```c
int TestAndSet(int * ptr, int new_val)
{
  int old_val = *ptr;
  *ptr = new_val;
  return old_val;
}

int lock = 0;
int counter = 0;

void* increment_counter(void* arg)
{
  printf("Thread %s starting\n", (char*)arg);
  for (int i = 0; i < 100000; i++) {
    while (TestAndSet(&lock, 1) == 1);
    counter++;
    lock = 0;
  }
  return NULL;
}

int main()
{
  pthread_t t1, t2;
  pthread_create(&t1, NULL, increment_counter, "Red");
  pthread_create(&t2, NULL, increment_counter, "Blue");
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Final counter value: %d\n", counter);
  return 0;
}
```

spin locks

- correctness - ✓
- performance - X

a thread acquires lock --> preempted -->
some other thread acquires lock - spins - spins --> preempted -->
some other thread acquires lock - spins - spins --> preempted -->
some other thread acquires lock - spins - spins --> preempted -->
a thread which had lock will be scheduled --> releases lock!

```c
atomic_flag lock = ATOMIC_FLAG_INIT;
int counter = 0;
void acquire_lock(atomic_flag *lock) {
  while (atomic_flag_test_and_set(lock)) {
    sched_yield();
  }
}

void release_lock(atomic_flag *lock) {
  atomic_flag_clear(lock);
}

void* increment_counter(void* arg) {
  for (int i = 0; i < 100000; i++) {
    acquire_lock(&lock);
    counter++;
    release_lock(&lock);
  }
  return NULL;
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, increment_counter, "Red");
  pthread_create(&t2, NULL, increment_counter, "Blue");
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Final counter value: %d\n", counter);
  return 0;
}
```

---

fairness? - starvation
: depends on which thread gets scheduled

---

spin locks

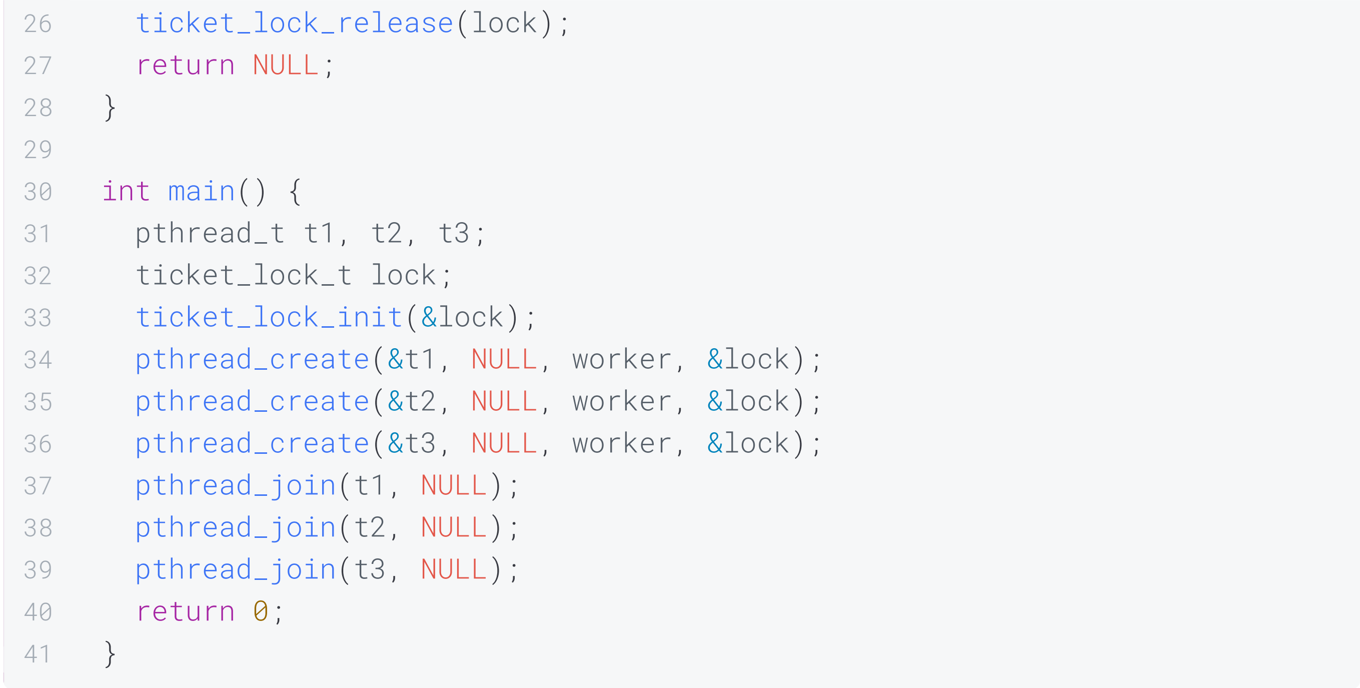
T2 $>_{\text{priority}}$ T1
T1 - acquires lock
T2 spins forever :(

not spin locks

T3 > T2 > T1
T1 - acquires lock
T3 - tries to acquire lock - waits
T2 - starts running
but since T2 is running, T1 will never get a chance
T3's priority :(

