

```
1  int TestAndSet(int * ptr, int new_val)
2  {
3      int old_val = *ptr;
4      *ptr = new_val;
5      return old_val;
6  }
7
8  int lock = 0;
9  int counter = 0;
10
11 void* increment_counter(void* arg)
12 {
13     printf("Thread %s starting\n", (char*)arg);
14     for (int i = 0; i < 100000; i++) {
15         while (TestAndSet(&lock, 1) == 1);
16         counter++;
17         lock = 0;
18     }
19     return NULL;
20 }
21
22 int main()
23 {
24     pthread_t t1, t2;
25     pthread_create(&t1, NULL, increment_counter, "Red");
26     pthread_create(&t2, NULL, increment_counter, "Blue");
27     pthread_join(t1, NULL);
28     pthread_join(t2, NULL);
29     printf("Final counter value: %d\n", counter);
30     return 0;
31 }
```

---

## 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!

---

```
1  atomic_flag lock = ATOMIC_FLAG_INIT;
2  int counter = 0;
3  void acquire_lock(atomic_flag *lock) {
4      while (atomic_flag_test_and_set(lock)) {
5          sched_yield();
6      }
7  }
8
9  void release_lock(atomic_flag *lock) {
10     atomic_flag_clear(lock);
11 }
12
13 void* increment_counter(void* arg) {
14     for (int i = 0; i < 100000; i++) {
15         acquire_lock(&lock);
16         counter++;
17         release_lock(&lock);
18     }
19     return NULL;
20 }
21
22 int main() {
23     pthread_t t1, t2;
24     pthread_create(&t1, NULL, increment_counter, "Red");
25     pthread_create(&t2, NULL, increment_counter, "Blue");
26     pthread_join(t1, NULL);
27     pthread_join(t2, NULL);
28     printf("Final counter value: %d\n", counter);
29     return 0;
30 }
```

---

fairness? - starvation

: depends on which thread gets scheduled

---

## spin locks

T2 ><sub>priority</sub> 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 :(

---

```
1  typedef struct {
2      atomic_int next_ticket;
3      atomic_int serving;
4  } ticket_lock_t;
5
6  void ticket_lock_init(ticket_lock_t *lock) {
7      atomic_init(&lock->next_ticket, 0);
8      atomic_init(&lock->serving, 0);
9  }
10
11 void ticket_lock_acquire(ticket_lock_t *lock) {
12     int my_ticket = atomic_fetch_add(&lock->next_ticket, 1);
13     while (atomic_load(&lock->serving) != my_ticket) {}
14 }
15
16 void ticket_lock_release(ticket_lock_t *lock) {
17     atomic_fetch_add(&lock->serving, 1);
18 }
19
20 void *worker(void *arg) {
21     ticket_lock_t *lock = (ticket_lock_t *)arg;
22     ticket_lock_acquire(lock);
23     printf("Thread %ld has entered the critical section\n", pthread_self());
24     sleep(1);
25     printf("Thread %ld is leaving the critical section\n", pthread_self());
```

has issue of  
overflow...

```

26     ticket_lock_release(lock);
27     return NULL;
28 }
29
30 int main() {
31     pthread_t t1, t2, t3;
32     ticket_lock_t lock;
33     ticket_lock_init(&lock);
34     pthread_create(&t1, NULL, worker, &lock);
35     pthread_create(&t2, NULL, worker, &lock);
36     pthread_create(&t3, NULL, worker, &lock);
37     pthread_join(t1, NULL);
38     pthread_join(t2, NULL);
39     pthread_join(t3, NULL);
40     return 0;
41 }

```

Solution?? use system calls and let kernel manage shit

```

1
2  int futex_word = 0;
3
4  void futex_wait(int *futex_word) {
5      syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
6  }
7
8  void futex_wake(int *futex_word) {
9      syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
10 }
11
12 void lock() {
13     while (__sync_lock_test_and_set(&futex_word, 1)) {
14         futex_wait(&futex_word);
15     }
16 }
17
18 void unlock() {
19     __sync_lock_release(&futex_word);
20     futex_wake(&futex_word);
21 }
22
23 void *worker(void *arg) {
24     lock();
25     printf("Thread %ld acquired the lock\n", pthread_self());
26     sleep(1);

