Sharing with Theads

Try changing **`t_echo.c`** to count total bytes:

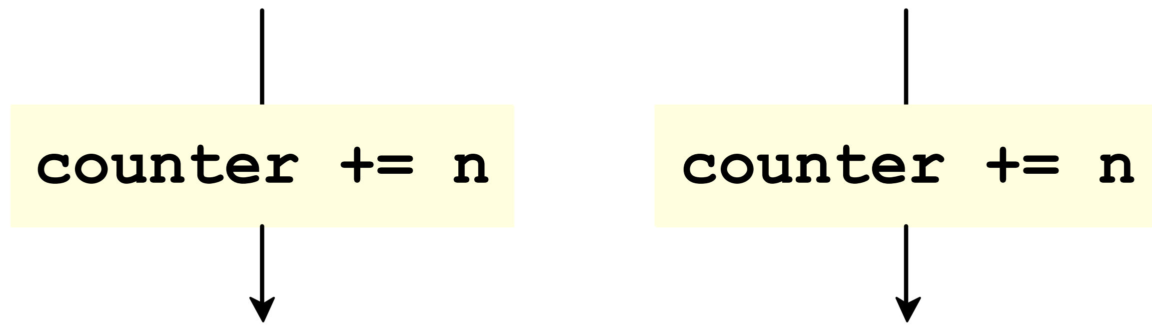
`t_echo.c`

```
....
static size_t counter = 0;

int main() {
  ....
  Pthread_create(&th, NULL, echo, connfd_p);
  ....
}

void *echo(void *connfd_p) {
  ....
  while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    // printf("server received %ld bytes\n", n);
    counter += n;
    Rio_writen(connfd, buf, n);
  }
  printf("total bytes so far: %ld\n", counter);
  ....
}
```

Concurrent Variable Updates



Problem: the program has a ***race condition***

Two threads race to update `counter`

Concurrent Variable Updates

```
movl   <counter>, %rdx
movl   <n>, %rax
addl   %rdx, %rax
movl   %rax, <counter>
```

```
movl   <counter>, %rdx
movl   <n>, %rax
addl   %rdx, %rax
movl   %rax, <counter>
```

Concurrent Variable Updates

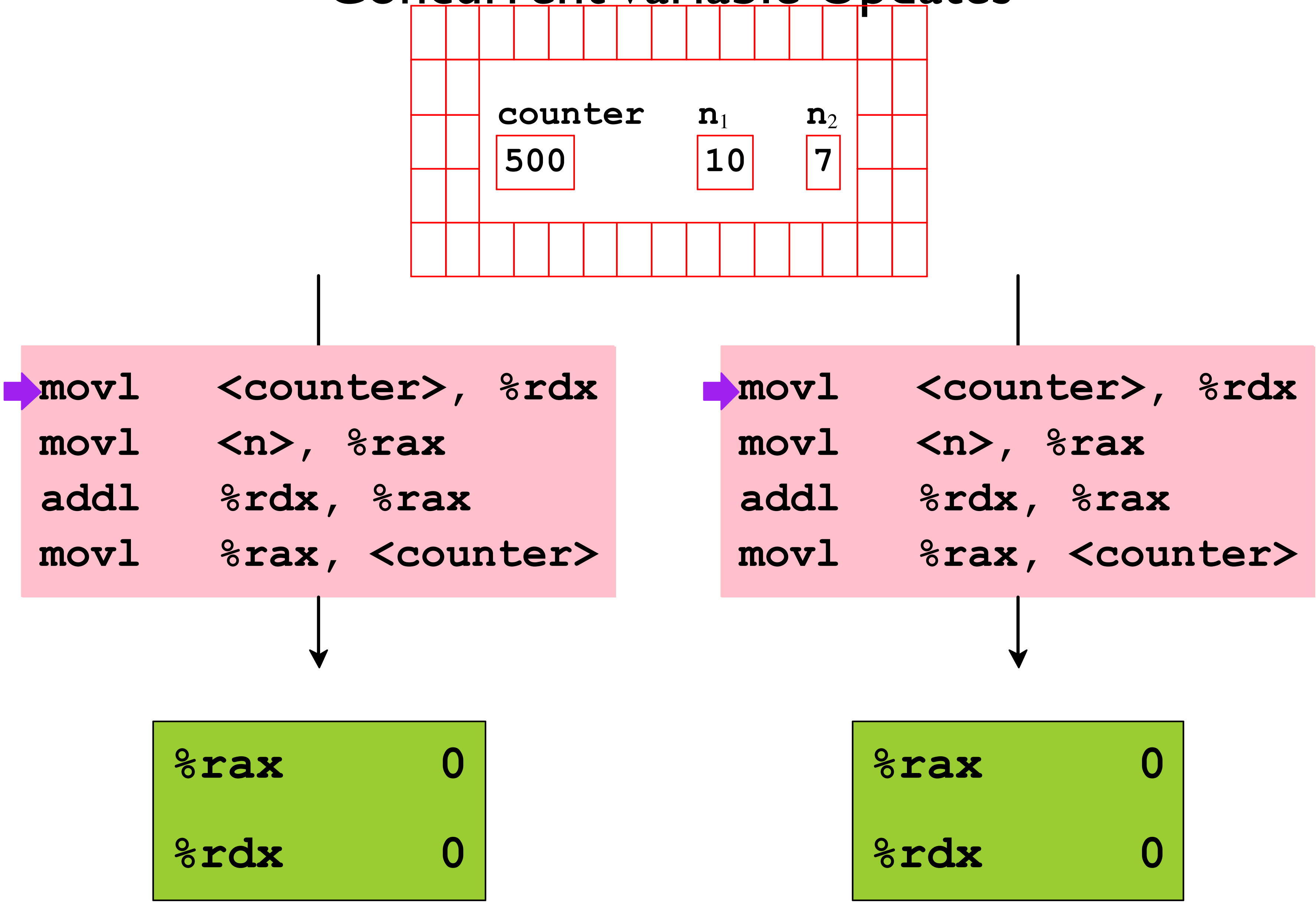
```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	0

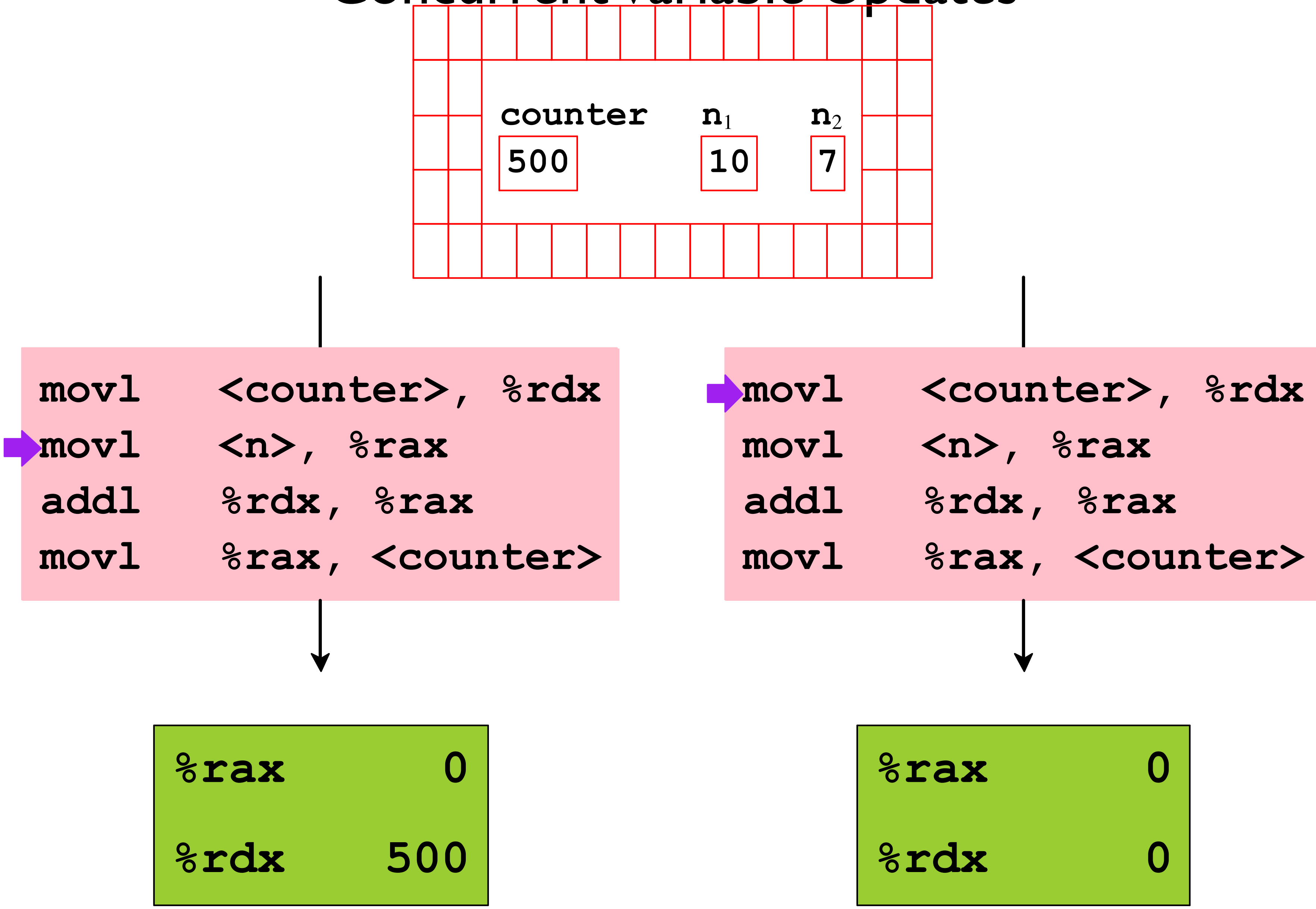
```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	0