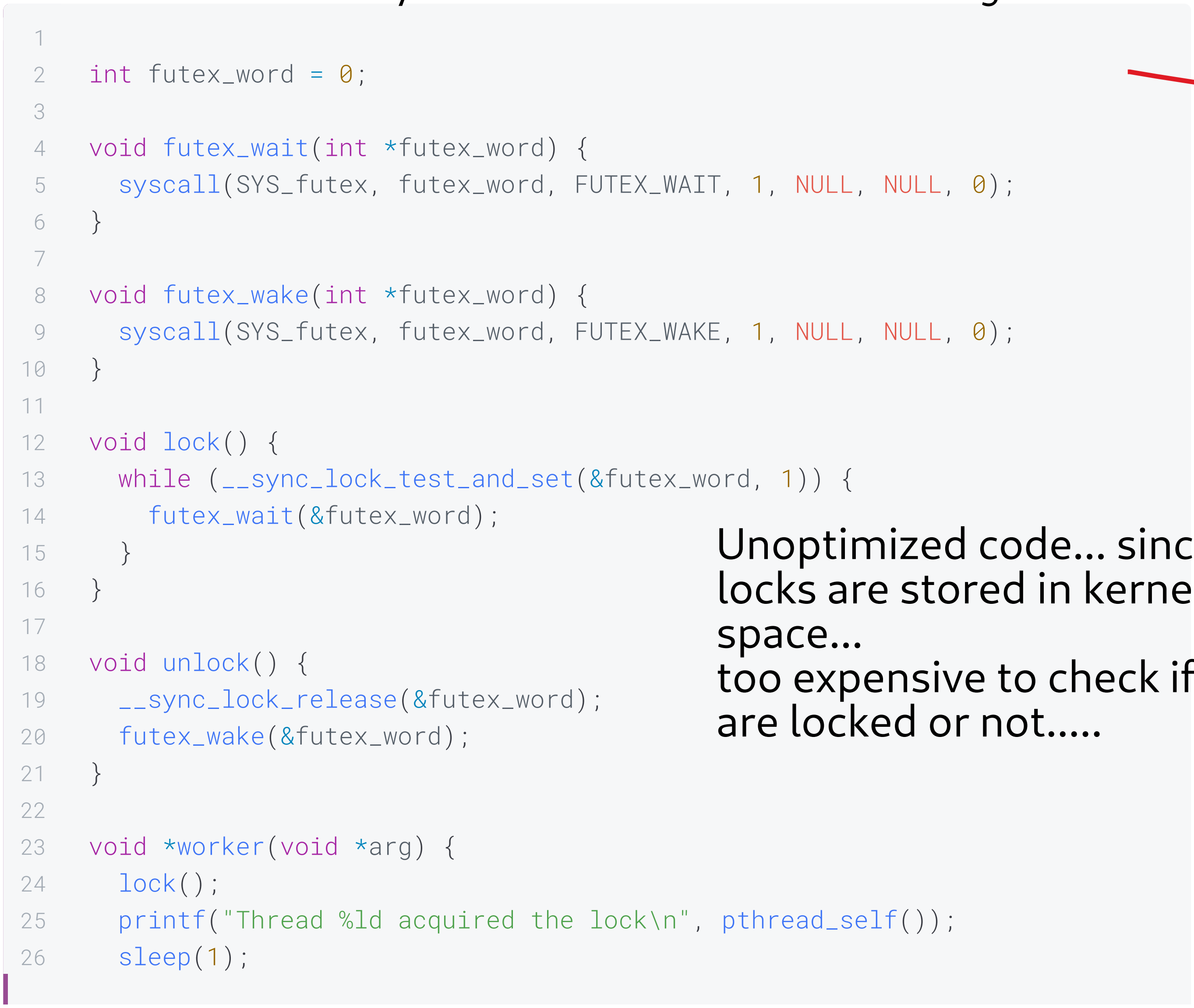
```c
typedef struct {
  atomic_int next_ticket;
  atomic_int serving;
} ticket_lock_t;

void ticket_lock_init(ticket_lock_t *lock) {
  atomic_init(&lock->next_ticket, 0);
  atomic_init(&lock->serving, 0);
}

void ticket_lock_acquire(ticket_lock_t *lock) {
  int my_ticket = atomic_fetch_add(&lock->next_ticket, 1);
  while (atomic_load(&lock->serving) != my_ticket) {}
}

void ticket_lock_release(ticket_lock_t *lock) {
  atomic_fetch_add(&lock->serving, 1);
}

void *worker(void *arg) {
  ticket_lock_t *lock = (ticket_lock_t *)arg;
  ticket_lock_acquire(lock);
  printf("Thread %ld has entered the critical section\n", pthread_self());
  sleep(1);
  printf("Thread %ld is leaving the critical section\n", pthread_self());
```

has issue of overflow...



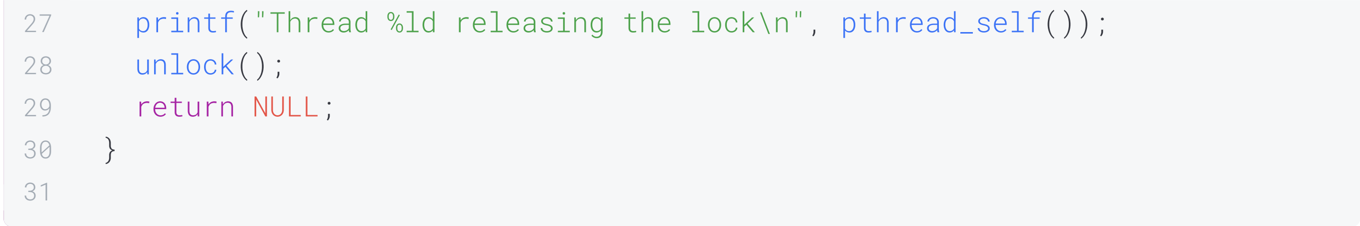
```c
  ticket_lock_release(lock);
  return NULL;
}

int main() {
  pthread_t t1, t2, t3;
  ticket_lock_t lock;
  ticket_lock_init(&lock);
  pthread_create(&t1, NULL, worker, &lock);
  pthread_create(&t2, NULL, worker, &lock);
  pthread_create(&t3, NULL, worker, &lock);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  return 0;
}
```

Solution?? use system calls and let kernel manage shit

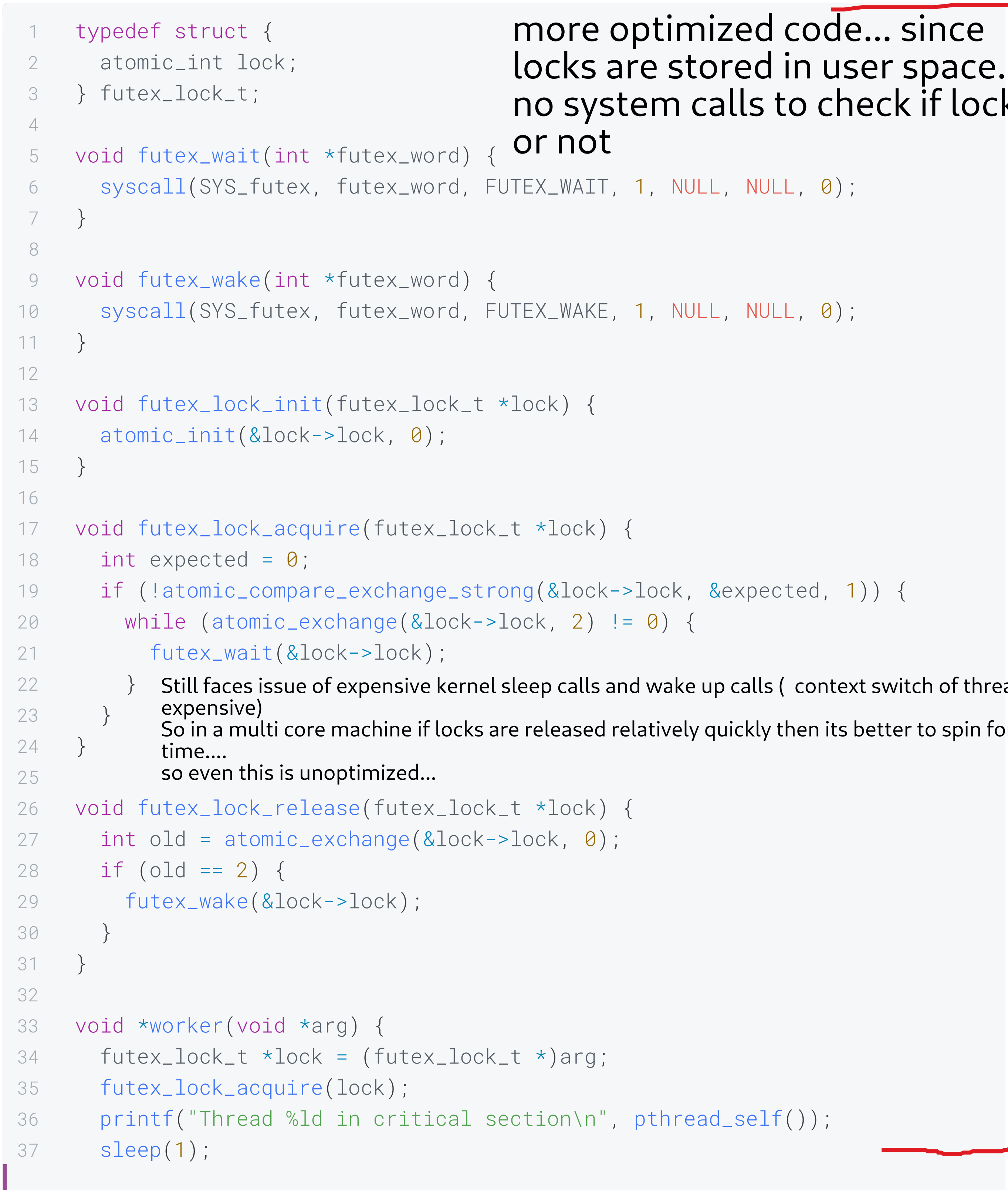


```c

int futex_word = 0;

void futex_wait(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
}

void futex_wake(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
}

void lock() {
  while (__sync_lock_test_and_set(&futex_word, 1)) {
    futex_wait(&futex_word);
  }
}

void unlock() {
  __sync_lock_release(&futex_word);
  futex_wake(&futex_word);
}

void *worker(void *arg) {
  lock();
  printf("Thread %ld acquired the lock\n", pthread_self());
  sleep(1);
```

