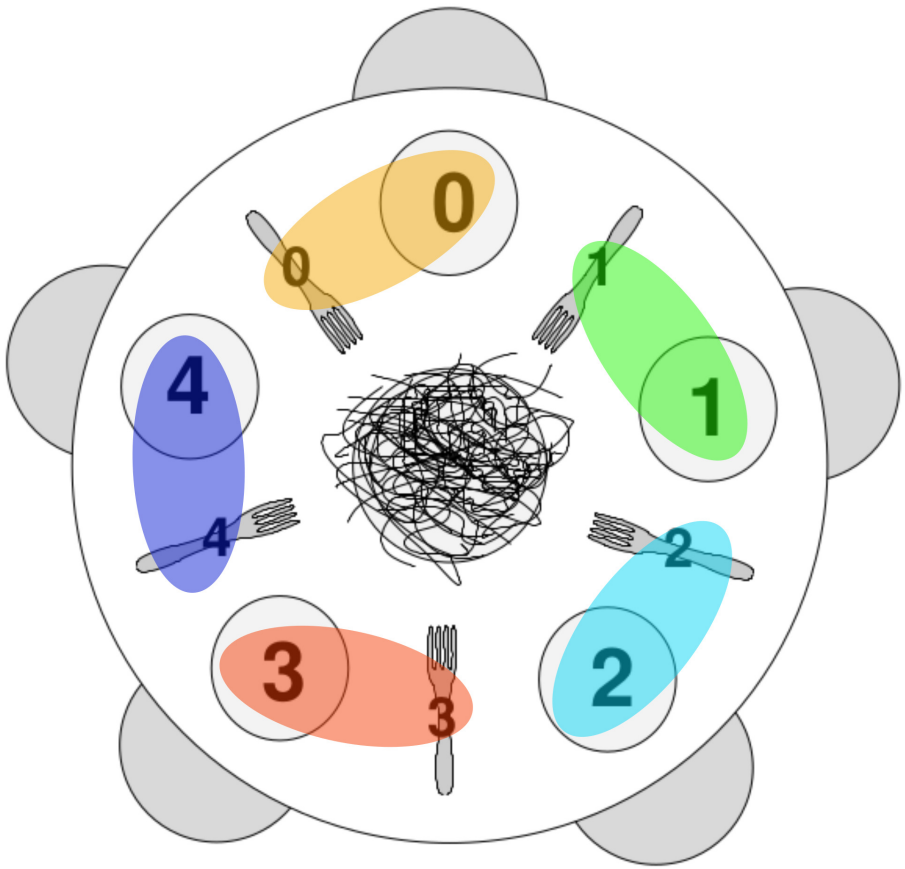
Hint: we need some semaphores to solve this problem. Let us assume we have five, one for each fork:

```
1 sem_t forks[5];
2 void forks_init(semt_t *forks){
3   for (int i=0; i < 5; i++)
4     sem_init(&forks[i], 1);
5 }
```

Broken Solution

```
1 void get_forks(int p) {
2   sem_wait(&forks[right(p)]);
3   sem_wait(&forks[left(p)]);
4 }
5 void put_forks(int p) {
6   sem_post(&forks[right(p)]);
7   sem_post(&forks[left(p)]);
8 }
```

Find the execution path for the following **deadlock**:



Stop & think:

Write a solution to this problem that prevents deadlock.

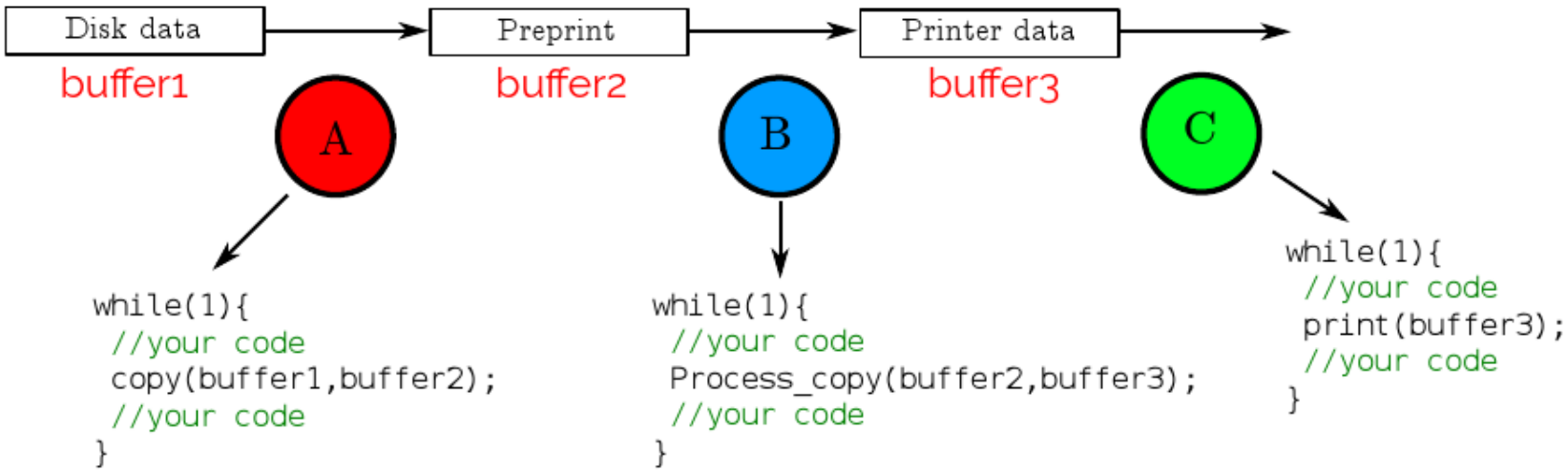
Solution: Breaking The Dependency

The simplest way to attack this problem is to change how forks are acquired by at least one of the philosophers

```
1 void get_forks(int p) {
2   if (p == 4) {
3     sem_wait(&forks[left(p)]);
4     sem_wait(&forks[right(p)]);
5   } else {
6     sem_wait(&forks[right(p)]);
7     sem_wait(&forks[left(p)]);
8   }
9 }
```

سوال ۴:

فرض کنید در عملیات چاپ یک فایل، سه فرایند درگیر باشند. فرایند A محتوای فایل را که توسط دیسک در بافر ۱ نوشته شده است در بافر ۲ کپی می‌کند. فرایند B اطلاعات را پس از پردازش از بافر اول به بافر دوم کپی می‌کند. در نهایت نیز فرایند C فایل را از بافر دوم برداشته و محتوای آن را چاپ می‌کند.



فرض کنید که فرایند A توسط interrupt handler دیسک و پس از اتمام عملیات نوشتن محتویات فایل در داخل بافر توسط دستور sem_post(&full1) متوجه وجود داده جدید در بافر می‌شود (full1 یک سمافور است). همچنین فرض کنید ظرفیت هر بافر نیز به اندازه یک فایل است. مشخص کنید که دربخش های مشخص شده در کد هر فرایند چه دستوراتی باید قرار گیرد.

راهنمایی: برای پر کردن بخش‌های خواسته شده از کدها تنها مجازید از دستورات sem_wait و sem_post استفاده کنید (تعریف سمافورهای لازم و تعیین مقدار اولیه آنها نیز الزامی است).

Concurrency Bugs

Therac-25

From Wikipedia, the free encyclopedia

The **Therac-25** was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982 after the Therac-6 and Therac-20 units (the earlier units had been produced in partnership with CGR of France).

Therac-25

It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation.[1]:425 Because of concurrent programming errors (also known as race conditions), it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.[2] These accidents highlighted the dangers of software control of safety-critical systems, and they have become a standard case study in health informatics and software engineering. Additionally the overconfidence of the engineers[1]:428 and lack of proper due diligence to resolve reported software bugs are highlighted as an extreme case where the engineers' overconfidence in their initial work and failure to believe the end users' claims caused drastic repercussions.

Therac-25

From Wikipedia, the free encyclopedia

The **Therac-25** was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982 after the Therac-6 and Therac-20 units (the earlier units had been produced in partnership with CGR of France).


Therac-25

It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation.[1]:425 Because of concurrent programming errors (also known as race conditions), it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.[2] These accidents highlighted the dangers of software control of safety-critical systems, and they have become a standard case study in health informatics and software engineering. Additionally the overconfidence of the engineers[1]:428 and lack of proper due diligence to resolve reported software bugs are highlighted as an extreme case where the engineers' overconfidence in their initial work and failure to believe the end users' claims caused drastic repercussions.

symptoms of radiation poisoning; in three cases, the injured patients later died as a result of the overdose.[5]

Getting concurrency right can sometimes save lives!

What Types Of Bugs Exist?

A study by Lu et al:

Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Non-Deadlock Bugs

Non-Deadlock Bugs

Atomicity

Order

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations.