```

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

```

27     printf("Thread %ld releasing the lock\n", pthread_self());
28     unlock();
29     return NULL;
30 }
31

```

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

```

1  typedef struct {
2      atomic_int lock;
3  } futex_lock_t;
4
5  void futex_wait(int *futex_word) {
6      syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
7  }
8
9  void futex_wake(int *futex_word) {
10     syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
11 }
12
13 void futex_lock_init(futex_lock_t *lock) {
14     atomic_init(&lock->lock, 0);
15 }
16
17 void futex_lock_acquire(futex_lock_t *lock) {
18     int expected = 0;
19     if (!atomic_compare_exchange_strong(&lock->lock, &expected, 1)) {
20         while (atomic_exchange(&lock->lock, 2) != 0) {
21             futex_wait(&lock->lock);
22         } Still faces issue of expensive kernel sleep calls and wake up calls ( context switch of threads is
23         } expensive)
24     } So in a multi core machine if locks are released relatively quickly then its better to spin for some
25     time....
26     so even this is unoptimized...
27
28 void futex_lock_release(futex_lock_t *lock) {
29     int old = atomic_exchange(&lock->lock, 0);
30     if (old == 2) {
31         futex_wake(&lock->lock);
32     }
33 }
34
35 void *worker(void *arg) {
36     futex_lock_t *lock = (futex_lock_t *)arg;
37     futex_lock_acquire(lock);
38     printf("Thread %ld in critical section\n", pthread_self());
39     sleep(1);

```

```

38     printf("Thread %ld leaving critical section\n", pthread_self());
39     futex_lock_release(lock);
40     return NULL;
41 }
42
43 int main() {
44     pthread_t t1, t2, t3;
45     futex_lock_t lock;
46     futex_lock_init(&lock);
47     pthread_create(&t1, NULL, worker, &lock);
48     pthread_create(&t2, NULL, worker, &lock);
49     pthread_create(&t3, NULL, worker, &lock);
50     pthread_join(t1, NULL);
51     pthread_join(t2, NULL);
52     pthread_join(t3, NULL);
53     return 0;
54 }
55

```

## multi-core?

```

1  typedef struct {
2      atomic_int lock;
3  } two_phase_lock_t;
4
5  void futex_wait(int *futex_word) {
6      syscall(SYS_futex, futex_word, FUTEX_WAIT, 1, NULL, NULL, 0);
7  }
8
9  void futex_wake(int *futex_word) {
10     syscall(SYS_futex, futex_word, FUTEX_WAKE, 1, NULL, NULL, 0);
11 }
12
13 void two_phase_lock_init(two_phase_lock_t *lock) {
14     atomic_init(&lock->lock, 0);
15 }
16
17 void two_phase_lock_acquire(two_phase_lock_t *lock) {
18     int spin_count = 0, max_spins = 100;
19     while (atomic_exchange_explicit(&lock->lock, 1, memory_order_acquire)) {
20         spin_count++;

```

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

Spins for some time before calling syscall for sleep... SYS\_futex

```

21     if (spin_count >= max_spins) {
22         futex_wait(&lock->lock);
23         spin_count = 0;
24     }
25 }
26 }
27
28 void two_phase_lock_release(two_phase_lock_t *lock) {
29     atomic_store_explicit(&lock->lock, 0, memory_order_release);
30     futex_wake(&lock->lock);
31 }
32
33 void *worker(void *arg) {
34     two_phase_lock_t *lock = (two_phase_lock_t *)arg;
35     two_phase_lock_acquire(lock);
36     printf("Thread %ld in critical section\n", pthread_self());
37     sleep(1);
38     printf("Thread %ld leaving critical section\n", pthread_self());
39     two_phase_lock_release(lock);
40     return NULL;
41 }
42
43 int main() {
44     pthread_t t1, t2, t3;
45     two_phase_lock_t lock;
46     two_phase_lock_init(&lock);
47     pthread_create(&t1, NULL, worker, &lock);
48     pthread_create(&t2, NULL, worker, &lock);
49     pthread_create(&t3, NULL, worker, &lock);
50     pthread_join(t1, NULL);
51     pthread_join(t2, NULL);
52     pthread_join(t3, NULL);
53     return 0;
54 }