Unoptimized code... since locks are stored in kernel space...
too expensive to check if locks are locked or not.....



```c
  printf("Thread %ld releasing the lock\n", pthread_self());
  unlock();
  return NULL;
}

```



more optimized code... since locks are stored in user space... no system calls to check if locked or not

```c
typedef struct {
  atomic_int lock;
} futex_lock_t;

void futex_wait(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
}

void futex_wake(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
}

void futex_lock_init(futex_lock_t *lock) {
  atomic_init(&lock->lock, 0);
}

void futex_lock_acquire(futex_lock_t *lock) {
  int expected = 0;
  if (!atomic_compare_exchange_strong(&lock->lock, &expected, 1)) {
    while (atomic_exchange(&lock->lock, 2) != 0) {
      futex_wait(&lock->lock);
    }
  }
}

void futex_lock_release(futex_lock_t *lock) {
  int old = atomic_exchange(&lock->lock, 0);
  if (old == 2) {
    futex_wake(&lock->lock);
  }
}

void *worker(void *arg) {
  futex_lock_t *lock = (futex_lock_t *)arg;
  futex_lock_acquire(lock);
  printf("Thread %ld in critical section\n", pthread_self());
  sleep(1);
```

Still faces issue of expensive kernel sleep calls and wake up calls (context switch of threads is expensive)
So in a multi core machine if locks are released relatively quickly then its better to spin for some time....
so even this is unoptimized...

```
  printf("Thread %ld leaving critical section\n", pthread_self());
  futex_lock_release(lock);
  return NULL;
}

int main() {
  pthread_t t1, t2, t3;
  futex_lock_t lock;
  futex_lock_init(&lock);
  pthread_create(&t1, NULL, worker, &lock);
  pthread_create(&t2, NULL, worker, &lock);
  pthread_create(&t3, NULL, worker, &lock);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  return 0;
}

```

---

multi-core?

---

fixes the issue of expensive kernel sleep calls and wake up calls (context switch of threads is expensive)

```
typedef struct {
  atomic_int lock;
} two_phase_lock_t;

void futex_wait(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
}

void futex_wake(int *futex_word) {
  syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
}

void two_phase_lock_init(two_phase_lock_t *lock) {
  atomic_init(&lock->lock, 0);
}

void two_phase_lock_acquire(two_phase_lock_t *lock) {
  int spin_count = 0, max_spins = 100;
  while (atomic_exchange_explicit(&lock->lock, 1, memory_order_acquire)) {
    spin_count++;
```

Spins for some time before calling syscall for sleep... SYS_futex

```
    if (spin_count >= max_spins) {
      futex_wait(&lock->lock);
      spin_count = 0;
    }
  }
}

void two_phase_lock_release(two_phase_lock_t *lock) {
  atomic_store_explicit(&lock->lock, 0, memory_order_release);
  futex_wake(&lock->lock);
}

void *worker(void *arg) {
  two_phase_lock_t *lock = (two_phase_lock_t *)arg;
  two_phase_lock_acquire(lock);
  printf("Thread %ld in critical section\n", pthread_self());
  sleep(1);
  printf("Thread %ld leaving critical section\n", pthread_self());
  two_phase_lock_release(lock);
  return NULL;
}

int main() {
  pthread_t t1, t2, t3;
  two_phase_lock_t lock;
  two_phase_lock_init(&lock);
  pthread_create(&t1, NULL, worker, &lock);
  pthread_create(&t2, NULL, worker, &lock);
  pthread_create(&t3, NULL, worker, &lock);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  return 0;
}
```

signal among threads

```
volatile int done = 0;

void *child(void *arg) {
  printf("child\n");
  done = 1;
  return NULL;
```

```c
}

int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  while (done == 0)
    ;
  printf("parent: end\n");
  return 0;
}
```

```c
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
void thr_exit() {
  pthread_cond_signal(&c);
}

void thr_join() {
  pthread_cond_wait(&c);
}
```

```c
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
void thr_exit() {
  pthread_cond_signal(&c);
}

void thr_join() {
  pthread_cond_wait(&c);
}
```

child enters exit() signals on `c`
parent enters join() and endlessly waits!

```c
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
int done = 0;

void thr_exit() {
  done = 1;
```

Change if to while... fixes the issue

```
 6     pthread_cond_signal(&c);
 7   }
 8
 9   void thr_join() {
10     if (done == 0)
11       pthread_cond_wait(&c);
12   }
```

Issue since if
Inturrupt after 10
then it signals and
then waits
indefinatly...

```
 1   pthread_cond_t c = PTHREAD_COND_INITIALIZER;
 2   int done = 0;
 3
 4   void thr_exit() {
 5     done = 1;
 6     pthread_cond_signal(&c);
 7   }
 8
 9   void thr_join() {
10     if (done == 0)
11       pthread_cond_wait(&c);
12   }
```

parent calls join()
sees done is 0 and calls cond_wait() on `c`
child enters exit() sets done = 1, and signals on `c`
parent exits happy!

---

again!

```
 1   pthread_cond_t c = PTHREAD_COND_INITIALIZER;
 2   int done = 0;
 3
 4   void thr_exit() {
 5     done = 1;
 6     pthread_cond_signal(&c);
 7   }
 8
 9   void thr_join() {
10     if (done == 0)
11       pthread_cond_wait(&c);
12   }
```

parent calls join()
sees done is 0
gets interrupted!
child calls exit; sets done = 1; signals on `c`
parent wakes up on wait on `c`
and waits forever!