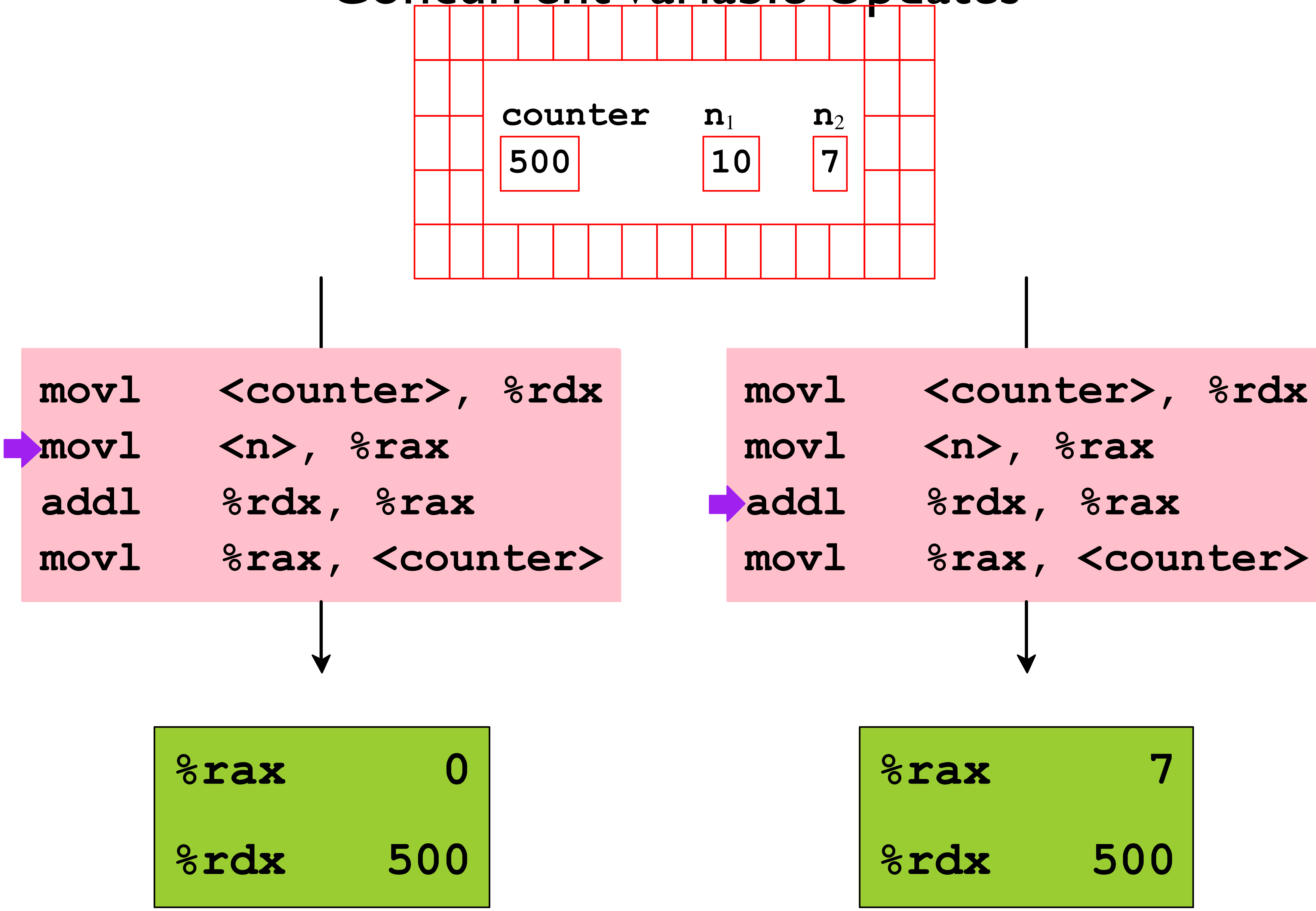
Concurrent Variable Updates



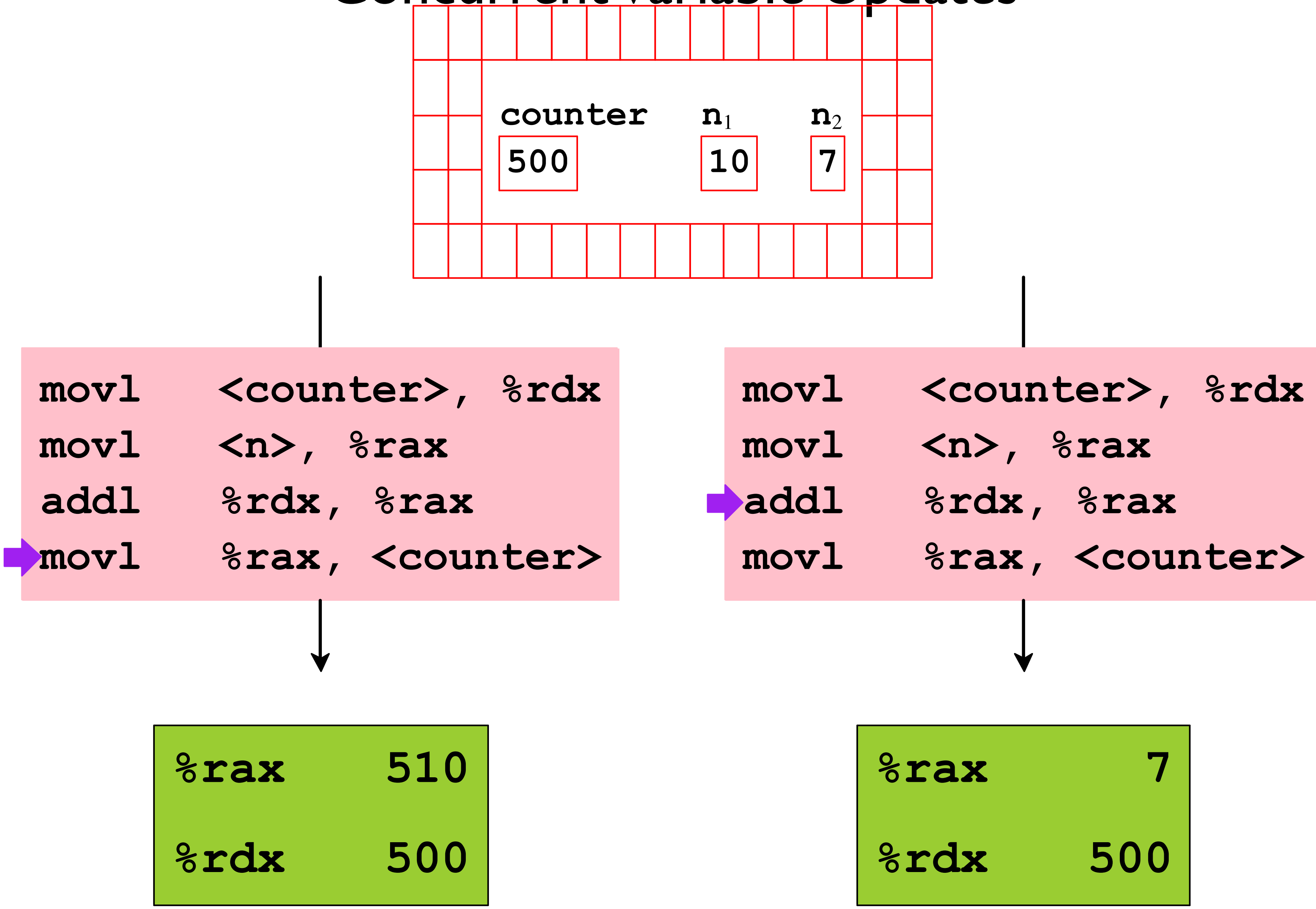
Concurrent Variable Updates



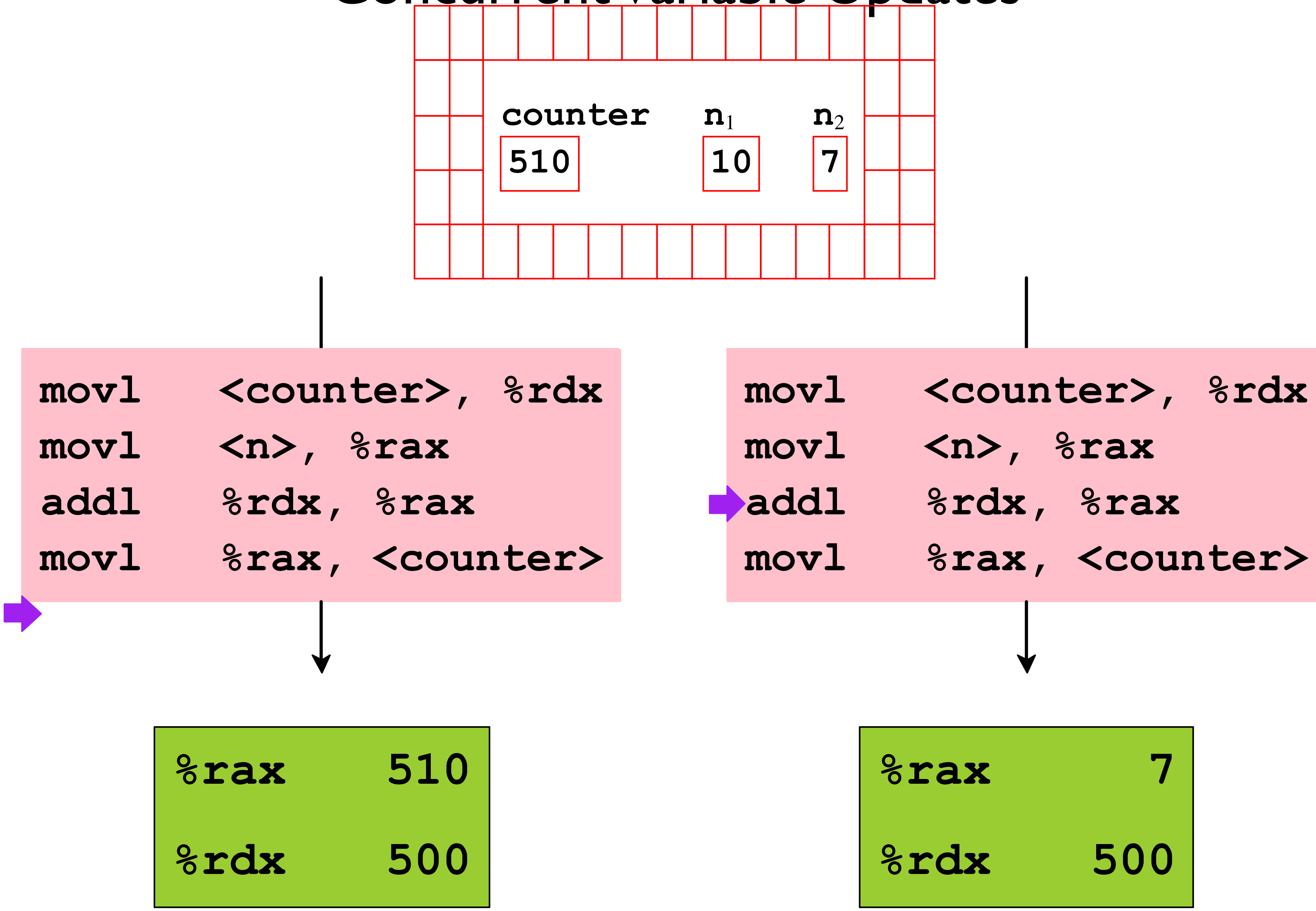
Concurrent Variable Updates



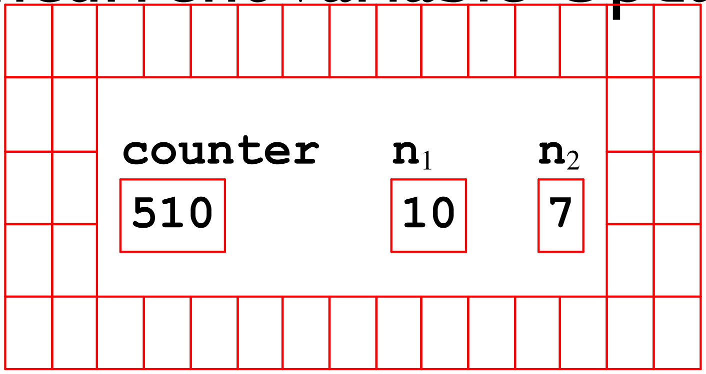
Concurrent Variable Updates



Concurrent Variable Updates



Concurrent Variable Updates



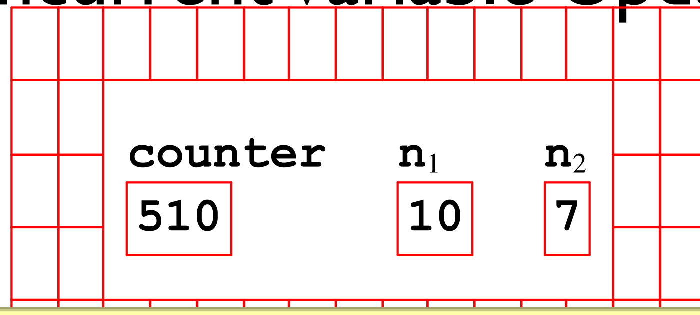

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	510
%rdx	500

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	507