Atomicity: MySQL

Thread 1:

```
if (thd->proc_info) {
...
fputs(thd->proc_info, ...);
...
}
```

Thread 2:

```
thd->proc_info = NULL;
```

What's wrong?

Atomicity: MySQL

Thread 1:

```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
...
fputs(thd->proc_info, ...);
...
} pthread_mutex_unlock(&lock);
```

Thread 2:

```
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Ordering: Mozilla

Thread 1:

```
void init() { ...
mThread
= PR_CreateThread(mMain, ...);
...
}
```

Thread 2:

```
...
mState = mThread->State;
...
```

What's wrong?

Ordering: Mozilla

Thread 1:

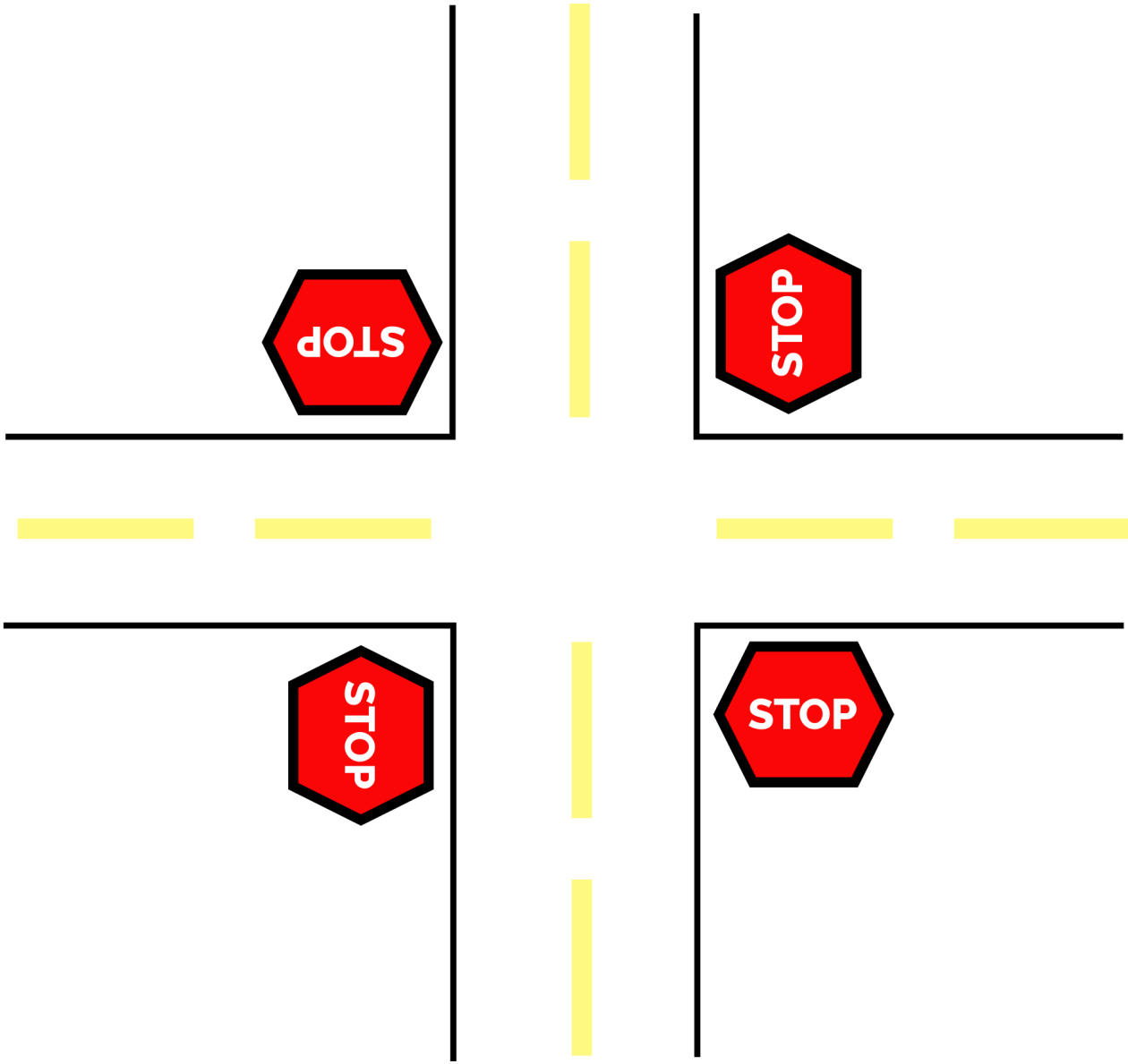
```
void init() { ...
mThread
= PR_CreateThread(...);
...
pthread_mutex_lock(&mtLock);
mtInit = 1;
pthread_cond_signal(&mtCond);