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
int done = 0;

void thr_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void thr_join() {
  pthread_mutex_lock(&m);
  while (done == 0)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}

void *child(void *arg) {
  printf("child\n");
  thr_exit();
  return NULL;
}

int main() {
  pthread_t t;
  printf("parent: begin\n");
  pthread_create(&t, NULL, child, NULL);
  thr_join();
  printf("parent: end\n");
  return 0;
}
```

is done still required?

Yes since if the child process first executes and signals and then the parent process calls up join then there is an issue since it wont get the signal in time.. and will wait indefinatly...

producer-consumer

```
int buffer;
int count = 0;
void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}
int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```

```
int loops;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int count = 0;

void producer() {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    if (count == 1)
      pthread_cond_wait(&cond, &mutex);
    put(i);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

void consumer() {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    if (count == 0)
      pthread_cond_wait(&cond, &mutex);
```

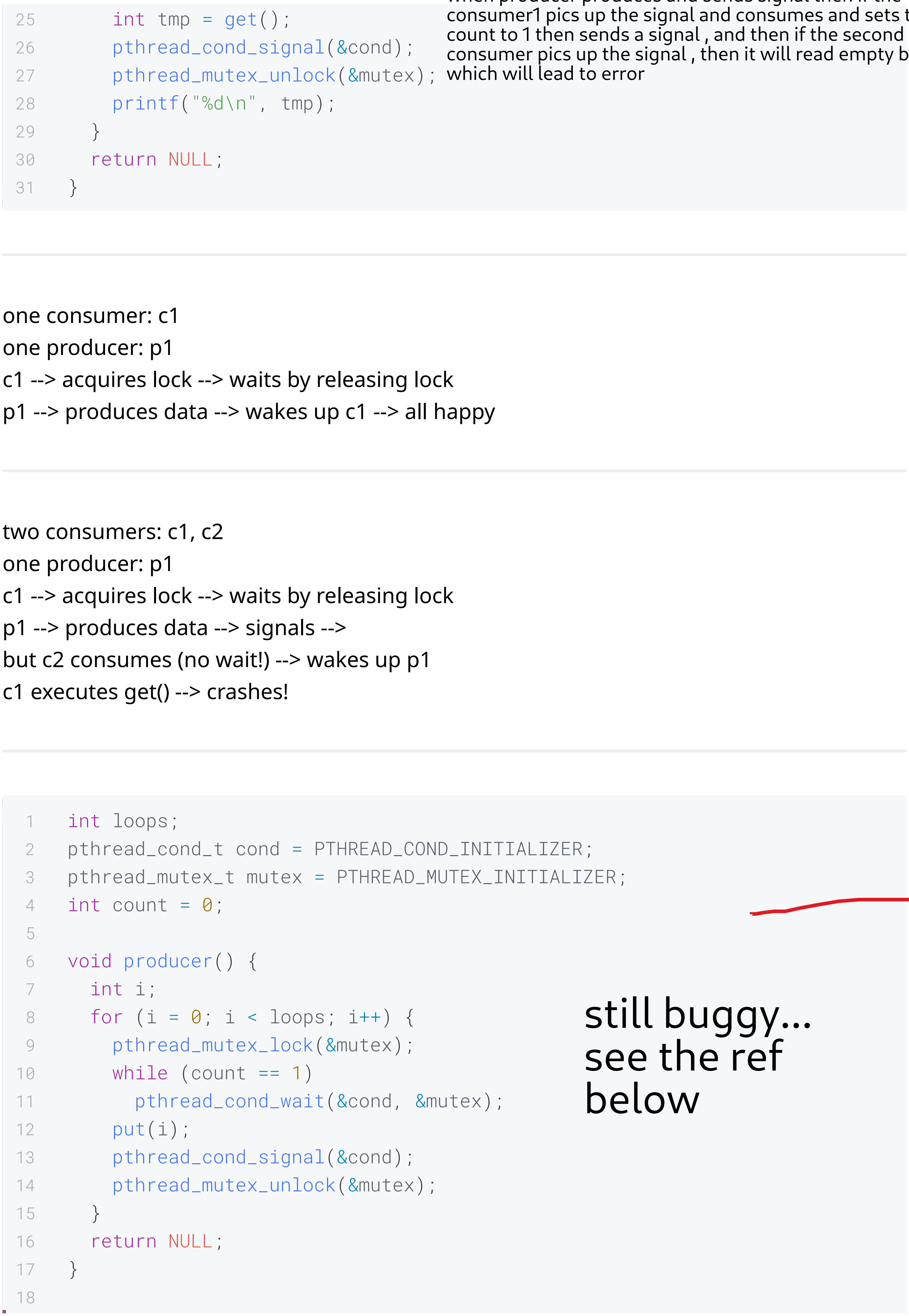
Imp because if consumer has not consumed then you donot want to currupt the buffer

replace with while to be safe

Imp because if producer has not produced then you donot want to consume empty buffer

replace with while to fix

Note that this suffers from major issue of 2 consumers ie that when producer produces and sends signal then if the consumer1 pics up the signal and consumes and sets the count to 1 then sends a signal , and then if the second consumer pics up the signal , then it will read empty buffer.... which will lead to error



```c
    int tmp = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
  return NULL;
}
```

one consumer: c1
one producer: p1
c1 --> acquires lock --> waits by releasing lock
p1 --> produces data --> wakes up c1 --> all happy

two consumers: c1, c2
one producer: p1
c1 --> acquires lock --> waits by releasing lock
p1 --> produces data --> signals -->
but c2 consumes (no wait!) --> wakes up p1
c1 executes get() --> crashes!

```c
int loops;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int count = 0;

void producer() {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == 1)
      pthread_cond_wait(&cond, &mutex);
    put(i);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

```

still buggy...
see the ref
below

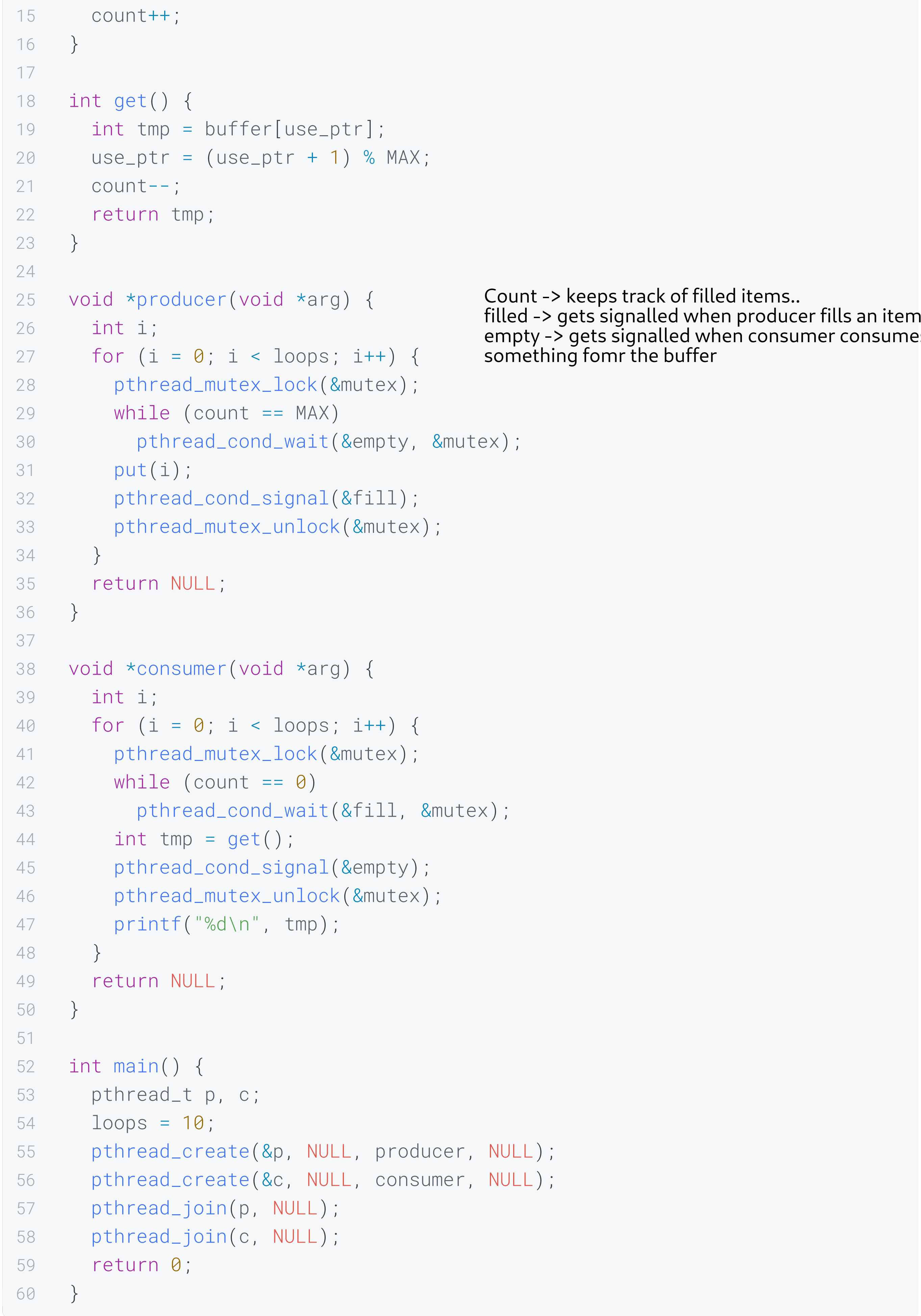
```c
void consumer() {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
      pthread_cond_wait(&cond, &mutex);
    int tmp = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
  return NULL;
}
```