```

---

## signal among threads

```

1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;

```

```
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     pthread_create(&c, NULL, child, NULL);
13     while (done == 0)
14         ;
15     printf("parent: end\n");
16     return 0;
17 }
```

---

```
1  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
2  void thr_exit() {
3      pthread_cond_signal(&c);
4  }
5
6  void thr_join() {
7      pthread_cond_wait(&c);
8  }
```

---

```
1  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
2  void thr_exit() {
3      pthread_cond_signal(&c);
4  }
5
6  void thr_join() {
7      pthread_cond_wait(&c);
8  }
```

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

---

```
1  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
2  int done = 0;
3
4  void thr_exit() {
5      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  
Interrupt after 10  
then it signals and  
then waits  
indefinitely...

```
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!

---

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
3  int done = 0;
4
5  void thr_exit() {
6      pthread_mutex_lock(&m);
7      done = 1;
8      pthread_cond_signal(&c);
9      pthread_mutex_unlock(&m);
10 }
11
12 void thr_join() {
13     pthread_mutex_lock(&m);
14     while (done == 0)
15         pthread_cond_wait(&c, &m);
16     pthread_mutex_unlock(&m);
17 }
18
19 void *child(void *arg) {
20     printf("child\n");
21     thr_exit();
22     return NULL;
23 }
24
25 int main() {
26     pthread_t t;
27     printf("parent: begin\n");
28     pthread_create(&t, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

---

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 indefinately...

## producer-consumer

```
1  int buffer;
2  int count = 0;
3  void put(int value) {
4      assert(count == 0);
5      count = 1;
6      buffer = value;
7  }
8  int get() {
9      assert(count == 1);
10     count = 0;
11     return buffer;
12 }
```

```
1  int loops;
2  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4  int count = 0;
5
6  void producer() {
7      int i;
8      for (i = 0; i < loops; i++) {
9          pthread_mutex_lock(&mutex);
10         if (count == 1)
11             pthread_cond_wait(&cond, &mutex);
12         put(i);
13         pthread_cond_signal(&cond);
14         pthread_mutex_unlock(&mutex);
15     }
16     return NULL;
17 }
18
19 void consumer() {
20     int i;
21     for (i = 0; i < loops; i++) {
22         pthread_mutex_lock(&mutex);
23         if (count == 0)
24             pthread_cond_wait(&cond, &mutex);
```

Imp because if consumer has not consumed then you donot want to corrupt 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

```

25     int tmp = get();
26     pthread_cond_signal(&cond);
27     pthread_mutex_unlock(&mutex);
28     printf("%d\n", tmp);
29 }
30 return NULL;
31 }

```

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

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!

```

1  int loops;
2  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4  int count = 0;
5
6  void producer() {
7      int i;
8      for (i = 0; i < loops; i++) {
9          pthread_mutex_lock(&mutex);
10         while (count == 1)
11             pthread_cond_wait(&cond, &mutex);
12         put(i);
13         pthread_cond_signal(&cond);
14         pthread_mutex_unlock(&mutex);
15     }
16     return NULL;
17 }
18

```

still buggy...  
see the ref  
below

```

19 void consumer() {
20     int i;
21     for (i = 0; i < loops; i++) {
22         pthread_mutex_lock(&mutex);
23         while (count == 0)
24             pthread_cond_wait(&cond, &mutex);
25         int tmp = get();
26         pthread_cond_signal(&cond);
27         pthread_mutex_unlock(&mutex);
28         printf("%d\n", tmp);
29     }
30     return NULL;
31 }

```

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

```

1  #define MAX 10
2  int buffer[MAX];
3  int fill_ptr = 0;
4  int use_ptr = 0;
5  int count = 0;
6  int loops;
7
8  pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
9  pthread_cond_t fill = PTHREAD_COND_INITIALIZER;
10 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
11
12 void put(int value) {
13     buffer[fill_ptr] = value;
14     fill_ptr = (fill_ptr + 1) % MAX;