%rdx	500

Concurrent Variable Updates



read–add–write sequence is not ***atomic***

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

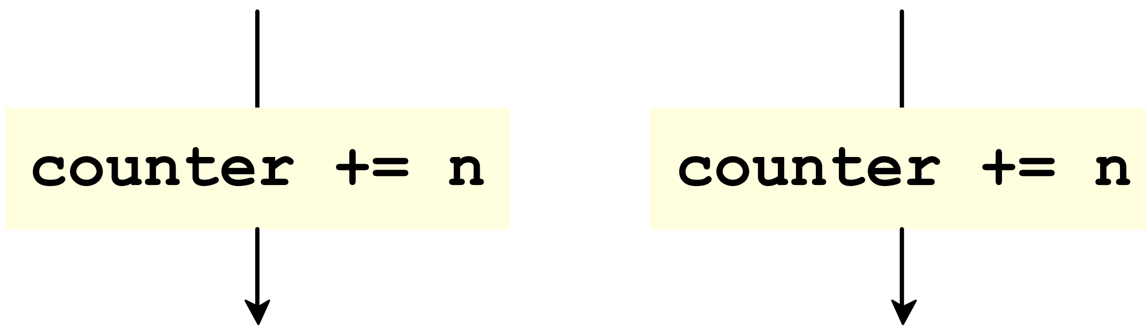
%rax	510
%rdx	500

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	507
%rdx	500

Concurrent Variable Updates

Try compiling with **-O2**



Concurrent Variable Updates

Try compiling with `-O2`

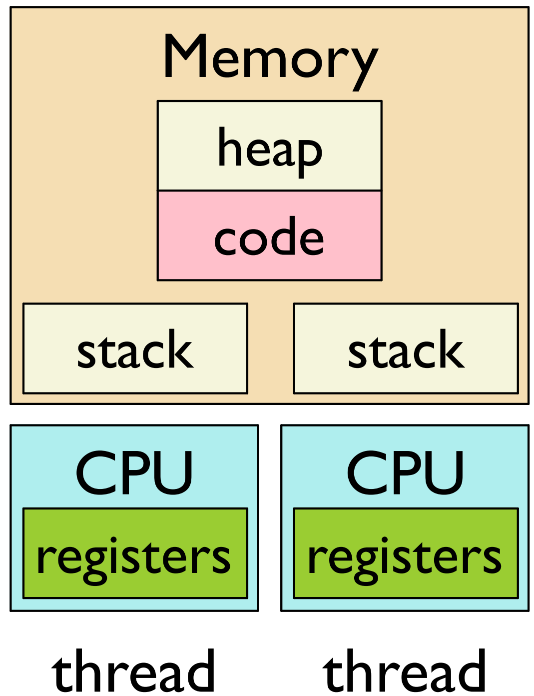
```
movl    <n>, %rax
addl    %rax, <counter>
```

```
movl    <n>, %rax
addl    %rax, <counter>
```