---

two consumers: c1, c2
one producer: p1

c1 runs --> goes to sleep
c2 runs --> goes to sleep
p1 runs --> produces --> signals --> goes to sleep
c1 runs --> consumes --> signals
--> c2 catches the signal --> while (count == 0) --> goes to sleep
p1 is sleeping
c2 is sleeping
c1 wakes up --> while (count == 0) --> goes to sleep

```c
#define MAX 10
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;
int loops;

pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t fill = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void put(int value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1) % MAX;
```



```c
  count++;
}

int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1) % MAX;
  count--;
  return tmp;
}

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == MAX)
      pthread_cond_wait(&empty, &mutex);
    put(i);
    pthread_cond_signal(&fill);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
      pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
  return NULL;
}

int main() {
  pthread_t p, c;
  loops = 10;
  pthread_create(&p, NULL, producer, NULL);
  pthread_create(&c, NULL, consumer, NULL);
  pthread_join(p, NULL);
  pthread_join(c, NULL);
  return 0;
}
```

Count -> keeps track of filled items..
filled -> gets signalled when producer fills an item in buffer
empty -> gets signalled when consumer consumes something fomr the buffer

```c
int bytesLeft = MAX_HEAP_SIZE;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *allocate(int size) {
  pthread_mutex_lock(&m);
  while (bytesLeft < size)
    pthread_cond_wait(&c, &m);
  void *ptr = malloc();
  bytesLeft -= size;
  pthread_mutex_unlock(&m);
  return ptr;
}

void free(void *ptr, int size) {
  pthread_mutex_lock(&m);
  bytesLeft += size;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Amount of mem requested

3 threads: t1, t2, t3
t1 wants 1MB
t2 wants 2MB
currently 0MB in system
t3 frees 1MB
t2 wakes up --> and goes back to sleep
t1 is sleeping

Semaphores

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore;

void *thread_function(void *arg) {
  sem_wait(&semaphore);
  printf("Inside thread %ld\n", (long)arg);
  sem_post(&semaphore);
  return NULL;
}

int main() {
  pthread_t threads[2];
  sem_init(&semaphore, 0, 1)
  pthread_create(&threads[0], NULL, thread_function, (void *)1);
  pthread_create(&threads[1], NULL, thread_function, (void *)2);
  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);
  sem_destroy(&semaphore);
  return 0;
}
```

```
int sem_wait(sem_t *s) {
  s--
  if s < 0
    block until s is positive
}

int sem_post(sem_t *s) {
  s++
  if there are one or more threads waiting on s
    wake one of the waiting threads
}

```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t s;

void *child(void *arg) {
  printf("child\n");
  sem_post(&s);
```

```c
  return NULL;
}

int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0);
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  sem_wait(&s);
  printf("parent: end\n");
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int main() {
  int size = 256;
  void *shared_memory = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
  if (shared_memory == MAP_FAILED) {
    perror("mmap failed");
    exit(1);
  }

  pid_t pid = fork();

  if (pid == 0) {
    sprintf((char *)shared_memory, "Hello from child process!");
    munmap(shared_memory, size);
    exit(0);
  } else if (pid > 0) {
    wait(NULL);
    printf("Parent received: %s\n", (char *)shared_memory);
    munmap(shared_memory, size);
  } else {
    perror("fork failed");
    exit(1);
  }
```

Kernel Managed Mem Space... No need
to free up
Un named....

```c

  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>
// kernel-managed anonymous memory
int main() {
  sem_t *semaphore = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
                          MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  if (semaphore == MAP_FAILED) {
    perror("mmap failed");
    exit(1);
  }

  sem_init(semaphore, 1, 1);

  if (fork() == 0) {
    sem_wait(semaphore);
    printf("Child process entering critical section\n");
    sleep(2);
    printf("Child process leaving critical section\n");
    sem_post(semaphore);
    exit(0);
  } else {
    sem_wait(semaphore);
    printf("Parent process entering critical section\n");
    sleep(2);
    printf("Parent process leaving critical section\n");
    sem_post(semaphore);
    wait(NULL);
  }

  sem_destroy(semaphore);
  munmap(semaphore, sizeof(sem_t));
  return 0;
}
```

Very Good practice since you want to wait for your children to complete after that you can exit

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h>

#define SEM_NAME "/example_semaphore"
// POSIX shared memory object namespace
int main() {
  sem_t *semaphore = sem_open(SEM_NAME, O_CREAT, 0644, 1);
  if (semaphore == SEM_FAILED) {
    perror("sem_open failed");
    exit(1);
  }

  sem_wait(semaphore);
  printf("Process %d entering critical section\n", getpid());
  sleep(2);
  printf("Process %d leaving critical section\n", getpid());
  sem_post(semaphore);

  sem_close(semaphore);
  sem_unlink(SEM_NAME);
  return 0;
}
```

```c
// sender
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <string.h>
#include <unistd.h>

#define SHM_NAME "/shared_memory_example"
#define SEM_NAME "/semaphore_example"
#define MSG "Hello from process1!"