```

```

15     count++;
16 }
17
18 int get() {
19     int tmp = buffer[use_ptr];
20     use_ptr = (use_ptr + 1) % MAX;
21     count--;
22     return tmp;
23 }
24
25 void *producer(void *arg) {
26     int i;
27     for (i = 0; i < loops; i++) {
28         pthread_mutex_lock(&mutex);
29         while (count == MAX)
30             pthread_cond_wait(&empty, &mutex);
31         put(i);
32         pthread_cond_signal(&fill);
33         pthread_mutex_unlock(&mutex);
34     }
35     return NULL;
36 }
37
38 void *consumer(void *arg) {
39     int i;
40     for (i = 0; i < loops; i++) {
41         pthread_mutex_lock(&mutex);
42         while (count == 0)
43             pthread_cond_wait(&fill, &mutex);
44         int tmp = get();
45         pthread_cond_signal(&empty);
46         pthread_mutex_unlock(&mutex);
47         printf("%d\n", tmp);
48     }
49     return NULL;
50 }
51
52 int main() {
53     pthread_t p, c;
54     loops = 10;
55     pthread_create(&p, NULL, producer, NULL);
56     pthread_create(&c, NULL, consumer, NULL);
57     pthread_join(p, NULL);
58     pthread_join(c, NULL);
59     return 0;
60 }

```

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

---

```
1  int bytesLeft = MAX_HEAP_SIZE;
2  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
3  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
4
5  void *allocate(int size) {
6      pthread_mutex_lock(&m);
7      while (bytesLeft < size)
8          pthread_cond_wait(&c, &m);
9      void *ptr = malloc();
10     bytesLeft -= size;
11     pthread_mutex_unlock(&m);
12     return ptr;
13 }
14
15 void free(void *ptr, int size) {
16     pthread_mutex_lock(&m);
17     bytesLeft += size;
18     pthread_cond_signal(&c);
19     pthread_mutex_unlock(&m);
20 }
```

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

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  sem_t semaphore;
6
7  void *thread_function(void *arg) {
```

---

```

8     sem_wait(&semaphore);
9     printf("Inside thread %ld\n", (long)arg);
10    sem_post(&semaphore);
11    return NULL;
12 }
13
14 int main() {
15     pthread_t threads[2];
16     sem_init(&semaphore, 0, 1)
17     pthread_create(&threads[0], NULL, thread_function, (void *)1);
18     pthread_create(&threads[1], NULL, thread_function, (void *)2);
19     pthread_join(threads[0], NULL);
20     pthread_join(threads[1], NULL);
21     sem_destroy(&semaphore);
22     return 0;
23 }

```

---

```

1  int sem_wait(sem_t *s) {
2      s--
3      if s < 0
4          block until s is positive
5  }
6
7  int sem_post(sem_t *s) {
8      s++
9      if there are one or more threads waiting on s
10         wake one of the waiting threads
11  }
12

```

---

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5
6  sem_t s;
7
8  void *child(void *arg) {
9      printf("child\n");
10     sem_post(&s);

```



```

11     return NULL;
12 }
13
14 int main(int argc, char *argv[]) {
15     sem_init(&s, 0, 0);
16     printf("parent: begin\n");
17     pthread_t c;
18     pthread_create(&c, NULL, child, NULL);
19     sem_wait(&s);
20     printf("parent: end\n");
21     return 0;
22 }

```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/mman.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6  #include <string.h>
7
8  int main() {
9      int size = 256;
10     void *shared_memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
11     MAP_SHARED | MAP_ANONYMOUS, -1, 0);
12     if (shared_memory == MAP_FAILED) {
13         perror("mmap failed");
14         exit(1);
15     }
16
17     pid_t pid = fork();
18
19     if (pid == 0) {
20         sprintf((char *)shared_memory, "Hello from child process!");
21         munmap(shared_memory, size);
22         exit(0);
23     } else if (pid > 0) {
24         wait(NULL);
25         printf("Parent received: %s\n", (char *)shared_memory);
26         munmap(shared_memory, size);
27     } else {
28         perror("fork failed");
29         exit(1);
30     }

```

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