Doesn't work with a multiprocessor

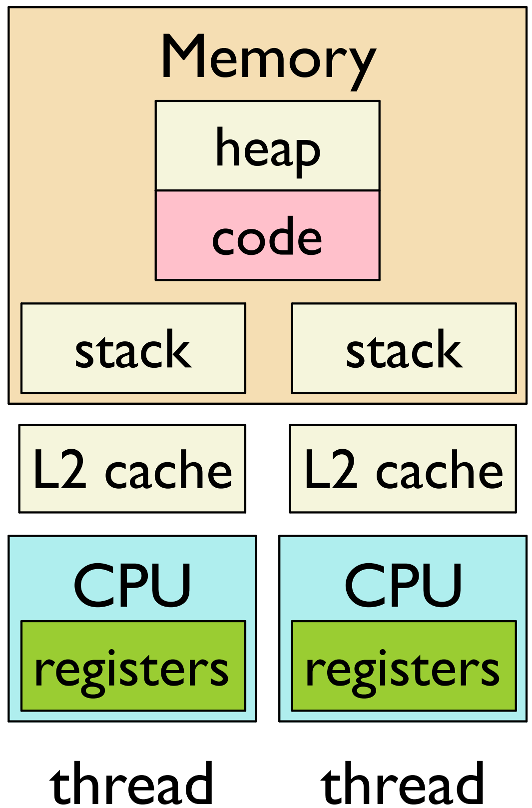
Threads and Processors

Intended illusion:



Threads and Processors

Observable behavior:



Cache coherence is expensive, so the machine just doesn't do it! ... unless you insist

Global Variables and Optimization

Remember that C compilers can make assumptions:

```
long counter = 1;

void count_to(long n) {
  while (counter < n)
   counter++;
}

void wait_for_it() {
  while (counter < 100000)
   ;
}

....
```

Global Variables and Optimization

Remember that C compilers can make assumptions:

```
long counter = 1;

void count_to(long n) {
  while (counter < n)
   counter++;
}

void wait_for_it() {
  while (counter < 100000)
   ;
}

....
```

```
long counter = 0;

void count(long n) {
  long v = counter;
  while (v < n)
    v++;
  counter = v;
}

void wait_for_it() {
  if (counter < 100000)
   while (1)
     ;
}

....
```

Threads and Sharing

Successful sharing among threads requires explicit synchronization

- ✓ Side-steps question of machine-code atomicity
- ✓ Declares need for cache coherence
- ✓ Exposes constraints to compiler

A program with a race condition is practically always a buggy program

Synchronization for Sharing

Several general approaches to sharing:

No sharing — pass messages, instead

✓ No one changes your data while you look at it

✗ Communication must be explicitly scheduled

Transactions — system finds a good ordering

✓ Programmer declares needed atomicity

✗ Requires substantial extra infrastructure

Locks — constrain allowed orders

✓ Almost like declaring atomicity

✗ Declare and using locks correctly is still difficult

Synchronization for Sharing

Several general approaches to sharing:

No sharing — pass messages, instead

✓ No one changes your data while you look at it

✗ Communication must be explicitly scheduled

Transactions — system finds a good ordering

✓ Programmer declares needed atomicity

✗ P

Most common, especially for systems programming

Locks — constrain allowed orders

✓ Almost like declaring atomicity

✗ Declare and using locks correctly is still difficult

Machine-Level Synchronization

`lock cmpxchg`*x* *source*`,` *dest*

Atomically checks whether **`%rax`** matches *dest* and

- if equal, copies *source* to *dest*, sets **`ZF`**
- if not equal, clears **`ZF`**

Atomicity means that if **`dest`** is a memory address, caches are forced to agree during the instruction

A.K.A. ***compare and swap*** (CAS)

Accessible in **`gcc`** via

`__sync_bool_compare_and_swap(`*addr*`,` *old_val*`,` *new_val***`)`**

Machine-Level Synchronization

```
#include "csapp.h"

volatile int counter;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++)
    counter++;

  return NULL;
}

int main(int argc, char **argv) {
  pthread_t a, b;
  int n = 30000;
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  Pthread_join(a, NULL);
  Pthread_join(b, NULL);
  printf("result: %d\n", counter);
}
```

Copy

Machine-Level Synchronization

```
#include "csapp.h"

volatile int counter;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++)
    counter++;

  return NULL;
}

int main(int argc, char **argv) {
  pthread_t a, b;
  int n = 30000;
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  Pthread_join(a, NULL);
  Pthread_join(b, NULL);
  printf("result: %d\n", counter);
}
```

Copy

`volatile` forces separate load and store on **`counter`**

Result is unspecified

Machine-Level Synchronization

CAS ensures a consistent result:

```
....

    int old_counter;
    do {
      old_counter = counter;
    } while (!__sync_bool_compare_and_swap(&counter,
                                           old_counter,
                                           old_counter+1));
....
```

Copy

CAS is too low-level for most purposes

- ✗ Failure is a form of busy waiting
- ✗ Sometimes, multiple values need to change together

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  counter++;
}
```

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one t

Only one thread should increment at a time

```
for (i   0; i < n; i++) {
  counter++;
}
```

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  lock();
  counter++;
  unlock();
}
```

`lock()` returns if currently unlocked, otherwise waits

`unlock()` only if previously **`lock()`**ed

`lock` and **`unlock`** are not actual function names...

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  lock();
  count = lookup(name);
  if (count < 10)
    update(name, count + 1);
  unlock();
}
```

`lock()` returns if currently unlocked, otherwise waits

`unlock()` only if previously **`lock()`**ed

Locking for Specific Data

Locks can be more ***fine-grained***, such as locking specific object instead of a section of code

```
for (i = 0; i < n; i++) {
  lock(locks[i]);
  count = lookup(orders[i], name);
  if (count < 10)
    update(orders[i], name, count + 1);
  unlock(locks[i]);
}
```

Locking as a Signaling Mechanism

Since **`lock()`** waits for another thread's **`unlock()`**, locks can effectively send a "signal" from one thread to another

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  ....
  value = 1;
  unlock(ready_lock);
  ....
}

void *go(void *ignored) {
  lock(ready_lock);
  .... value ....
}
```

Locking as a Signaling Mechanism

Since **`lock()`** waits for another thread's **`unlock()`**, locks can effectively send a "signal" from one thread to another

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  ....
  value = 1;
  unlock(ready_lock);
  ....
}

void *go(void *ignored) {
  lock(ready_lock);
  .... value ....
}
```

Cannot proceed until main thread gets to **`unlock`**

Locking as a Signaling Mechanism

If **`unlock()`** doesn't have to be in the **`lock()`** thread, signaling can work the other way, too

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  lock(ready_lock);
  .... value ....
}

void *go(void *ignored) {
  value = 1;
  unlock(ready_lock);
  ....
}
```

Locking as a Signaling Mechanism

If **`unlock()`** doesn't have to be in the **`lock()`** thread, signaling can work the other way, too