```

```c
int main() {
  int size = 256;
  int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
  ftruncate(shm_fd, size);
  void *shared_memory = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);

  sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0666, 0);

  sprintf((char *)shared_memory, MSG);
  printf("process1: message written to shared memory\n");

  sem_post(sem);

  munmap(shared_memory, size);
  close(shm_fd);
  sem_close(sem);

  return 0;
}
```

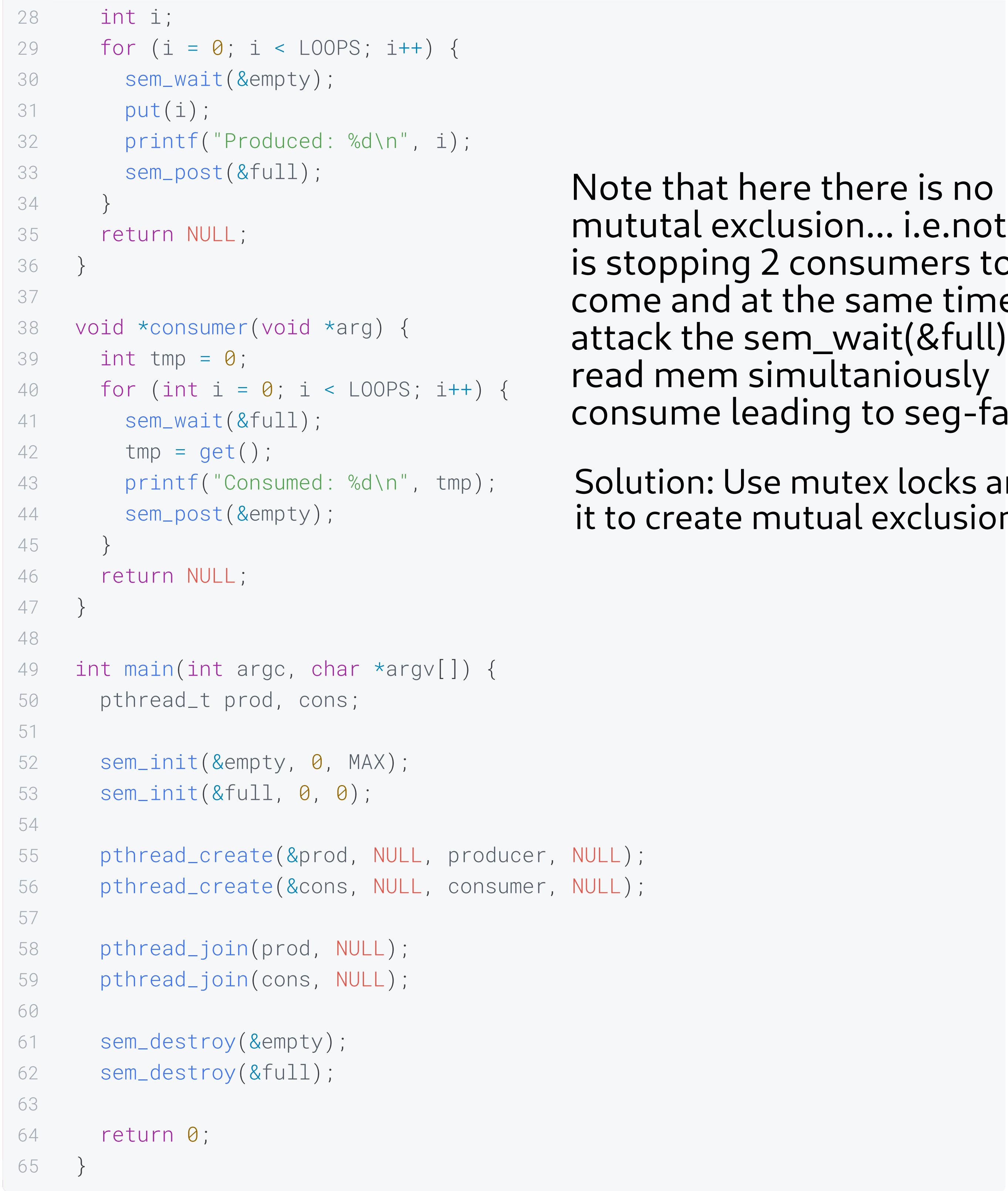
```c
// receiver
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h>

#define SHM_NAME "/shared_memory_example"
#define SEM_NAME "/semaphore_example"

int main() {
  int size = 256;
  int shm_fd = shm_open(SHM_NAME, O_RDONLY, 0666);
  void *shared_memory = mmap(0, size, PROT_READ, MAP_SHARED, shm_fd, 0);

  sem_t *sem = sem_open(SEM_NAME, 0);

  sem_wait(sem);

  printf("process2: received message - %s\n", (char *)shared_memory);

  munmap(shared_memory, size);
  close(shm_fd);
  sem_close(sem);
  sem_unlink(SEM_NAME);
  shm_unlink(SHM_NAME);

  return 0;
}
```

```c
// sender
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FILEPATH "/tmp/shared_memory_file"
#define SIZE 256

int main() {
  int fd = open(FILEPATH, O_CREAT | O_RDWR, 0666);
  ftruncate(fd, SIZE);

  void *shared_memory = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  sprintf((char *)shared_memory, "Hello from process 1");

  munmap(shared_memory, SIZE);
  close(fd);

  return 0;
}
```

```c
// receiver
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
```

```c

#define FILEPATH "/tmp/shared_memory_file"
#define SIZE 256

int main() {
  int fd = open(FILEPATH, O_RDONLY, 0666);

  void *shared_memory = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0);
  printf("Received message: %s\n", (char *)shared_memory);

  munmap(shared_memory, SIZE);
  close(fd);

  return 0;
}
```

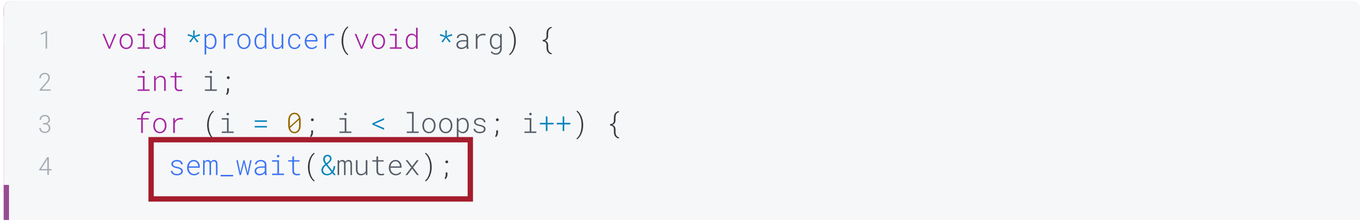
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX 10
#define LOOPS 20

int buffer[MAX];
int fill = 0;
int use = 0;

sem_t empty;
sem_t full;

void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % MAX;
}

int get() {
  int tmp = buffer[use];
  use = (use + 1) % MAX;
  return tmp;
}

void *producer(void *arg) {
```



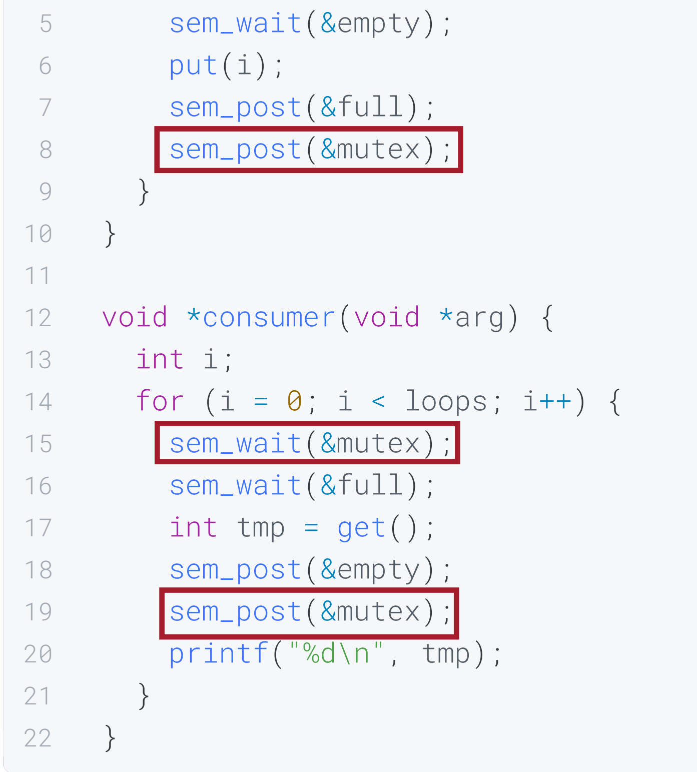
```c
  int i;
  for (i = 0; i < LOOPS; i++) {
    sem_wait(&empty);
    put(i);
    printf("Produced: %d\n", i);
    sem_post(&full);
  }
  return NULL;
}