```
30
31     return 0;
32 }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/mman.h>
4  #include <semaphore.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7  // kernel-managed anonymous memory
8  int main() {
9      sem_t *semaphore = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
10                             MAP_SHARED | MAP_ANONYMOUS, -1, 0);
11
12      if (semaphore == MAP_FAILED) {
13          perror("mmap failed");
14          exit(1);
15      }
16
17      sem_init(semaphore, 1, 1);
18
19      if (fork() == 0) {
20          sem_wait(semaphore);
21          printf("Child process entering critical section\n");
22          sleep(2);
23          printf("Child process leaving critical section\n");
24          sem_post(semaphore);
25          exit(0);
26      } else {
27          sem_wait(semaphore);
28          printf("Parent process entering critical section\n");
29          sleep(2);
30          printf("Parent process leaving critical section\n");
31          sem_post(semaphore);
32          wait(NULL);          Very Good practice since you want to wait for your children to
                               complete after that you can exit
33      }
34
35      sem_destroy(semaphore);
36      munmap(semaphore, sizeof(sem_t));
37      return 0;
38 }
```

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <sys/stat.h>
5  #include <semaphore.h>
6  #include <unistd.h>
7
8  #define SEM_NAME "/example_semaphore"
9  // POSIX shared memory object namespace
10 int main() {
11     sem_t *semaphore = sem_open(SEM_NAME, O_CREAT, 0644, 1);
12     if (semaphore == SEM_FAILED) {
13         perror("sem_open failed");
14         exit(1);
15     }
16
17     sem_wait(semaphore);
18     printf("Process %d entering critical section\n", getpid());
19     sleep(2);
20     printf("Process %d leaving critical section\n", getpid());
21     sem_post(semaphore);
22
23     sem_close(semaphore);
24     sem_unlink(SEM_NAME);
25     return 0;
26 }
```

---

```
1  // sender
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <fcntl.h>
6  #include <sys/stat.h>
7  #include <semaphore.h>
8  #include <string.h>
9  #include <unistd.h>
10
11 #define SHM_NAME "/shared_memory_example"
12 #define SEM_NAME "/semaphore_example"
13 #define MSG "Hello from process1!"
14
```

```
15 int main() {
16     int size = 256;
17     int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
18     ftruncate(shm_fd, size);
19     void *shared_memory = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);
20
21     sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0666, 0);
22
23     sprintf((char *)shared_memory, MSG);
24     printf("process1: message written to shared memory\n");
25
26     sem_post(sem);
27
28     munmap(shared_memory, size);
29     close(shm_fd);
30     sem_close(sem);
31
32     return 0;
33 }
```

```
1 // receiver
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/mman.h>
5 #include <fcntl.h>
6 #include <sys/stat.h>
7 #include <semaphore.h>
8 #include <unistd.h>
9
10 #define SHM_NAME "/shared_memory_example"
11 #define SEM_NAME "/semaphore_example"
12
13 int main() {
14     int size = 256;
15     int shm_fd = shm_open(SHM_NAME, O_RDONLY, 0666);
16     void *shared_memory = mmap(0, size, PROT_READ, MAP_SHARED, shm_fd, 0);
17
18     sem_t *sem = sem_open(SEM_NAME, 0);
19
20     sem_wait(sem);
21
22     printf("process2: received message - %s\n", (char *)shared_memory);
23 }
```

```
24     munmap(shared_memory, size);
25     close(shm_fd);
26     sem_close(sem);
27     sem_unlink(SEM_NAME);
28     shm_unlink(SHM_NAME);
29
30     return 0;
31 }
```

---

```
1  // sender
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  #define FILEPATH "/tmp/shared_memory_file"
10 #define SIZE 256
11
12 int main() {
13     int fd = open(FILEPATH, O_CREAT | O_RDWR, 0666);
14     ftruncate(fd, SIZE);
15
16     void *shared_memory = mmap(NULL, SIZE, PROT_READ | PROT_WRITE,
17     MAP_SHARED, fd, 0);
18     sprintf((char *)shared_memory, "Hello from process 1");
19
20     munmap(shared_memory, SIZE);
21     close(fd);
22
23     return 0;
24 }
```