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  lock(ready_lock);
  .... value ....
}

void *go(void *ignored) {
  value = 1;
  unlock(ready_lock);
  ....
}
```

Cannot proceed until new thread gets to **`unlock`**

Kinds of Locks

Mutex

- **`pthread_mutex_t`**
- **`pthread_mutex_init(`***mutex***`,`** ***attr*****`)`**
- **`pthread_mutex_lock(`***mutex***`)`**
- **`pthread_mutex_unlock(`***mutex***`)`**

`...lock()` and balancing **`...unlock()`** must be same thread

Semaphore

- **`sem_t`**
- **`Sem_init(`***sem***`,`** ***ps_share*****`,`** ***value*****`)`**
- **`P(`***sem***`)`** = **`lock()`**, but with a counter
- **`V(`***sem***`)`** = **`unlock()`**, with the counter

`P()` and balancing **`V()`** threads can be different

Kinds of Locks

Mutex

- `pthread_mutex_t`
- `pthread_mutex_init(`*mutex*`,` *attr*`)`
- `pthread_mutex_lock(`*mutex*`)`
- `pthr`

`...lock`

Sometimes, we create a semaphore and name it **`mutex`**, because it's used that way

Semaphore

- `sem_t`
- `Sem_init(`*sem*`,` *ps_share*`,` *value*`)`
- `P(`*sem*`)` = `lock()`, but with a counter
- `V(`*sem*`)` = `unlock()`, with the counter

`P()` and balancing `V()` threads can be different

Semaphores

```
#include "csapp.h"

void Sem_init(sem_t *sem, int ps_share, unsigned int value);
void P(sem_t *sem);
void V(sem_t *sem);
void Sem_destroy(sem_t *sem);
```

`Sem_init` creates **`sem`** with initial count **`value`**

`1` as **`value`** for a mutex

0 as **`ps_share`**

`P` waits until **`sem`** has a non-0 count, then decrements

corresponds to **`lock`**, also called “wait”

`V` increments **`sem`**’s count

corresponds to **`unlock`**, also called “post”

`Sem_destroy` destroys **`sem`**

Semaphore Example

```
....
sem_t count_sem;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++) {
    P(&count_sem);
    counter++;
    V(&count_sem);
  }

  return NULL;
}

int main(int argc, char **argv) {
  ....
  Sem_init(&count_sem, 0, 1);
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  .....
}
```

Copy

Semaphores for Echo

t_echo.c

```
....
sem_t ready_sem, count_sem;

int main(int argc, char **argv) {
  ....
  Sem_init(&count_sem, 0, 1);
  Sem_init(&ready_sem, 0, 0);
  ....
     Pthread_create(&th, NULL, echo, &connfd);
     P(&ready_sem);
  ....
}

void *echo(void *connfd_p) {
  ....
  V(&ready_sem);
  ....
    P(&count_sem);
    counter += n;
    V(&count_sem);
  ....
}
```

Semaphores as Per-Object Locks

counter.c

```
typedef struct {
  int val;
  sem_t sem;
} counter;

counter *make_counter() {
  counter *c = malloc(sizeof(counter));
  c->val = 0;
  Sem_init(&c->sem, 0, 1);
  return c;
}