pthread_mutex_unlock(&mtLock);
}
```

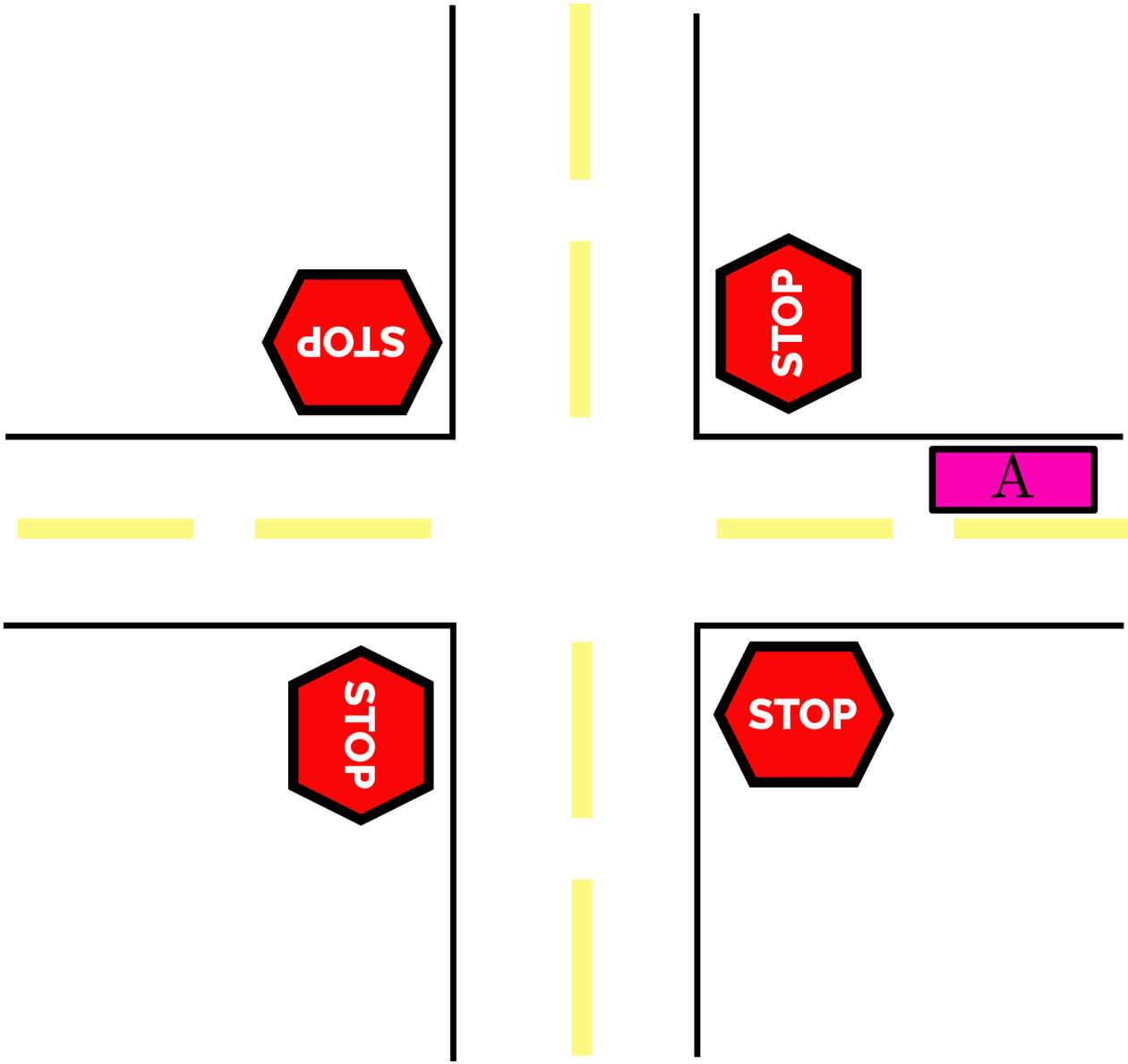
Thread 2:

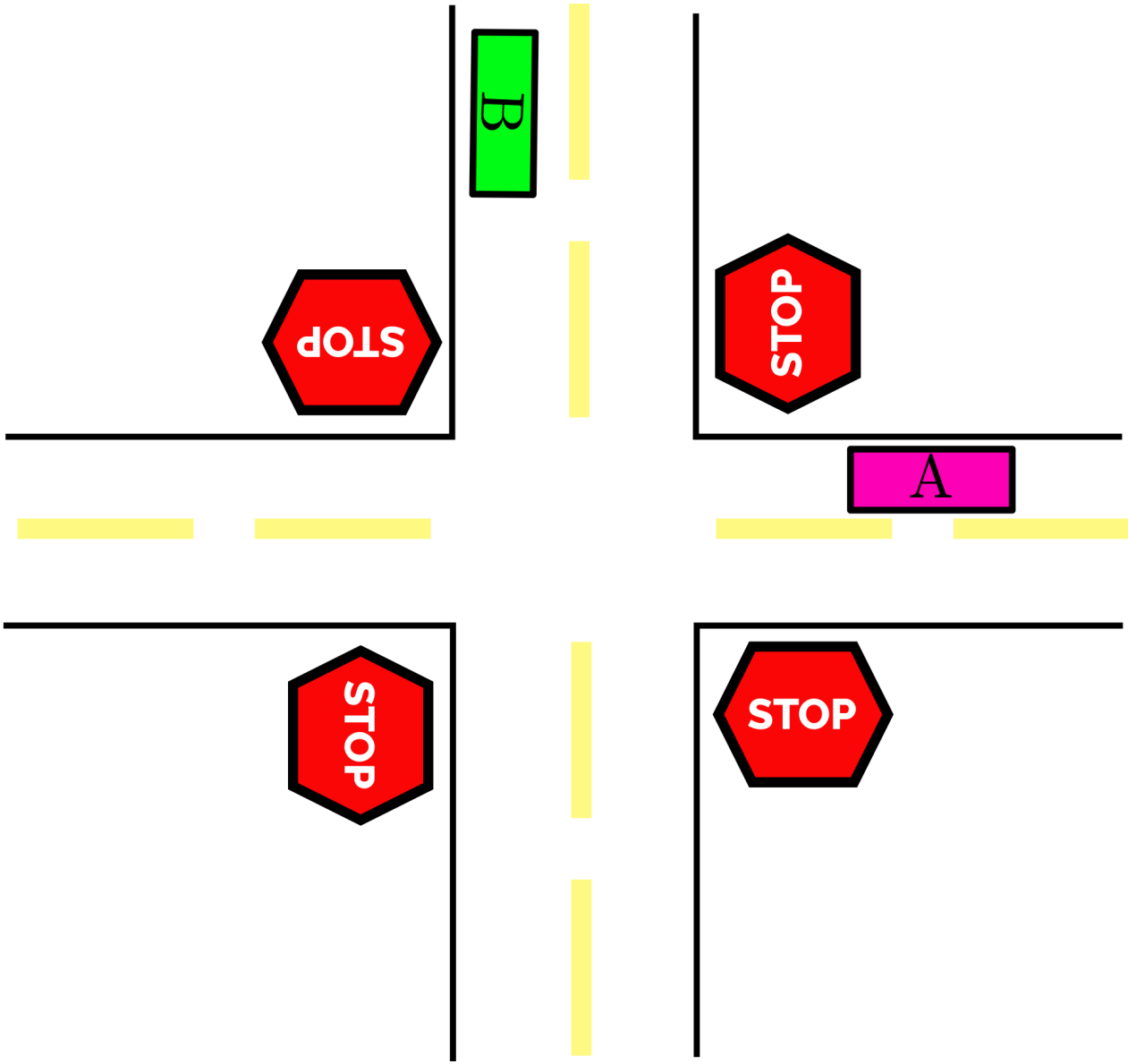
```
...
Mutex_lock(&mtLock);
while(mtInit == 0)
  Cond_wait(&mtCond, &mtLock);
Mutex_unlock(&mtLock);
mState = mThread->State;
...
```

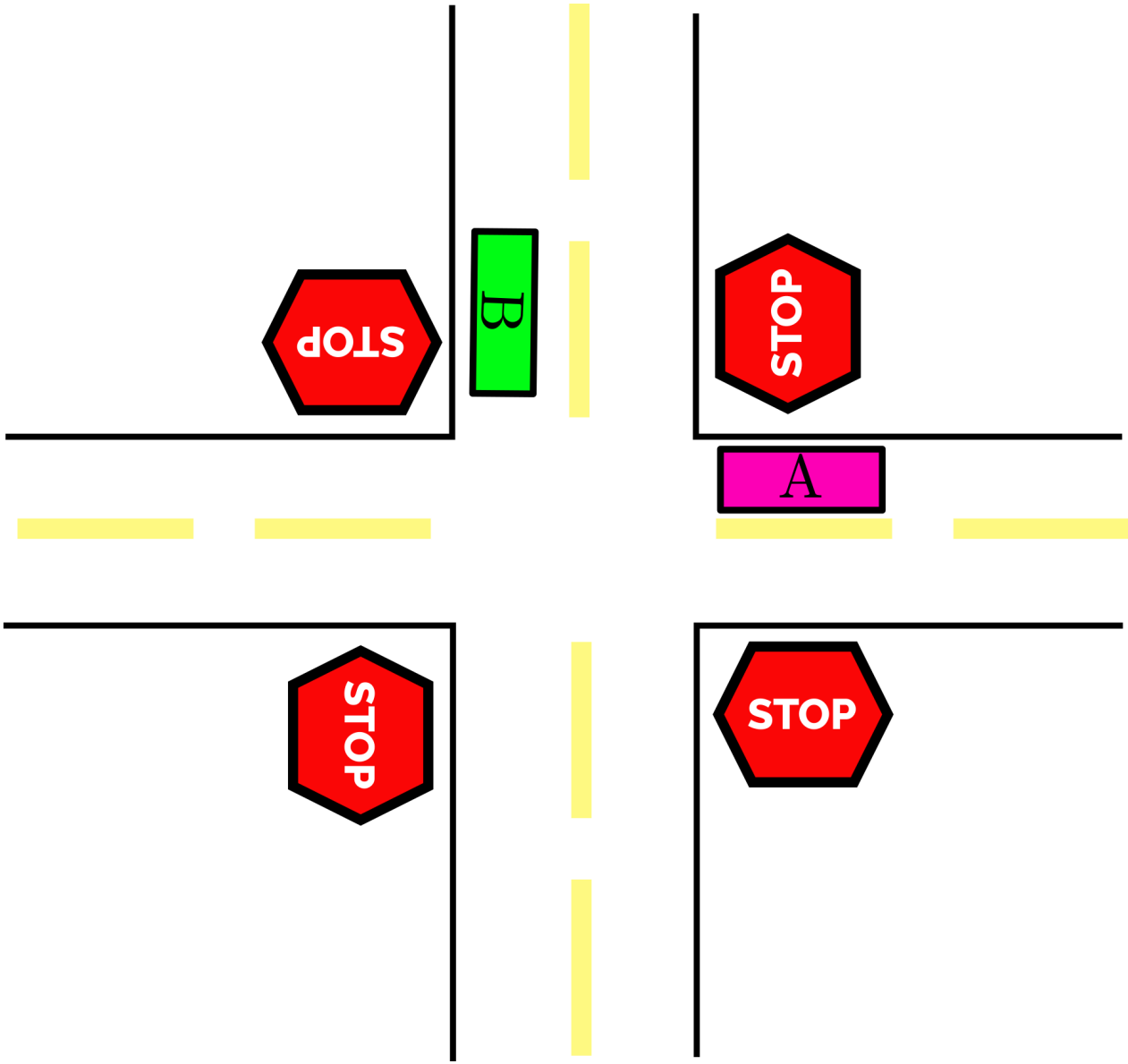
Deadlock Bugs