---

```
1  // receiver
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <fcntl.h>
6  #include <unistd.h>
```

```
7
8  #define FILEPATH "/tmp/shared_memory_file"
9  #define SIZE 256
10
11  int main() {
12      int fd = open(FILEPATH, O_RDONLY, 0666);
13
14      void *shared_memory = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0);
15      printf("Received message: %s\n", (char *)shared_memory);
16
17      munmap(shared_memory, SIZE);
18      close(fd);
19
20      return 0;
21  }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5
6  #define MAX 10
7  #define LOOPS 20
8
9  int buffer[MAX];
10 int fill = 0;
11 int use = 0;
12
13 sem_t empty;
14 sem_t full;
15
16 void put(int value) {
17     buffer[fill] = value;
18     fill = (fill + 1) % MAX;
19 }
20
21 int get() {
22     int tmp = buffer[use];
23     use = (use + 1) % MAX;
24     return tmp;
25 }
26
27 void *producer(void *arg) {
```

```

28     int i;
29     for (i = 0; i < LOOPS; i++) {
30         sem_wait(&empty);
31         put(i);
32         printf("Produced: %d\n", i);
33         sem_post(&full);
34     }
35     return NULL;
36 }
37
38 void *consumer(void *arg) {
39     int tmp = 0;
40     for (int i = 0; i < LOOPS; i++) {
41         sem_wait(&full);
42         tmp = get();
43         printf("Consumed: %d\n", tmp);
44         sem_post(&empty);
45     }
46     return NULL;
47 }
48
49 int main(int argc, char *argv[]) {
50     pthread_t prod, cons;
51
52     sem_init(&empty, 0, MAX);
53     sem_init(&full, 0, 0);
54
55     pthread_create(&prod, NULL, producer, NULL);
56     pthread_create(&cons, NULL, consumer, NULL);
57
58     pthread_join(prod, NULL);
59     pthread_join(cons, NULL);
60
61     sem_destroy(&empty);
62     sem_destroy(&full);
63
64     return 0;
65 }

```

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

Solution: Use mutex locks around it to create mutual exclusion

```

1 void *producer(void *arg) {
2     int i;
3     for (i = 0; i < loops; i++) {
4         sem_wait(&mutex);

```

```

5     sem_wait(&empty);
6     put(i);
7     sem_post(&full);
8     sem_post(&mutex);
9 }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&mutex);
16         sem_wait(&full);
17         int tmp = get();
18         sem_post(&empty);
19         sem_post(&mutex);
20         printf("%d\n", tmp);
21     }
22 }

```

Please don't lock and then wait for someone to do something.... since the other guy won't be able to do anything.... since you have locked....

It's just like waiting for Krishna when you have locked him outside...

```

1 void *producer(void *arg) {
2     int i;
3     for (i = 0; i < loops; i++) {
4         sem_wait(&empty);
5         sem_wait(&mutex);
6         put(i);
7         sem_post(&mutex);
8         sem_post(&full);
9     }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);
16         sem_wait(&mutex);
17         int tmp = get();
18         sem_post(&mutex);
19         sem_post(&empty);
20         printf("%d\n", tmp);
21     }
22 }

```



```

1  #include <pthread.h>
2  #include <semaphore.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  #define N 5
7
8  sem_t forks[N];
9  sem_t mutex;
10
11 void *philosopher(void *arg) {
12     int id = *(int *)arg;
13     int left = id;
14     int right = (id + 1) % N;
15
16     for (int i = 0; i < 3; i++) {
17         printf("Philosopher %d is thinking.\n", id);
18         usleep(100000);
19
20         sem_wait(&mutex);
21         sem_wait(&forks[left]);
22         sem_wait(&forks[right]);
23         sem_post(&mutex);
24
25         printf("Philosopher %d is eating.\n", id);
26         usleep(100000);
27
28         sem_post(&forks[left]);
29         sem_post(&forks[right]);
30     }
31     return NULL;
32 }
33
34 int main() {
35     pthread_t threads[N];
36     int ids[N];
37
38     sem_init(&mutex, 0, 1);
39     for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);
40
41     for (int i = 0; i < N; i++) {
42         ids[i] = i;
43         pthread_create(&threads[i], NULL, philosopher, &ids[i]);
44     }
45
46     for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);

