```
//        ONE
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

#define SEM_MUTEX_KEY "/tmp/sem-mutex-key"
#define SEM_BUFFER_COUNT_KEY "/tmp/sem-buffer-count-key"
#define SEM_SPOOL_SIGNAL_KEY "/tmp/sem-spool-signal-key"
#define MAX_BUFFERS 10

char buf[MAX_BUFFERS][100];
int buffer_index;
int buffer_print_index;
int mutex_sem, buffer_count_sem, spool_signal_sem;

void *producer(void *arg);
void *spooler(void *arg);

int main() {
    key_t s_key;
    union semun {
        int val;
        struct semid_ds *buf;
        ushort array[1];
    } sem_attr;
    pthread_t tid_producer[10], tid_spooler;
    int i, r;

    buffer_index = buffer_print_index = 0;

    if ((s_key = ftok(SEM_MUTEX_KEY, 'a')) == -1) {
        perror("ftok");
        exit(1);
    }
    if ((mutex_sem = semget(s_key, 1, 0660 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }
    sem_attr.val = 1;
    if (semctl(mutex_sem, 0, SETVAL, sem_attr) == -1) {
        perror("semctl SETVAL");
        exit(1);
    }

    if ((s_key = ftok(SEM_BUFFER_COUNT_KEY, 'a')) == -1) {
        perror("ftok");
        exit(1);
```

```
}
if ((buffer_count_sem = semget(s_key, 1, 0660 | IPC_CREAT)) == -1) {
    perror("semget");
    exit(1);
}
sem_attr.val = MAX_BUFFERS;
if (semctl(buffer_count_sem, 0, SETVAL, sem_attr) == -1) {
    perror(" semctl SETVAL ");
    exit(1);
}

if ((s_key = ftok(SEM_SPOOL_SIGNAL_KEY, 'a')) == -1) {
    perror("ftok");
    exit(1);
}
if ((spool_signal_sem = semget(s_key, 1, 0660 | IPC_CREAT)) == -1) {
    perror("semget");
    exit(1);
}
sem_attr.val = 0;
if (semctl(spool_signal_sem, 0, SETVAL, sem_attr) == -1) {
    perror(" semctl SETVAL ");
    exit(1);
}

if ((r = pthread_create(&tid_spooler, NULL, spooler, NULL)) != 0) {
    fprintf(stderr, "Error = %d (%s)\n", r, strerror(r));
    exit(1);
}

int thread_no[10];
for (i = 0; i < 10; i++) {
    thread_no[i] = i;
    if ((r = pthread_create(&tid_producer[i], NULL, producer, (void *)&thread_no[i])) != 0) {
        fprintf(stderr, "Error = %d (%s)\n", r, strerror(r));
        exit(1);
    }
}

for (i = 0; i < 10; i++) {
    if ((r = pthread_join(tid_producer[i], NULL)) == -1) {
        fprintf(stderr, "Error = %d (%s)\n", r, strerror(r));
        exit(1);
    }
}

struct sembuf asem[1];
asem[0].sem_num = 0;
asem[0].sem_op = 0;
asem[0].sem_flg = 0;
if (semop(spool_signal_sem, asem, 1) == -1) {
    perror("semop: spool_signal_sem");
```

```
        exit(1);
    }

    if ((r = pthread_cancel(tid_spooler)) != 0) {
        fprintf(stderr, "Error = %d (%s)\n", r, strerror(r));
        exit(1);
    }

    if (semctl(mutex_sem, 0, IPC_RMID) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }
    if (semctl(buffer_count_sem, 0, IPC_RMID) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }
    if (semctl(spool_signal_sem, 0, IPC_RMID) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }

    exit(0);
}

void *producer(void *arg) {
    int i;
    int my_id = *((int *)arg);
    struct sembuf asem[1];
    int count = 0;

    asem[0].sem_num = 0;
    asem[0].sem_op = 0;
    asem[0].sem_flg = 0;

    for (i = 0; i < 10; i++) {
        asem[0].sem_op = -1;
        if (semop(buffer_count_sem, asem, 1) == -1) {
            perror("semop: buffer_count_sem");
            exit(1);
        }

        asem[0].sem_op = -1;
        if (semop(mutex_sem, asem, 1) == -1) {
            perror("semop: mutex_sem");
            exit(1);
        }

        int j = buffer_index;
        buffer_index++;
        if (buffer_index == MAX_BUFFERS)
            buffer_index = 0;
        asem[0].sem_op = 1;
        if (semop(mutex_sem, asem, 1) == -1) {
            perror("semop: mutex_sem");
            exit(1);
        }

        sprintf(buf[j], "Thread %d: %d\n", my_id, ++count);
        asem[0].sem_op = 1;
        if (semop(spool_signal_sem, asem, 1) == -1) {
            perror("semop: spool_signal_sem");
            exit(1);
        }

        sleep(1);
    }
}

void *spooler(void *arg) {
    struct sembuf asem[1];

    asem[0].sem_num = 0;
    asem[0].sem_op = 0;
    asem[0].sem_flg = 0;

    while (1) {
        asem[0].sem_op = -1;
        if (semop(spool_signal_sem, asem, 1) == -1) {
            perror("semop: spool_signal_sem");
            exit(1);
        }

        printf("%s", buf[buffer_print_index]);

        buffer_print_index++;
        if (buffer_print_index == MAX_BUFFERS)
            buffer_print_index = 0;

        asem[0].sem_op = 1;
        if (semop(buffer_count_sem, asem, 1) == -1) {
            perror("semop: buffer_count_sem");
            exit(1);
        }
    }
}
```