void counter_add(counter *c, int amt) {
  P(&c->sem);
  c->val += amt;
  V(&c->sem);
}
....
void destroy_counter(counter *c) {
  Sem_destroy(&c->sem);
  free(c);
}
```

Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

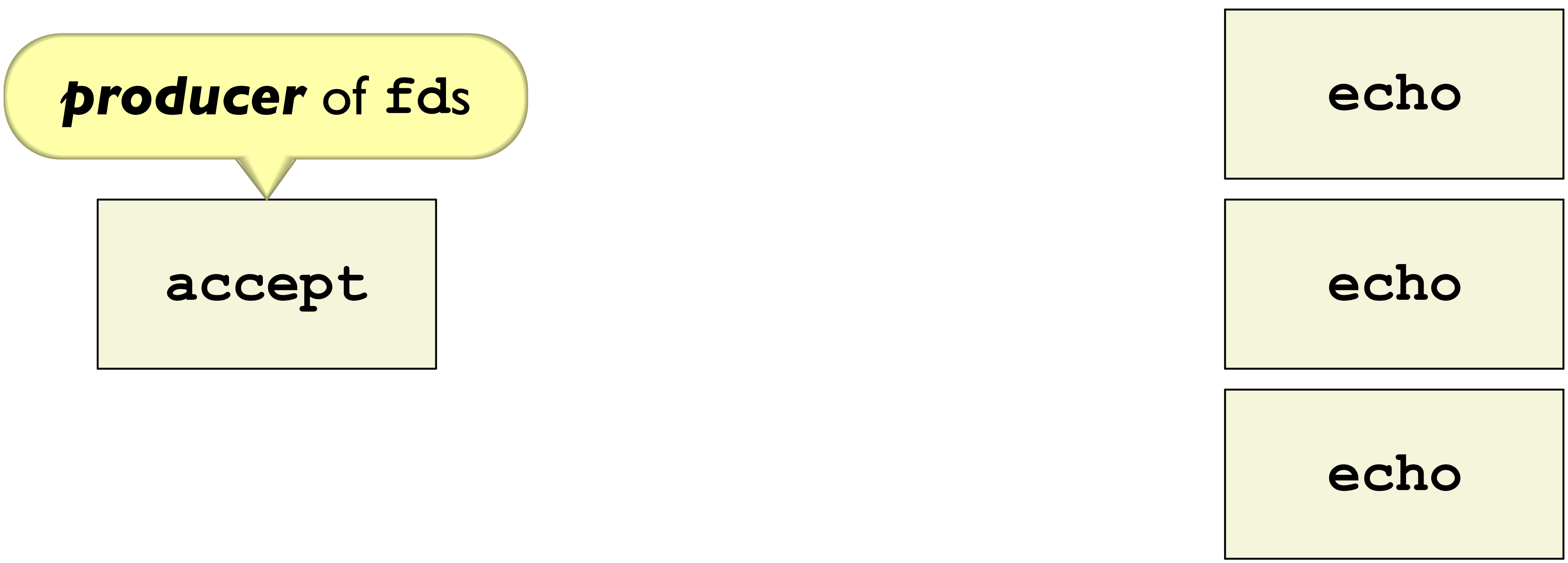
- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connection

Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connection

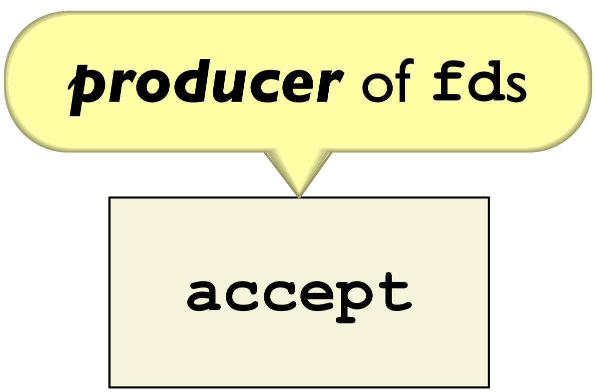


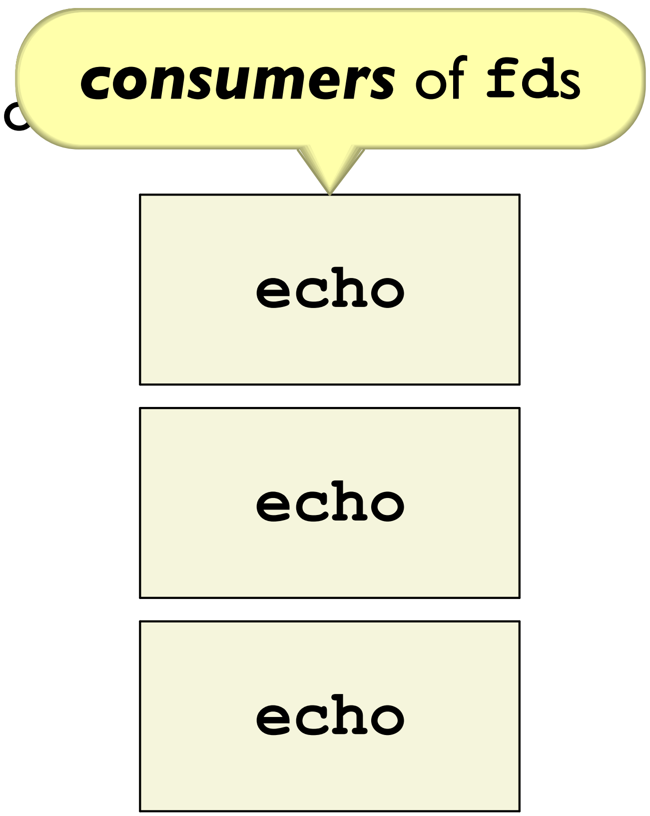
Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio



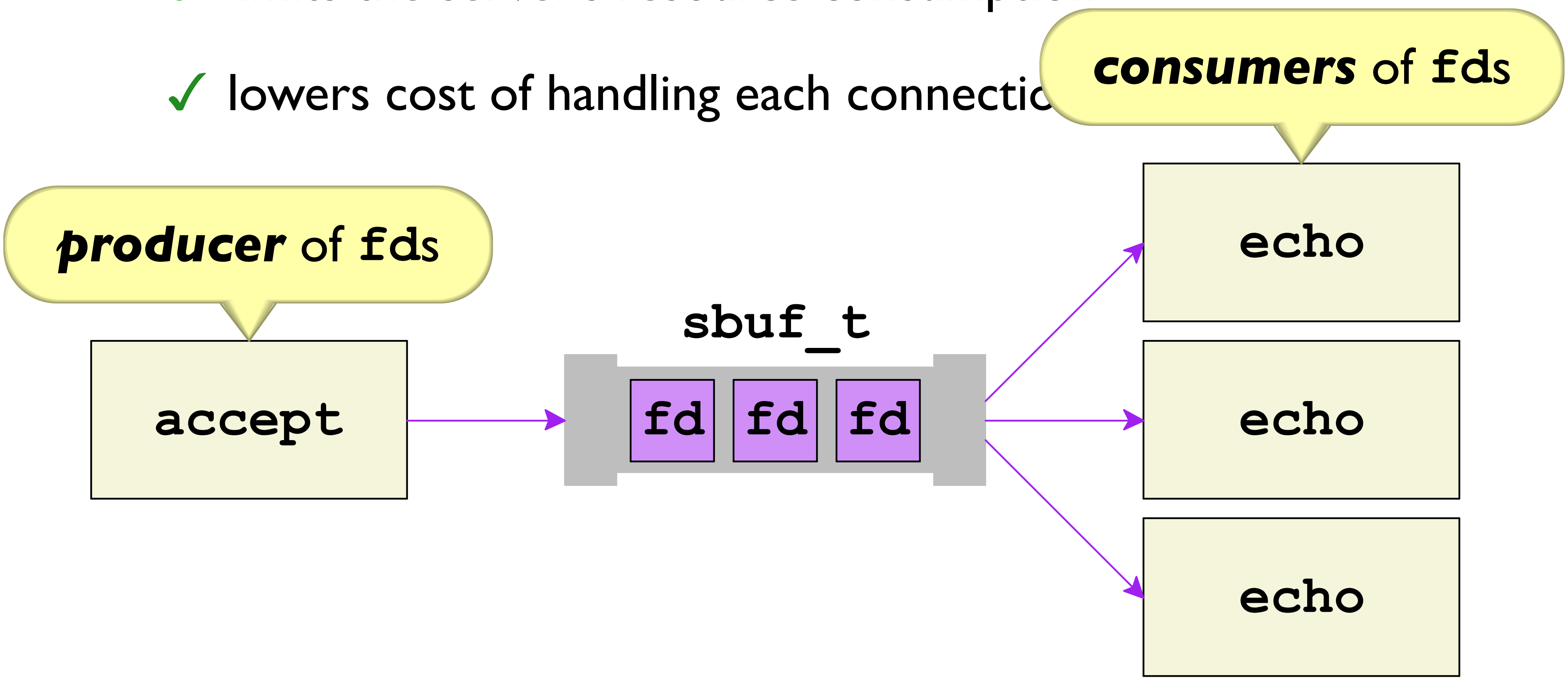


Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio

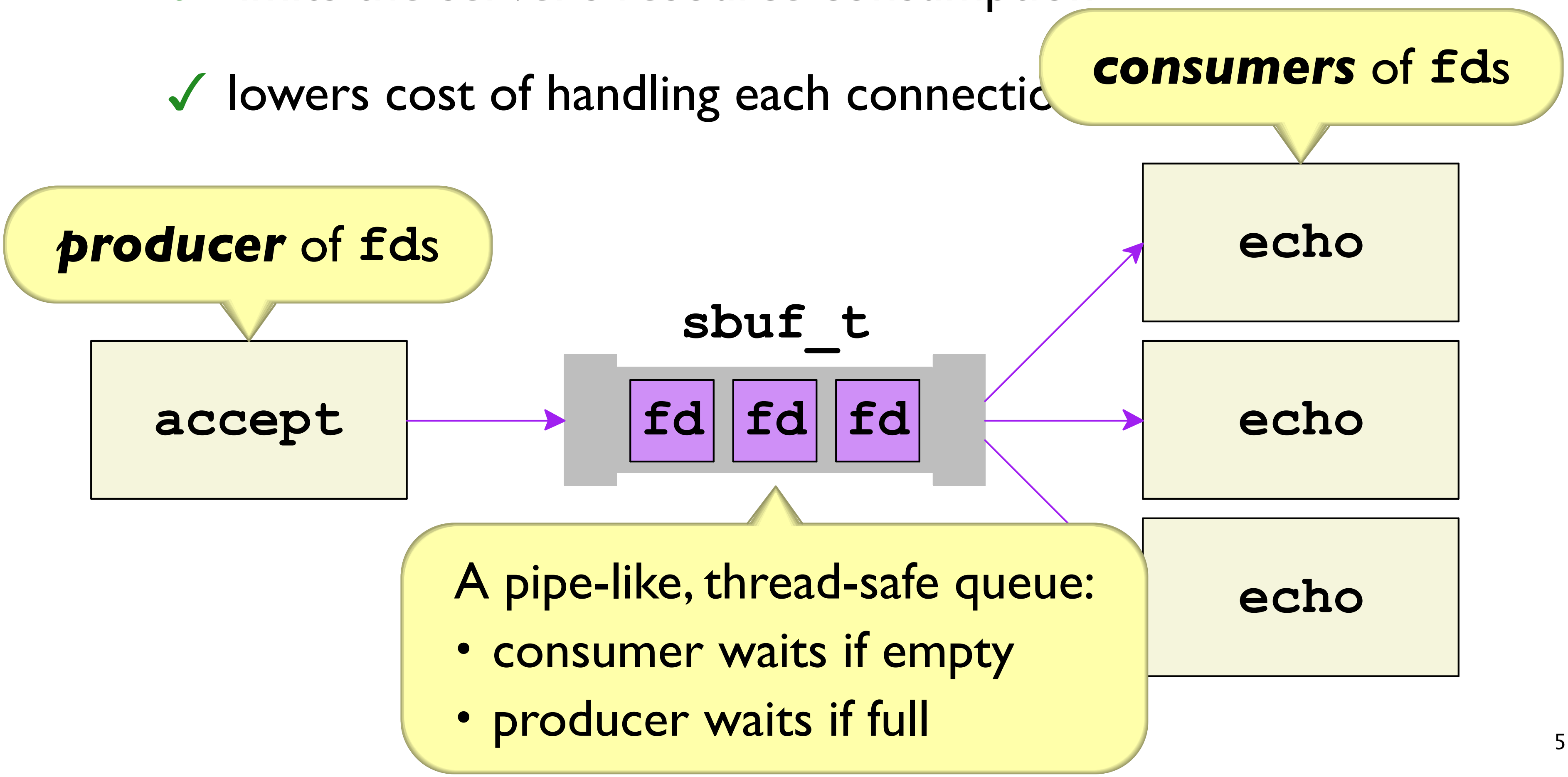


Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio



Implementing a Limited Queue with Semaphores

Strategy: use semaphore count to reflect availability

- **`sbuf_insert`** (for producer) — count is available slots
- **`sbuf_remove`** (for consumer) — count is available values

⇒ two counter semaphores, plus one as a mutex

Implementing a Limited Queue with Semaphores

sbuf.h