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. –Kansas state law, early 1900s

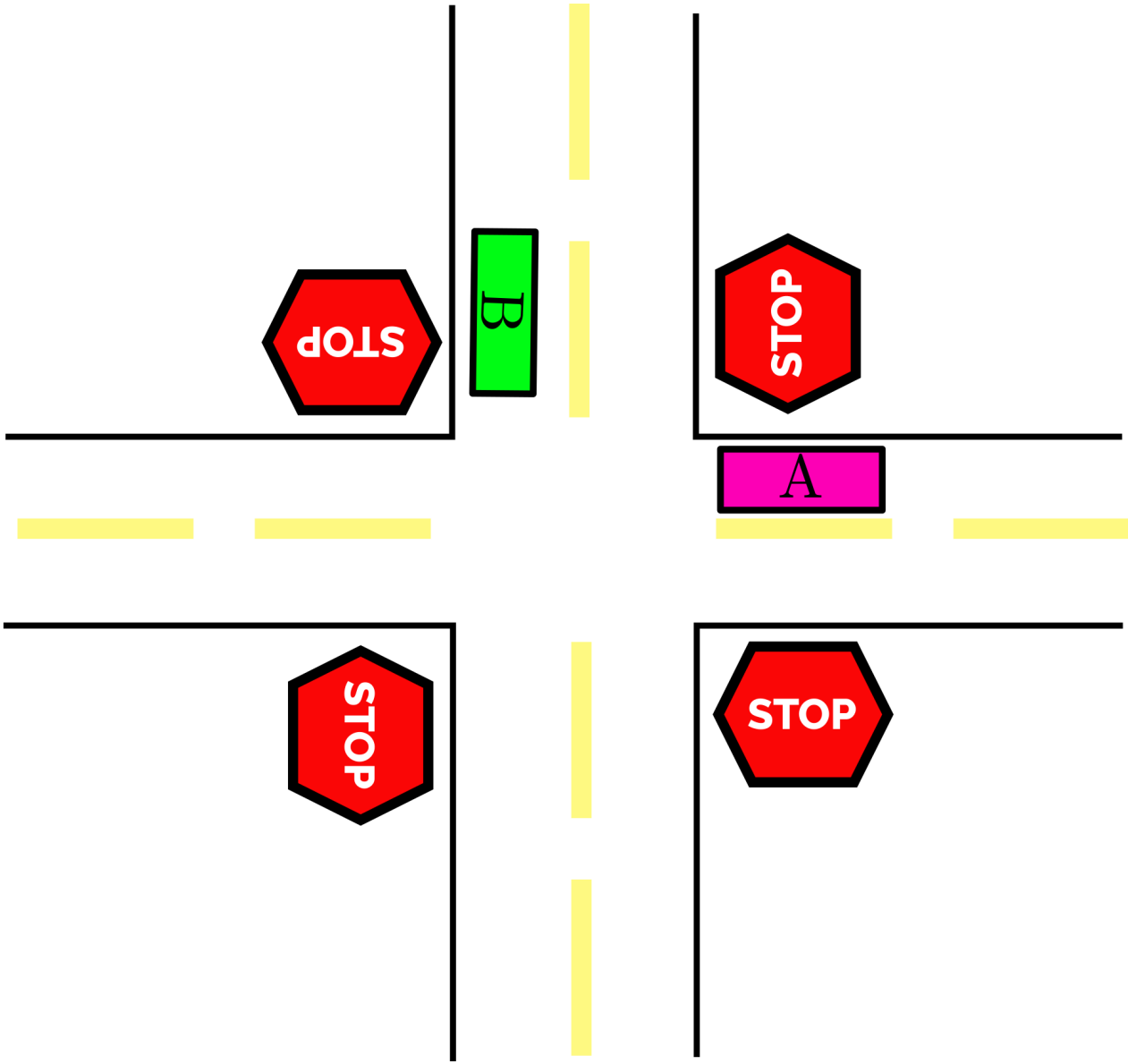

STOP
STOP
STOP
STOP

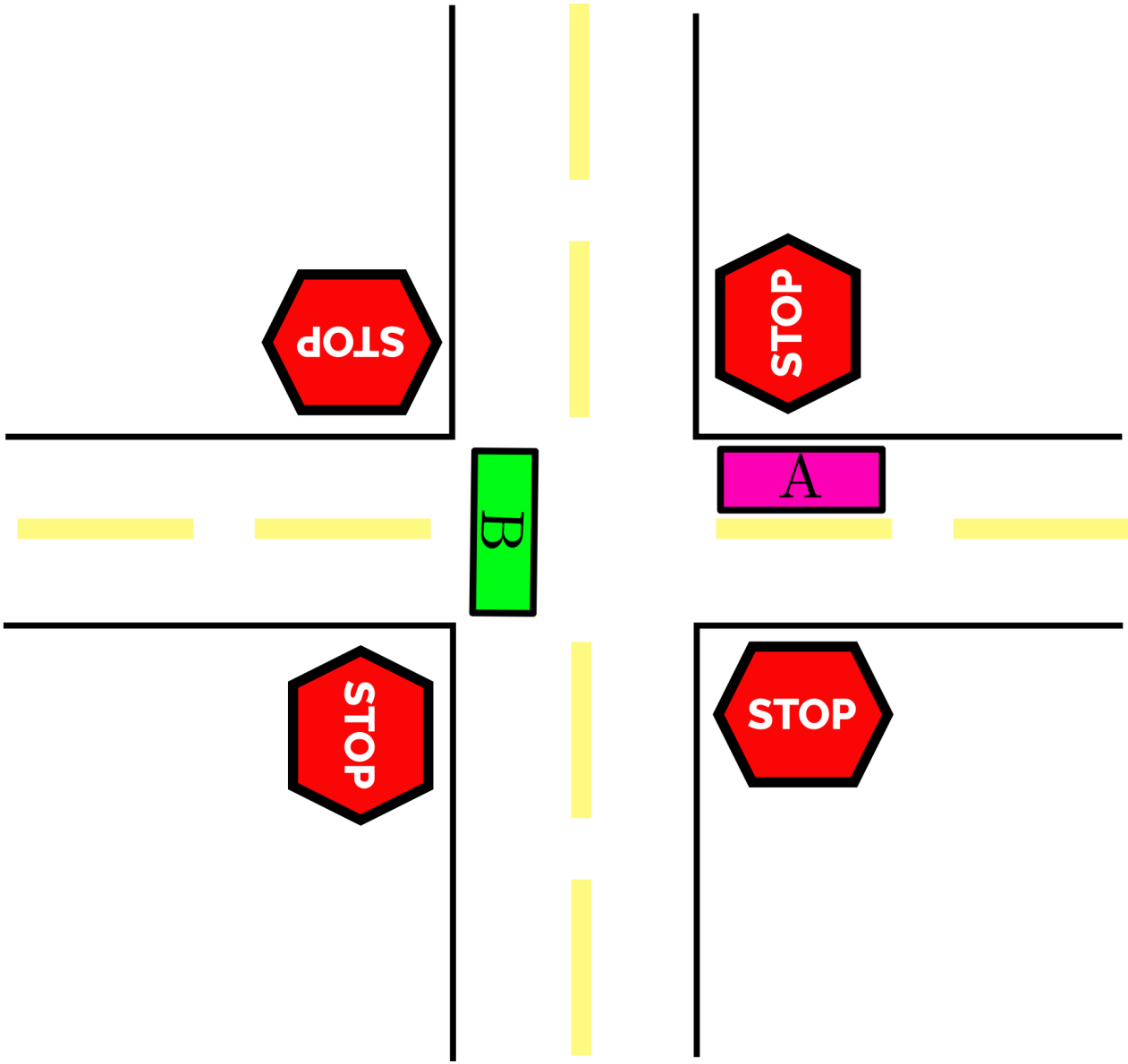

STOP
STOP
A
STOP
STOP

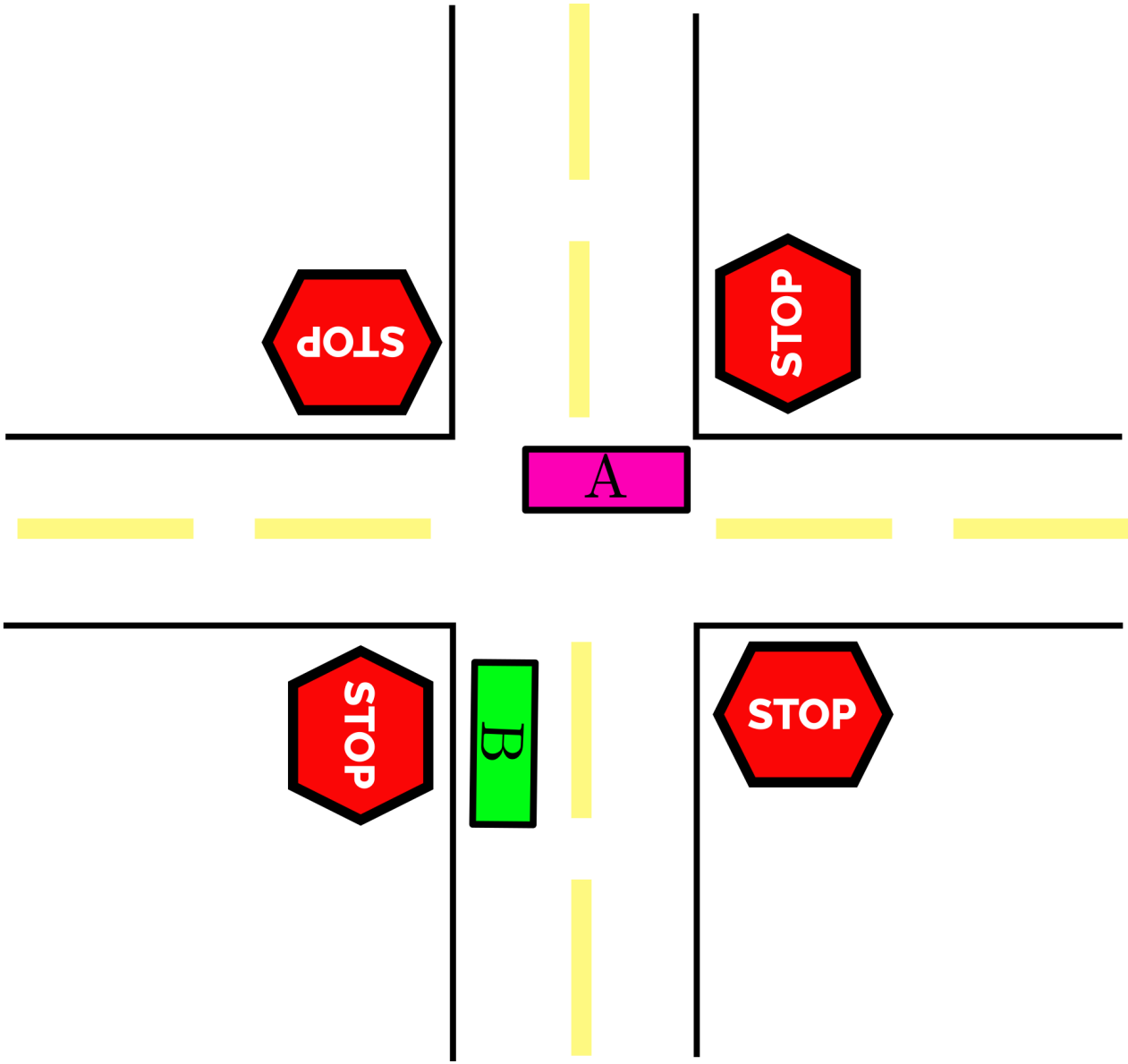

B
STOP
STOP
A
STOP
STOP

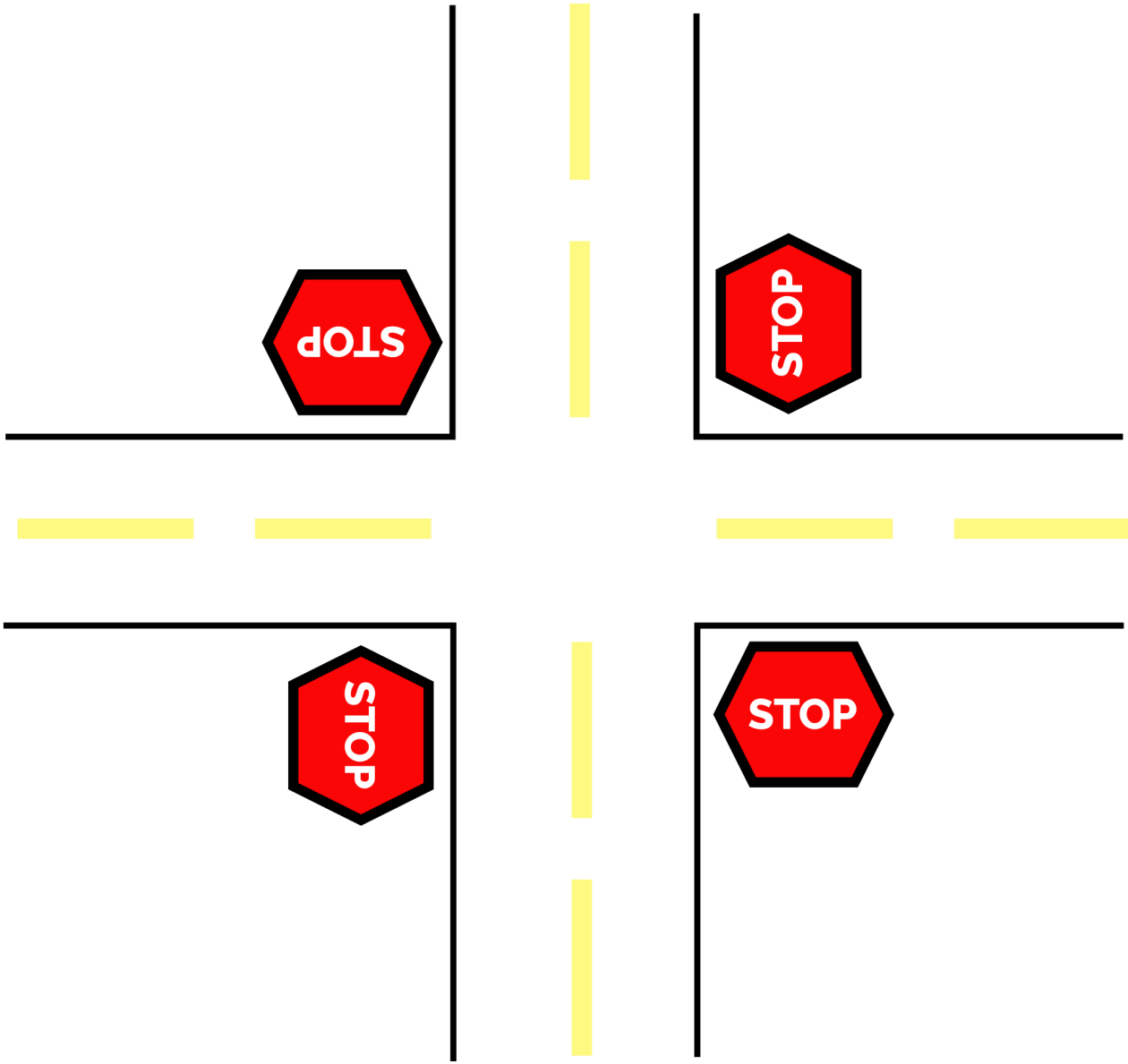

STOP
B
STOP
A
STOP
STOP

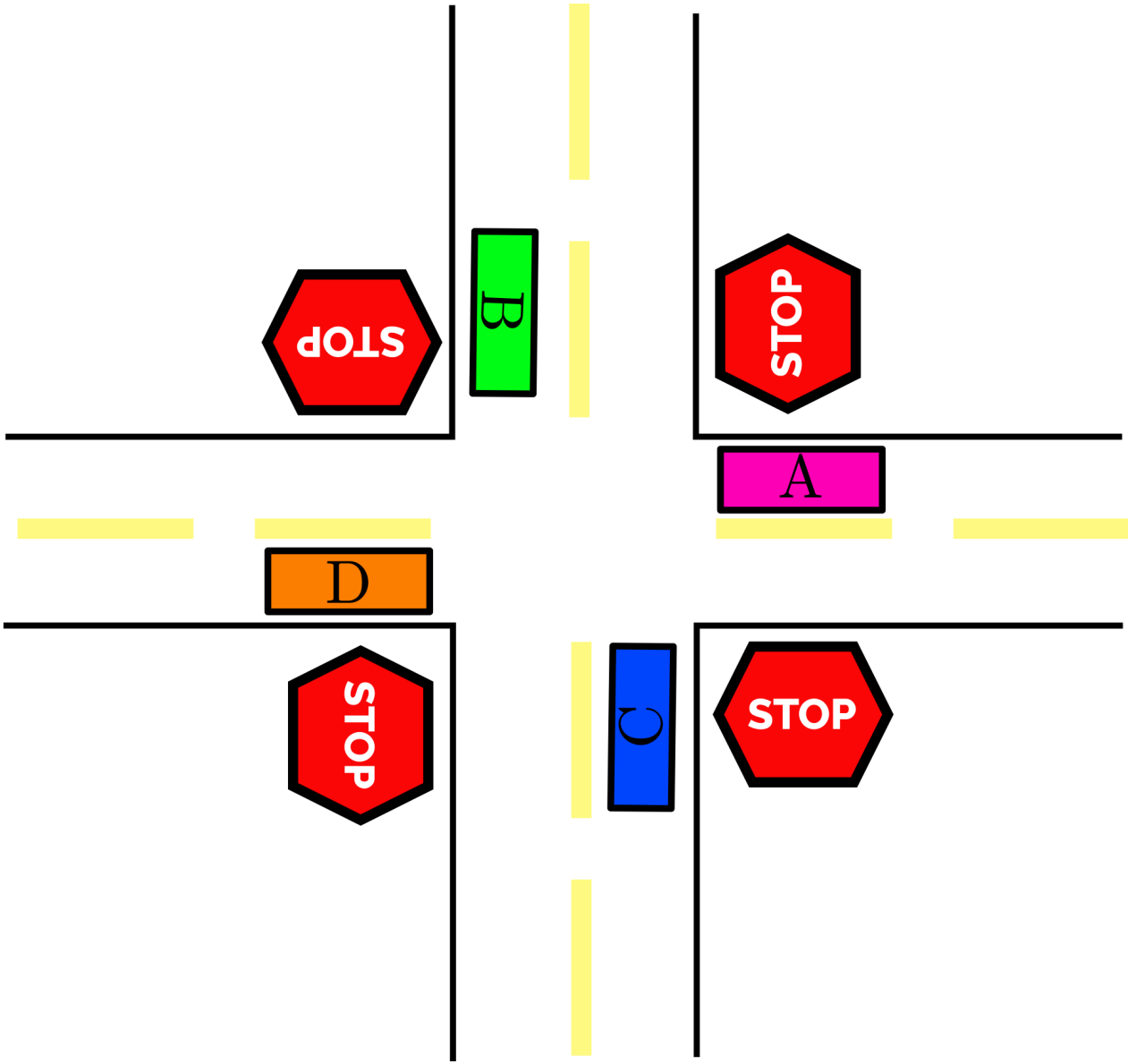
Who goes?