void *consumer(void *arg) {
  int tmp = 0;
  for (int i = 0; i < LOOPS; i++) {
    sem_wait(&full);
    tmp = get();
    printf("Consumed: %d\n", tmp);
    sem_post(&empty);
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t prod, cons;

  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);

  pthread_create(&prod, NULL, producer, NULL);
  pthread_create(&cons, NULL, consumer, NULL);

  pthread_join(prod, NULL);
  pthread_join(cons, NULL);

  sem_destroy(&empty);
  sem_destroy(&full);

  return 0;
}
```

Note that here there is no mututal exclusion... i.e.nothing is stopping 2 consumers to come and at the same time attack the sem_wait(&full) and read mem simultaniously consume leading to seg-fault

Solution: Use mutex locks around it to create mutual exclusion

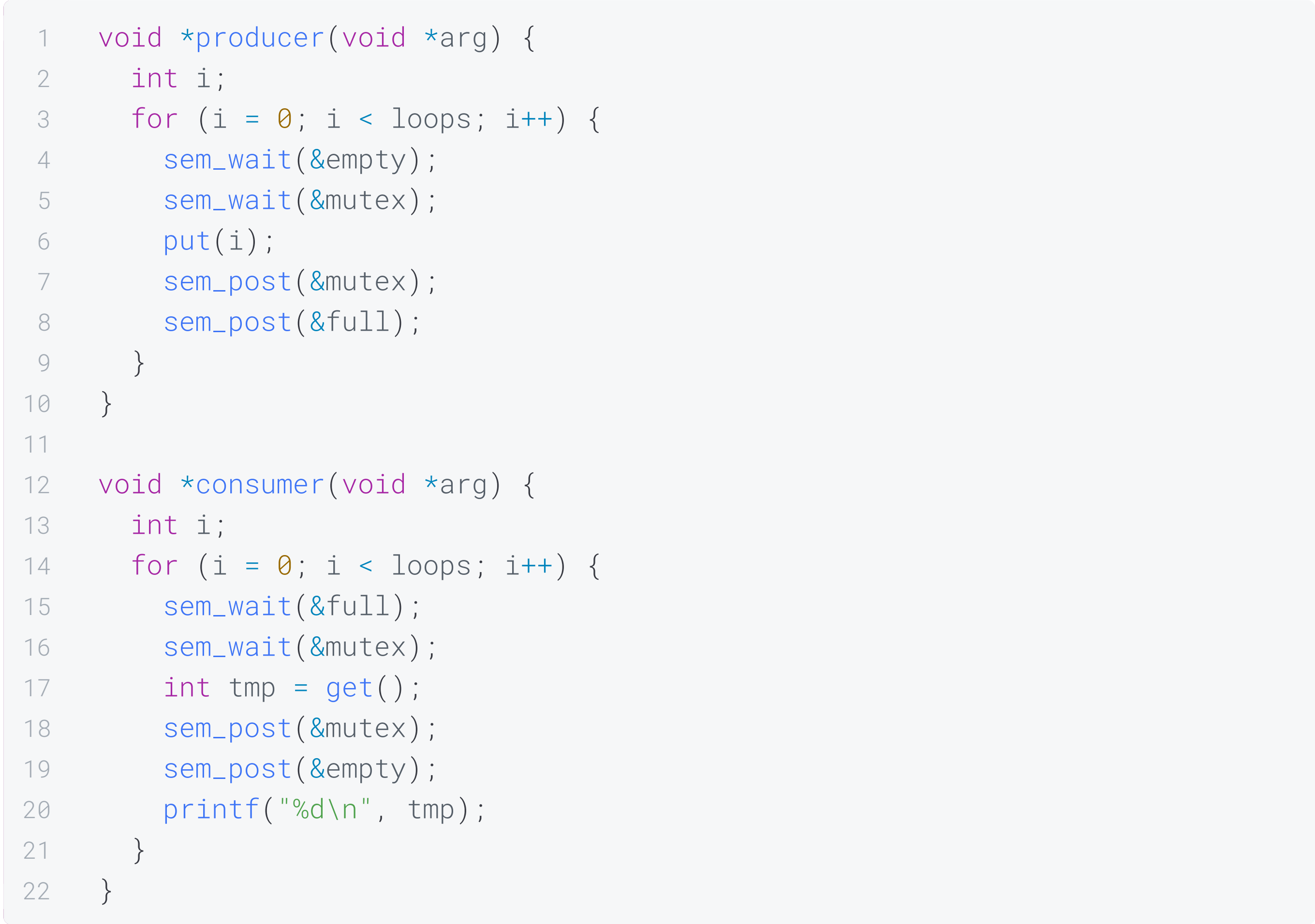