```
 typedef struct {
    int *buf;       /* Buffer array */
    int n;          /* Maximum number of slots */
    int front;      /* buf[(front+1)%n] is first item */
    int rear;       /* buf[rear%n] is last item */
    sem_t mutex;    /* Protects accesses to buf */
    sem_t slots;    /* Counts available slots */
    sem_t items;    /* Counts available items */
} sbuf_t;
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

void sbuf_init(sbuf_t *sp, int n) {
  sp->buf = Calloc(n, sizeof(int));
  sp->n = n;                      /* max of n items */
  sp->front = sp->rear = 0;   /* empty iff front == rear */
  Sem_init(&sp->mutex, 0, 1); /* for locking */
  Sem_init(&sp->slots, 0, n); /* initially n empty slots */
  Sem_init(&sp->items, 0, 0); /* initially zero data items */
}

....
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

void sbuf_insert(sbuf_t *sp, int item) {
  P(&sp->slots);    /* wait for available slot */
  P(&sp->mutex);    /* lock */
  sp->buf[(++sp->rear)%(sp->n)] = item;
  V(&sp->mutex);    /* unlock */
  V(&sp->items);    /* announce available item */
}

....
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

int sbuf_remove(sbuf_t *sp) {
  int item;
  P(&sp->items);  /* wait for available item */
  P(&sp->mutex);  /* lock */
  item = sp->buf[(++sp->front)%(sp->n)];
  V(&sp->mutex);  /* unlock */
  V(&sp->slots);  /* announce available slot */
  return item;
}

....
```

Producer–Consumer Echo Server

`pc_echo.c`

```
 ....
 sbuf_t connfds;

 int main(int argc, char **argv) {
   ....
   sbuf_init(&connfds, SBUF_SIZE);

   for (i = 0; i < NUM_THREADS; i++) {
     Pthread_create(&th, NULL, echo, NULL);
     Pthread_detach(th);
   }
   ....
     connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
     sbuf_insert(&connfds, connfd);
   ....
}
....
```

Producer–Consumer Echo Server

pc_echo.c

```
....

void *echo(void *ignored) {
  ....
  while (1) {
    connfd = sbuf_remove(&connfds);

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
      printf("server received %ld bytes\n", n);
      Rio_writen(connfd, buf, n);
    }

    Close(connfd);
  }
}
```

Threads and **`errno`**

Suppose one thread is running

```
fd = open(...);
if (fd < 0)
  fprintf(stderr, "%d", errno);
```

and another is running

```
fd = connect(...);
if (fd < 0)
  fprintf(stderr, "%d", errno);
```

Can the **`open`** thread get the **`errno`** value for **`connect`**?

No, **`errno`** is ***thread-local***

Whew!

Thread-Safe Functions

Standard library functions are generally ***thread-safe***

OK in multiple threads:

- **`malloc`** and **`free`**
- **`read`** on the same file descriptor
- **`fread`** on the same file handle
- **`getaddrinfo`** to fill different records

Not OK in multiple threads:

- **`getenv`** when **`setenv`** might be called
- **`rio_readnb`** on a specific buffer

Concurrency vs. Parallelism

Concurrency = multiple control flows overlapping in time

possibly on a uniprocessor

reduces ***latency***

Parallelism = multiple control flows at the same time

requires a multiprocessor

can improve ***throughput***

parallelism $\Rightarrow$ concurrency   concurrency $\not\Rightarrow$ parallelism