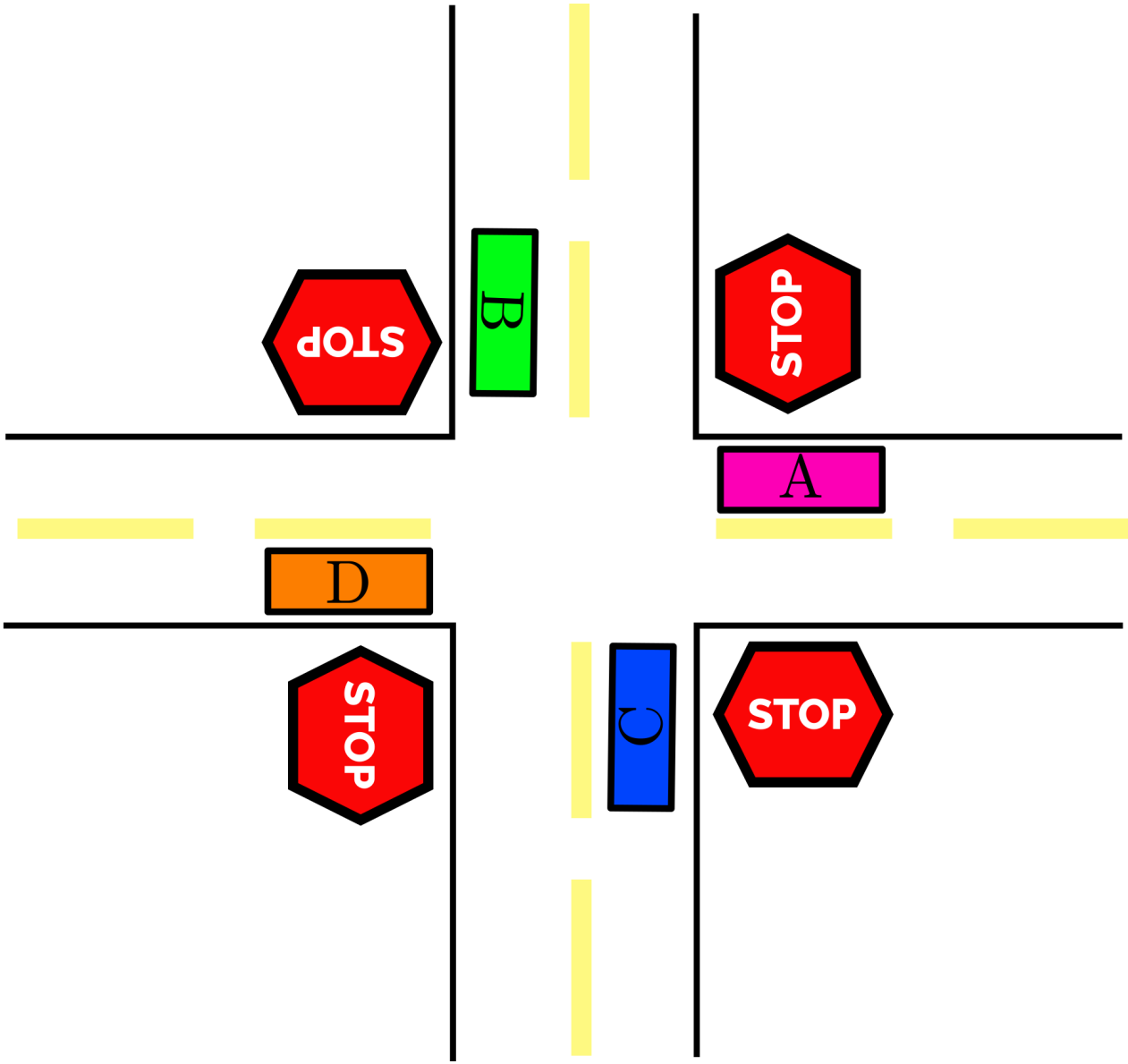

STOP
STOP
A
B
STOP
STOP


STOP
STOP
A
STOP
B
STOP


STOP
STOP
STOP
STOP

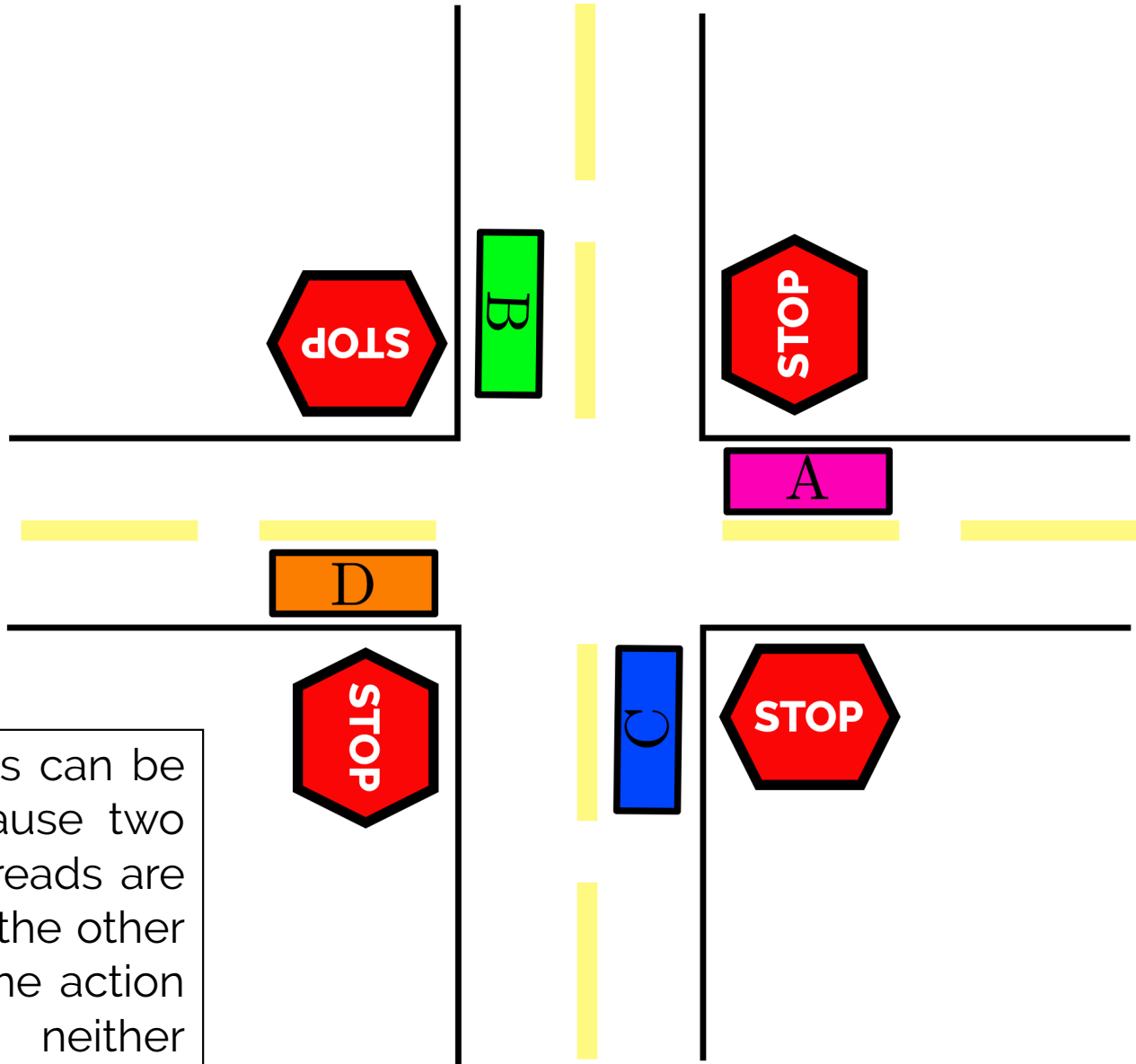

STOP
B
STOP
A
D
STOP
C
STOP

Who goes?



Who goes?

Deadlock



No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

Example: Simple deadlock

Thread 1 [RUNNING]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNABLE]:

```
lock(&B);
lock(&A);
```

Can deadlock happen with these two threads?

Example: Simple deadlock

Thread 1 [RUNNING]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNABLE]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [RUNNABLE]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNING]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [RUNNABLE]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNING]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [RUNNING]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNABLE]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [SLEEP]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNABLE]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [SLEEP]:

```
lock(&A);
lock(&B);
```

Thread 2 [RUNNING]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [SLEEP]:

```
lock(&A);
lock(&B);
```

Thread 2 [SLEEP]:

```
lock(&B);
lock(&A);
```

Example: Simple deadlock

Thread 1 [SLEEP]:

```
lock(&A);
lock(&B);
```

Thread 2 [SLEEP]:

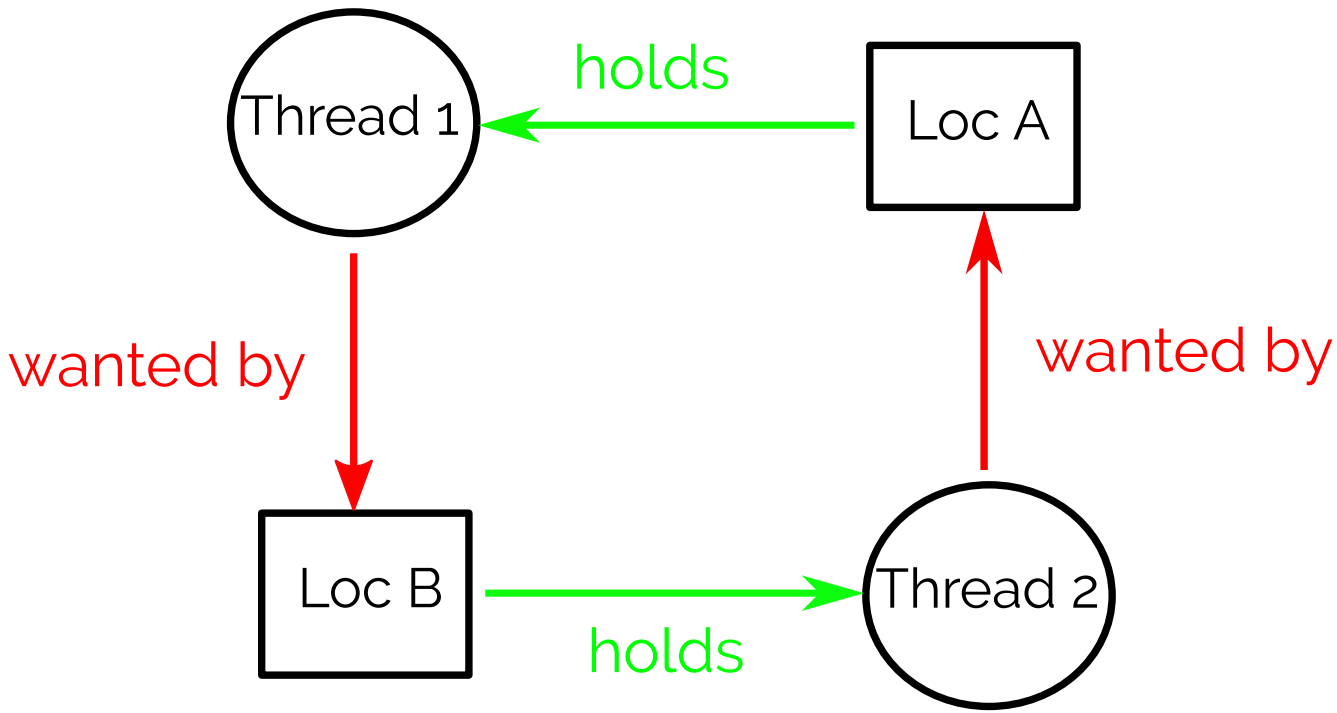
```
lock(&B);
lock(&A);
```

Deadlock

Stop & play:

Demo

Circular Dependency



Example: Simple deadlock

Thread 1 [RUNNING]:

```
lock(&A);
lock(&B);
```

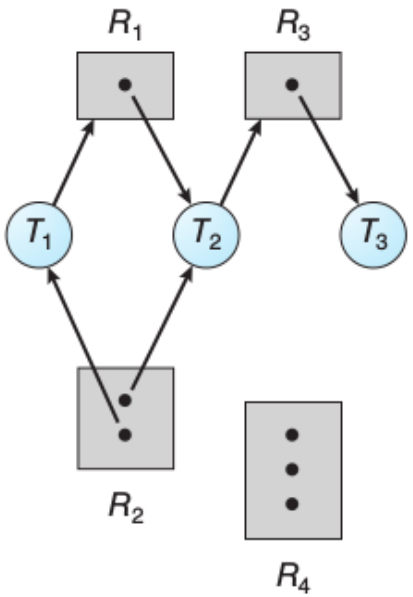
Thread 2 [RUNNABLE]:

```
lock(&A);
lock(&B);
```

Cann't deadlock

Resource-Allocation Graph

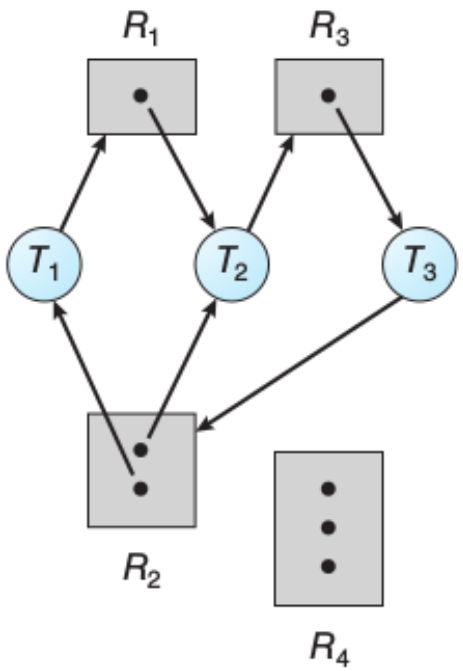
- Nodes: resources (rectangles), threads (circles)
  - Single instance resources & Multiple instance resources
- Directed edges: assignments (R→T), requests (T→R)



Necessary Condition for deadlock: cycle

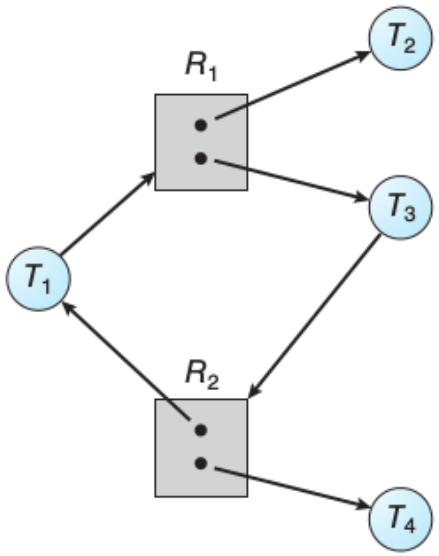
In single-instance resources conditions, cycle is also sufficient

Dynamics of the resource allocation graph: thread T3 requests an instance of resource type R2



Class Question

Does the following graph have a deadlock?



What's Wrong?

```
1  set_t *set_intsec(set_t *s1, set_t *s2) {
2    set_t *rv= Malloc(sizeof(*rv));
3    Mutex_lock(&s1->lock);
4    Mutex_lock(&s2->lock);
5    for(int i=0; i<s1->len; i++)
6      if(set_contains(s2, s1->items[i])
7        set_add(rv, s1->items[i]);
8    Mutex_unlock(&s2->lock);
9    Mutex_unlock(&s1->lock);
10   return rv;
11 }
```

Encapsulation

Modularity can make it harder to see deadlocks.

Thread 1 :

```
rv = set_intsec(A, B);
```

Thread 2:

```
rv = set_intsec(B, A);
```

Solution?

Encapsulation

Modularity can make it harder to see deadlocks.

Thread 1 :

```
rv = set_intsec(A, B);
```

Thread 2:

```
rv = set_intsec(B, A);
```

Solution?

```
1// Code assumes m1 != m2 (not same lock)
2if (m1 > m2) {
3 // grab locks in high-to-low address order
4   pthread_mutex_lock(m1);
5   pthread_mutex_lock(m2);
6}
7else {
8   pthread_mutex_lock(m2);
9   pthread_mutex_lock(m1);
10}
```

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Deadlock Theory

Deadlocks can only happen with these four conditions:

- ~~mutual exclusion~~
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Mutual Exclusion

Definition (Mutual Exclusion)

Threads claim exclusive control of resources that they require (e.g., thread grabs a lock).

Lock-Free Algorithms

Strategy: eliminate lock use.

Assume we have:

```
int CompAndSwap(int *addr, int expected, int new)
0: fail, 1: success
```

```
void add_v1(int *val, int amt)
{
  Mutex_lock(&m);
  *val += amt;
  Mutex_unlock(&m);
}
```

```
void add_v2(int *val, int amt) {
  do {
  int old = *val;
  } while(!CompAndSwap(val, old,
  old+amt);
}
```

Lock-Free Algorithms

Strategy: eliminate lock use.

Assume we have:

```
int CompAndSwap(int *addr, int expected, int new)
0: fail, 1: success
```

eliminate the lock!