// TWO

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

```
#include <stdbool.h>

#define NUM_PHILOSOPHERS 5
#define EATING 0
#define THINKING 1
#define HUNGRY 2

int state[NUM_PHILOSOPHERS];
sem_t mutex;
sem_t chopsticks[NUM_PHILOSOPHERS];
int total_eat_count = 0; // Total number of times all philosophers have eaten

void grab_forks(int philosopher_id) {
    sem_wait(&mutex);
    state[philosopher_id] = HUNGRY;
    printf("Philosopher %d is hungry.\n", philosopher_id);
    test(philosopher_id);
    sem_post(&mutex);
    sem_wait(&chopsticks[philosopher_id]);
}

void put_away_forks(int philosopher_id) {
    sem_wait(&mutex);
    state[philosopher_id] = THINKING;
    printf("Philosopher %d is thinking.\n", philosopher_id);
    test((philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS);
    test((philosopher_id + 1) % NUM_PHILOSOPHERS);
    sem_post(&mutex);
}

void test(int philosopher_id) {
    if (state[philosopher_id] == HUNGRY &&
        state[(philosopher_id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] !=
EATING &&
        state[(philosopher_id + 1) % NUM_PHILOSOPHERS] != EATING) {
        state[philosopher_id] = EATING;
        printf("Philosopher %d is eating.\n", philosopher_id);
        sem_post(&chopsticks[philosopher_id]);
        total_eat_count++;
    }
}

void* philosopher(void* arg) {
    int philosopher_id = *((int*)arg);
    while (total_eat_count < NUM_PHILOSOPHERS) {
        // Thinking
        sleep(1);

        // Hungry and trying to eat
        grab_forks(philosopher_id);

        // Eating
        sleep(1);

        // Done eating, put away forks
        put_away_forks(philosopher_id);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    sem_init(&mutex, 0, 1);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 0);
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    return 0;
}
```

//**THREE**

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

#define N 10

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
int count = 0;
char buf[N];

void monenter() {
    pthread_mutex_lock(&mutex);
}

void monexit() {
    pthread_mutex_unlock(&mutex);
}

void moninsert(char alpha) {
    monenter();
    while (count == N) {
        printf("Buffer is full. Waiting...\n");
        pthread_cond_wait(&full, &mutex);
    }
    buf[(count++) % N] = alpha; // Insert alpha into buf, wrapping around when necessary
    printf("Produced: %c\n", alpha);
    if (count == 1) {
        pthread_cond_signal(&empty);
    }
    monexit();
}

char monremove() {
    monenter();
    while (count == 0) {
        printf("Buffer is empty. Waiting...\n");
        pthread_cond_wait(&empty, &mutex);
    }
    char item = buf[--count % N]; // Remove an item from buf, wrapping around when necessary
    printf("Consumed: %c\n", item);
    if (count == N - 1) {
        pthread_cond_signal(&full);
    }
    monexit();
    return item;
}

void* producer(void* arg) {
    while (1) {
        char item = 'A' + rand() % 26; // Produce a random uppercase letter
        moninsert(item);
    }
    return NULL;
}

void* consumer(void* arg) {
    while (1) {
        char item = monremove();
    }
    return NULL;
}

int main() {
    pthread_t producer_threads[6];
    pthread_t consumer_threads[6];

    for (int i = 0; i < 6; i++) {
        pthread_create(&producer_threads[i], NULL, producer, NULL);
        pthread_create(&consumer_threads[i], NULL, consumer, NULL);
    }

    for (int i = 0; i < 6; i++) {
        pthread_join(producer_threads[i], NULL);
        pthread_join(consumer_threads[i], NULL);
    }
    return 0;
}
```

// FOUR

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

#define NUM_PHILOSOPHERS 5
#define MAX_EAT_COUNT 1

pthread_mutex_t forks[NUM_PHILOSOPHERS];
int eat_count[NUM_PHILOSOPHERS] = {0};

void monpickup(int philosopher_id) {
    pthread_mutex_lock(&forks[philosopher_id]);
    pthread_mutex_lock(&forks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);
}

void monputdown(int philosopher_id) {
    pthread_mutex_unlock(&forks[philosopher_id]);
    pthread_mutex_unlock(&forks[(philosopher_id + 1) % NUM_PHILOSOPHERS]);
}

void* philosopher(void* arg) {
    int philosopher_id = *(int*)arg;
    while (eat_count[philosopher_id] < MAX_EAT_COUNT) {
        // Think
        printf("Philosopher %d is thinking.\n", philosopher_id);
        // Pick up forks
        monpickup(philosopher_id);
        // Eat
        printf("Philosopher %d is eating.\n", philosopher_id);
        eat_count[philosopher_id]++;
        // Put down forks
        monputdown(philosopher_id);
    }
    return NULL;
}

int main() {
    pthread_t philosopher_threads[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
        philosopher_ids[i] = i;
        pthread_create(&philosopher_threads[i], NULL, philosopher, &philosopher_ids[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosopher_threads[i], NULL);
    }

    return 0;
}
// FIVE
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("locatoin of code: %p\n",(void*) main);
    printf("location of head: %p\n",(void*) malloc(1));

    int z=3;
    printf("location of the stack: %p\n",(void*) &z);

    int *x, *y;

    x = malloc(50 * sizeof(int));
    if(!x) {
        perror("malloc");
        return -1;
    }
    y = calloc(50,sizeof(int));
    if(!y) {
        perror("calloc");
        return -1;
    }

    for(int i = 0; i<50; i++){
        printf("%d", *x);
        x++;
    }printf("\n");
    for(int i = 0; i<50; i++){
        printf("%d", *y);
        y++;
    }printf("\n");

    return 0;
}
```