```

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

```
47
48     for (int i = 0; i < N; i++) sem_destroy(&forks[i]);
49     sem_destroy(&mutex);
50
51     return 0;
52 }
53
```

```
1  #include <pthread.h>
2  #include <semaphore.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  #define N 5
7
8  sem_t forks[N];
9
10 void *philosopher(void *arg) {
11     int id = *(int *)arg;
12     int left = id;
13     int right = (id + 1) % N;
14
15     for (int i = 0; i < 3; i++) {
16         printf("Philosopher %d is thinking.\n", id);
17         usleep(100000);
18
19         if (id % 2 == 0) {
20             sem_wait(&forks[left]);
21             sem_wait(&forks[right]);
22         } else {
23             sem_wait(&forks[right]);
24             sem_wait(&forks[left]);
25         }
26
27         printf("Philosopher %d is eating.\n", id);
28         usleep(100000);
29
30         sem_post(&forks[left]);
31         sem_post(&forks[right]);
32     }
33     return NULL;
34 }
35
```

```

36  int main() {
37      pthread_t threads[N];
38      int ids[N];
39
40      for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);
41
42      for (int i = 0; i < N; i++) {
43          ids[i] = i;
44          pthread_create(&threads[i], NULL, philosopher, &ids[i]);
45      }
46
47      for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);
48
49      for (int i = 0; i < N; i++) sem_destroy(&forks[i]);
50
51      return 0;
52  }

```

---

```

1  #include <pthread.h>
2  #include <semaphore.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  #define N 5
7
8  sem_t forks[N];
9
10 void *philosopher(void *arg) {
11     int id = *(int *)arg;
12     int left = id;
13     int right = (id + 1) % N;
14
15     for (int i = 0; i < 3; i++) {
16         printf("Philosopher %d is thinking.\n", id);
17         usleep(100000);
18
19         if (id == N - 1) {
20             sem_wait(&forks[right]);
21             sem_wait(&forks[left]);
22         } else {
23             sem_wait(&forks[left]);
24             sem_wait(&forks[right]);
25         }

```

```

26
27     printf("Philosopher %d is eating.\n", id);
28     usleep(100000);
29
30     sem_post(&forks[left]);
31     sem_post(&forks[right]);
32 }
33 return NULL;
34 }
35
36 int main() {
37     pthread_t threads[N];
38     int ids[N];
39
40     for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);
41
42     for (int i = 0; i < N; i++) {
43         ids[i] = i;
44         pthread_create(&threads[i], NULL, philosopher, &ids[i]);
45     }
46
47     for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);
48
49     for (int i = 0; i < N; i++) sem_destroy(&forks[i]);
50
51     return 0;
52 }

```

---

```

1  import threading
2
3  semaphore = threading.Semaphore(3)
4  shared_counter = 0
5
6  def increment():
7      global shared_counter
8      semaphore.acquire()
9      local_copy = shared_counter
10     local_copy += 1
11     shared_counter = local_copy
12     semaphore.release()
13
14     threads = [threading.Thread(target=increment) for _ in range(10)]
15     for thread in threads:

```

```
16     thread.start()
17 for thread in threads:
18     thread.join()
19
20 print(shared_counter)
```

---

```
1  import threading
2  import time
3
4  def compute_task():
5      total = 0
6      for i in range(10**7):
7          total += i
8      return total
9
10 def multithreaded_execution():
11     threads = [threading.Thread(target=compute_task) for _ in range(4)]
12     start_time = time.time()
13     for thread in threads:
14         thread.start()
15     for thread in threads:
16         thread.join()
17     return time.time() - start_time
18
19 if __name__ == "__main__":
20     threading_time = multithreaded_execution()
21     print({"Multithreading Time (s)": threading_time})
22
```

---

```
1  import time
2  from multiprocessing import Process
3
4  def compute_task():
5      total = 0
6      for i in range(10**8):
7          total += i
8      return total
9
10 def multiprocessing_execution():
11     processes = [Process(target=compute_task) for _ in range(4)]
```

```
12     start_time = time.time()
13     for process in processes:
14         process.start()
15     for process in processes:
16         process.join()
17     return time.time() - start_time
18
19 if __name__ == "__main__":
20     multiprocessing_time = multiprocessing_execution()
21     print({"Multiprocessing Time (s)": multiprocessing_time})
```

---

```
1  from multiprocessing import Process, Semaphore
2  import time
3
4  def worker(sem, task_id):
5      sem.acquire()
6      print(f"Process {task_id} started")
7      time.sleep(2)
8      print(f"Process {task_id} finished")
9      sem.release()
10
11 if __name__ == "__main__":
12     sem = Semaphore(2)
13     processes = [Process(target=worker, args=(sem, i)) for i in range(5)]
14     for p in processes:
15         p.start()
16     for p in processes:
17         p.join()
```

---