```
void insert(int val) {
  node_t *n = Malloc(sizeof(*n));
  n->val = val;
  lock(&m);
  n->next = head;
  head = n;
  unlock(&m);
}
```

Lock-Free Algorithms

Strategy: eliminate lock use.

Assume we have:

```
int CompAndSwap(int *addr, int expected, int new)
0: fail, 1: success
```

```
void insert(int val) {
  node_t *n = Malloc(sizeof(*n));
  n->val = val;
  do {
  n->next = head;
  } while (!CompAndSwap(&head, n->next, n));
}
```

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- ~~hold-and-wait~~
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Hold-and-Wait

Definition (Hold-and-Wait)

Threads hold resources allocated to them (e.g., locks they have already acquired) while waiting for additional resources (e.g., locks they wish to acquire).

Eliminate Hold-and-Wait

Strategy: acquire all locks atomically **once**! (cannot acquire again until all have been released).

For this, use a meta lock, like this:

```
1 pthread_mutex_lock(prevention);     // begin
2                                     // acquisition
3 pthread_mutex_lock(L1);
4 pthread_mutex_lock(L2);
5 ...
6 pthread_mutex_unlock(prevention); // end
```

Disadvantages?

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- ~~no preemption~~
- circular wait

Eliminate deadlock by eliminating one condition.

No preemption

Definition (No preemption)

Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.

Support Preemption

Strategy: if we can't get what we want, release what we have.

```
1 top:
2 pthread_mutex_lock(L1);
3 if (pthread_mutex_trylock(L2) != 0) {
4   pthread_mutex_unlock(L1);
5   goto top;
6 }
```

Livelock ...

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- ~~circular wait~~

Eliminate deadlock by eliminating one condition.

Circular wait

Definition (Circular Wait)

There exists a circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

Eliminating Circular Wait

Strategy:

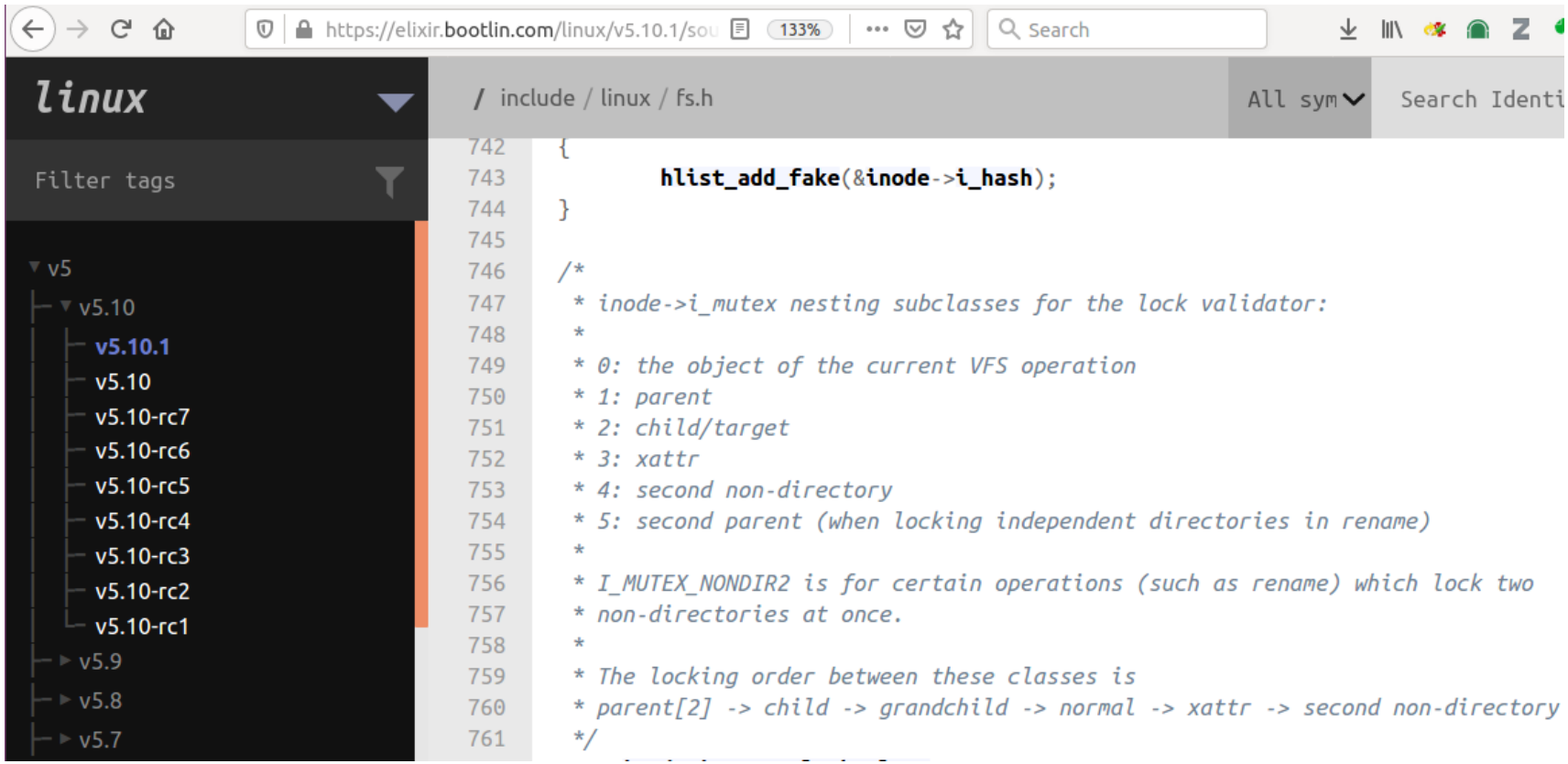
1. decide which locks should be acquired before others

   if A before B, never acquire A if B is already held!

   document this, and write code accordingly

2. Each thread can request resources only in an increasing order of enumeration.

   That is, a thread can initially request an instance of a resource –say, Ri . After that, the thread can request an instance of resource Rj if and only if F(Rj) > F(Ri).
   F can be a hash function
   Proof


https://elixir.bootlin.com/linux/v5.10.1/sou
133%
Search
linux
/ include / linux / fs.h
All sym
Search Identi
Filter tags
v5
v5.10
v5.10.1
v5.10
v5.10-rc7
v5.10-rc6
v5.10-rc5
v5.10-rc4
v5.10-rc3
v5.10-rc2
v5.10-rc1
v5.9
v5.8
v5.7
{
hlist_add_fake(&inode->i_hash);
}
/*
* inode->i_mutex nesting subclasses for the lock validator:
*
* 0: the object of the current VFS operation
* 1: parent
* 2: child/target
* 3: xattr
* 4: second non-directory
* 5: second parent (when locking independent directories in rename)
*
* I_MUTEX_NONDIR2 is for certain operations (such as rename) which lock two
* non-directories at once.
*
* The locking order between these classes is
* parent[2] -> child -> grandchild -> normal -> xattr -> second non-directory
*/

```
/*
 * inode->i_mutex nesting subclasses for the lock validator:
 *
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: xattr
 * 4: second non-directory
 * 5: second parent (when locking independent directories in rename)
 *
 * I_MUTEX_NONDIR2 is for certain operations (such as rename) which lock two
 * non-directories at once.
 *
 * The locking order between these classes is
 * parent[2] -> child -> grandchild -> normal -> xattr -> second non-directory
 */
```

Detect & Recover

One final general strategy is to **allow deadlocks to occasionally occur** ☺

Many database systems employ deadlock detection and recovery techniques.

A deadlock detector runs periodically, building a resource graph and checking it for cycles.

In the event of a cycle (deadlock), the system needs to be restarted.

سوال ۵:

در مساله ضیافت فیلسوفان فرض کنید که چیدمان میز تغییر کرده و تمام چنگال‌ها در یک ظرف در وسط میز قرار داده شده است. در این صورت اگر تعداد فیلسوفان n باشد، حداقل تعداد چنگال‌ها چه تعداد باشد تا بن بست رخ ندهد؟ (بیان اثبات الزامی است)

Edsger Wybe Dijkstra (1930-2002)

Turing Award

Specially for this paper: "Cooperating sequential processes",1968

Discovered much of this while working on the "THE" operating system

THE: Technische Hochshule of Eindhoven (THE)