```c
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);
```

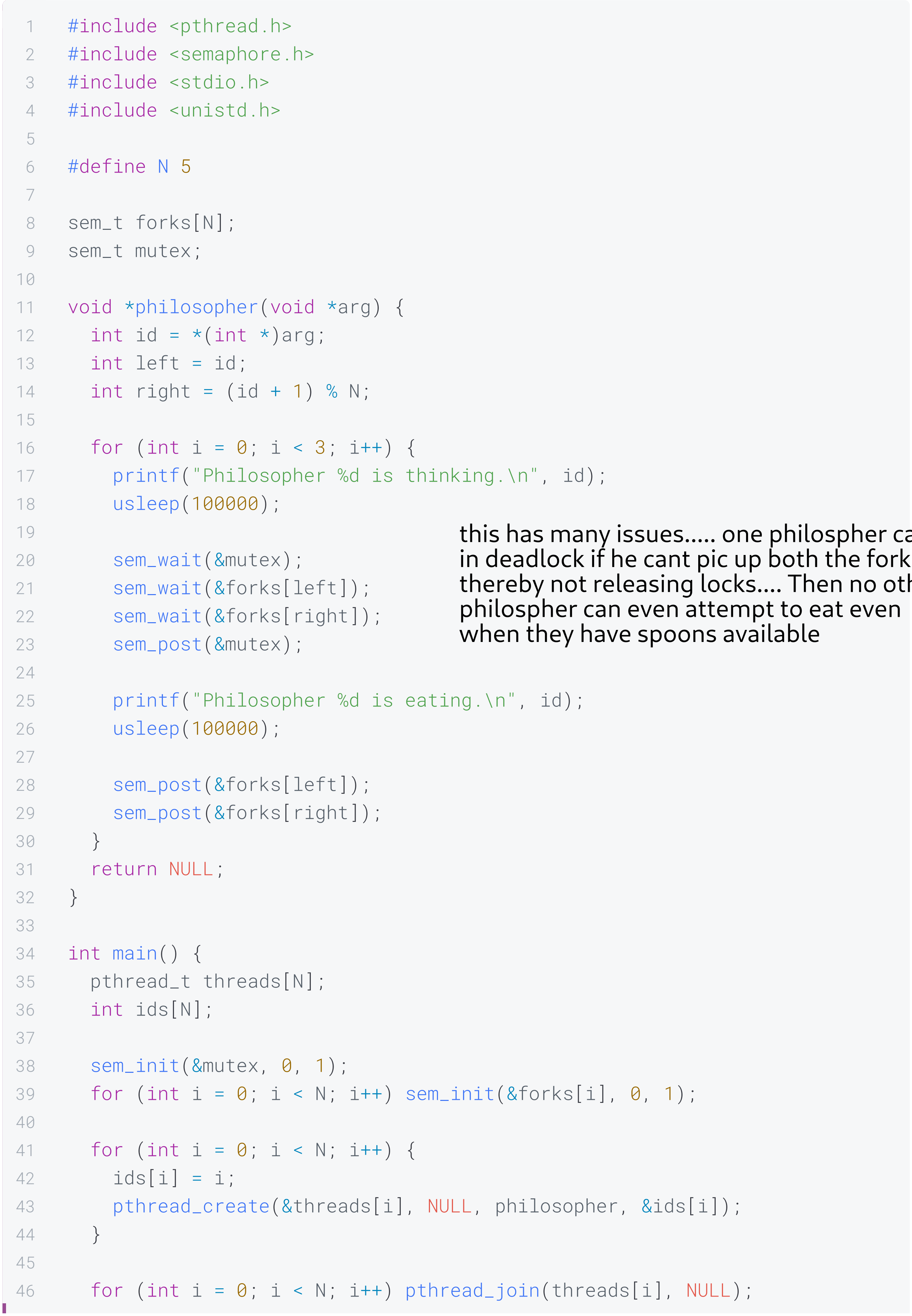


```
    sem_wait(&empty);
    put(i);
    sem_post(&full);
    sem_post(&mutex);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    printf("%d\n", tmp);
  }
}
```

Please dont lock and then wait for someone to do something....
since the other guy wont be able to do any thing.... since you have locked ....
Its just like waiting for krishna when you have locked him outside...



```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```



```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N 5

sem_t forks[N];
sem_t mutex;

void *philosopher(void *arg) {
  int id = *(int *)arg;
  int left = id;
  int right = (id + 1) % N;

  for (int i = 0; i < 3; i++) {
    printf("Philosopher %d is thinking.\n", id);
    usleep(100000);

    sem_wait(&mutex);
    sem_wait(&forks[left]);
    sem_wait(&forks[right]);
    sem_post(&mutex);

    printf("Philosopher %d is eating.\n", id);
    usleep(100000);

    sem_post(&forks[left]);
    sem_post(&forks[right]);
  }
  return NULL;
}

int main() {
  pthread_t threads[N];
  int ids[N];

  sem_init(&mutex, 0, 1);
  for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);

  for (int i = 0; i < N; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, philosopher, &ids[i]);
  }

  for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);
```

this has many issues..... one philospher can be in deadlock if he cant pic up both the forks.... thereby not releasing locks.... Then no other philospher can even attempt to eat even when they have spoons available

```c

  for (int i = 0; i < N; i++) sem_destroy(&forks[i]);
  sem_destroy(&mutex);

  return 0;
}

```

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N 5

sem_t forks[N];

void *philosopher(void *arg) {
  int id = *(int *)arg;
  int left = id;
  int right = (id + 1) % N;

  for (int i = 0; i < 3; i++) {
    printf("Philosopher %d is thinking.\n", id);
    usleep(100000);

    if (id % 2 == 0) {
      sem_wait(&forks[left]);
      sem_wait(&forks[right]);
    } else {
      sem_wait(&forks[right]);
      sem_wait(&forks[left]);
    }

    printf("Philosopher %d is eating.\n", id);
    usleep(100000);

    sem_post(&forks[left]);
    sem_post(&forks[right]);
  }
  return NULL;
}

```

```c
int main() {
  pthread_t threads[N];
  int ids[N];

  for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);

  for (int i = 0; i < N; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, philosopher, &ids[i]);
  }

  for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);

  for (int i = 0; i < N; i++) sem_destroy(&forks[i]);

  return 0;
}
```

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N 5

sem_t forks[N];

void *philosopher(void *arg) {
  int id = *(int *)arg;
  int left = id;
  int right = (id + 1) % N;

  for (int i = 0; i < 3; i++) {
    printf("Philosopher %d is thinking.\n", id);
    usleep(100000);

    if (id == N - 1) {
      sem_wait(&forks[right]);
      sem_wait(&forks[left]);
    } else {
      sem_wait(&forks[left]);
      sem_wait(&forks[right]);
    }

    printf("Philosopher %d is eating.\n", id);
    usleep(100000);

    sem_post(&forks[left]);
    sem_post(&forks[right]);
  }
  return NULL;
}

int main() {
  pthread_t threads[N];
  int ids[N];

  for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);

  for (int i = 0; i < N; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, philosopher, &ids[i]);
  }

  for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);

  for (int i = 0; i < N; i++) sem_destroy(&forks[i]);

  return 0;
}
```

```python
import threading

semaphore = threading.Semaphore(3)
shared_counter = 0

def increment():
    global shared_counter
    semaphore.acquire()
    local_copy = shared_counter
    local_copy += 1
    shared_counter = local_copy
    semaphore.release()

threads = [threading.Thread(target=increment) for _ in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print(shared_counter)
```

```
import threading
import time

def compute_task():
  total = 0
  for i in range(10**7):
    total += i
  return total

def multithreaded_execution():
  threads = [threading.Thread(target=compute_task) for _ in range(4)]
  start_time = time.time()
  for thread in threads:
    thread.start()
  for thread in threads:
    thread.join()
  return time.time() - start_time

if __name__ == "__main__":
  threading_time = multithreaded_execution()
  print({"Multithreading Time (s)": threading_time})

```

```
import time
from multiprocessing import Process

def compute_task():
  total = 0
  for i in range(10**8):
    total += i
  return total

def multiprocessing_execution():
  processes = [Process(target=compute_task) for _ in range(4)]
  start_time = time.time()
  for process in processes:
    process.start()
  for process in processes:
    process.join()
  return time.time() - start_time

if __name__ == "__main__":
  multiprocessing_time = multiprocessing_execution()
  print({"Multiprocessing Time (s)": multiprocessing_time})
```

```
from multiprocessing import Process, Semaphore
import time

def worker(sem, task_id):
  sem.acquire()
  print(f"Process {task_id} started")
  time.sleep(2)
  print(f"Process {task_id} finished")
  sem.release()

if __name__ == "__main__":
  sem = Semaphore(2)
  processes = [Process(target=worker, args=(sem, i)) for i in range(5)]
  for p in processes:
    p.start()
  for p in processes:
    p.join()
```
