Study tools 🗸 Home

My courses ∨ My books My folder

Life Career



Find solutions for your homework

Search

home / study / engineering / computer science / computer science questions and answers / the bounded-buffer solution in the below code uses a...

Question: The bounded-buffer solution in the below code uses a last-in-...

The bounded-buffer solution in the below code uses a last-in-first-out strategy (LIFO). Change the code to implement a FIFO (First-in-First-out) strategy. You may use the (in,out) pointer method (using semaphores to test if the queue is full or empty should alleviate the problem of only using up N-1 locations) or implement a FIFO queue. Use the correct counting semaphore implementation.

C code below

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

#define SIZE 5 #define NUMB THREADS 6 #define PRODUCER_LOOPS 2

typedef int buffer_t; buffer_t buffer[SIZE]; int buffer_index;

pthread_mutex_t buffer_mutex; /* initially buffer will be empty. full_sem will be initialized to buffer SIZE, which means SIZE number of producer threads can write to it. And empty_sem will be initialized to 0, so no consumer can read from buffer until a producer thread posts to empty sem */ sem_t full_sem; /* when 0, buffer is full */ sem_t empty_sem; /* when 0, buffer is empty. Kind of like an index for the buffer */

```
void insertbuffer(buffer t value) {
if (buffer_index < SIZE) {</pre>
buffer[buffer_index++] = value;
} else {
printf("Buffer overflow\n");
buffer_t dequeuebuffer() {
if (buffer index > 0) {
return buffer[--buffer_index]; // buffer_index-- would be error!
} else {
printf("Buffer underflow\n");
return 0;
void *producer(void *thread_n) {
int thread_numb = *(int *)thread_n;
buffer_t value;
int i=0:
while (i++ < PRODUCER_LOOPS) {
sleep(rand() % 10);
value = rand() % 100;
sem_wait(&full_sem); // sem=0: wait. sem>0: go and decrement it
/* possible race condition here. After this thread wakes up,
another thread could agcuire mutex before this one, and add to list.
Then the list would be full again
and when this thread tried to insert to buffer there would be
a buffer overflow error */
```

Post a question Answers from our experts for your tough homework questions Enter question Continue to post 20 questions remaining



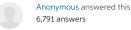
2nd Edition

6th Edition

16th Edition View all solutions

```
Home
                                   Study tools >
                                                      My courses ∨
                                                                         My books My folder
                                                                                                         Career
                                                                                                                   Life
sem post(&empty sem); // post (increment) emptybuffer semaphore
printf("Producer %d added %d to buffer\n", thread_numb, value);
pthread_exit(0);
void *consumer(void *thread_n) {
int thread_numb = *(int *)thread_n;
buffer_t value;
while (i++ < PRODUCER LOOPS) {
sem_wait(&empty_sem);
/* there could be race condition here, that could cause
buffer underflow error */
pthread_mutex_lock(&buffer_mutex);
value = dequeuebuffer(value);
pthread mutex unlock(&buffer mutex);
sem_post(&full_sem); // post (increment) fullbuffer semaphore
printf("Consumer %d dequeue %d from buffer\n", thread_numb, value);
pthread_exit(0);
int main(int argc, int **argv) {
buffer_index = 0;
pthread_mutex_init(&buffer_mutex, NULL);
sem_init(&full_sem, // sem_t *sem
0, // int pshared. 0 = shared between threads of process, 1 = shared between processes
SIZE); // unsigned int value. Initial value
sem_init(&empty_sem,
0,
0):
/* full_sem is initialized to buffer size because SIZE number of
producers can add one element to buffer each. They will wait
semaphore each time, which will decrement semaphore value.
empty_sem is initialized to 0, because buffer starts empty and
consumer cannot take any element from it. They will have to wait
until producer posts to that semaphore (increments semaphore
value) */
pthread_t thread[NUMB_THREADS];
int thread_numb[NUMB_THREADS];
int i;
for (i = 0; i < NUMB\_THREADS;) {
thread\_numb[i] = i;
pthread_create(thread + i, // pthread_t *t
NULL, // const pthread_attr_t *attr
producer, // void *(*start_routine) (void *)
thread_numb + i); // void *arg
thread_numb[i] = i;
// playing a bit with thread and thread_numb pointers...
pthread_create(&thread[i], // pthread_t *t
NULL, // const pthread_attr_t *attr
consumer, // void *(*start_routine) (void *)
&thread_numb[i]); // void *arg
i++;
for (i = 0; i < NUMB_THREADS; i++)
pthread_join(thread[i], NULL);
pthread_mutex_destroy(&buffer_mutex);
sem_destroy(&full_sem);
sem_destroy(&empty_sem);
return 0;
```

Expert Answer (1)



Was this answer helpful?





Please find the answer which is as shown below, Thankyou.



Home Study tools 🗸

My courses ✓ My books My folder

Career Life



```
#include linux/pipe_fs_i.h>
static void wait_for_partner(struct inode* inode, unsigned int *cnt)
    int cur = *cnt;
    while (cur == *cnt) {
        pipe_wait(inode->i_pipe);
        if (signal_pending(current))
             break;
static void wake_up_partner(struct inode* inode)
    wake_up_interruptible(&inode->i_pipe->wait);
static int fifo_open(struct inode *inode, struct file *filp)
    struct pipe_inode_info *pipe;
    int ret:
    mutex_lock(&inode->i_mutex);
    pipe = inode->i_pipe;
    if (!pipe) {
        ret = -ENOMEM;
        pipe = alloc_pipe_info(inode);
        if (!pipe)
             goto err_nocleanup;
        inode->i_pipe = pipe;
    filp->f_version = 0;
    /* We can only do regular read/write on fifos */
    filp->f_mode &= (FMODE_READ | FMODE_WRITE);
    switch (filp->f_mode) {
    case FMODE_READ:
    * O_RDONLY
      POSIX.1 says that O_NONBLOCK means return with the FIFO
      opened, even when there is no process writing the FIFO.
        filp->f_op = &read_pipefifo_fops;
        pipe->r_counter++;
        if (pipe->readers++ == 0)
             wake_up_partner(inode);
        if (!pipe->writers) {
             if ((filp->f_flags & O_NONBLOCK)) {
                  /* suppress POLLHUP until we have
                  * seen a writer */
                  filp->f_version = pipe->w_counter;
             } else
                  wait_for_partner(inode, &pipe->w_counter);
                  if(signal_pending(current))
                      goto err_rd;
        break;
    case FMODE_WRITE:
    * O_WRONLY
      POSIX.1 says that O_NONBLOCK means return -1 with
      errno=ENXIO when there is no process reading the FIFO.
        ret = -ENXIO:
        if ((filp->f_flags & O_NONBLOCK) && !pipe->readers)
             aoto err:
        filp->f_op = &write_pipefifo_fops;
        pipe->w_counter++;
        if (!pipe->writers++)
             wake_up_partner(inode);
```

```
Home Study tools ✓
                                             My courses ✓ My books My folder
                                                                                                Career Life
             if (signal_pending(current))
                  goto err_wr;
         break:
    case FMODE READ | FMODE WRITE:
     * POSIX.1 leaves this case "undefined" when O_NONBLOCK is set.
     * This implementation will NEVER block on a O_RDWR open, since
     * the process can at least talk to itself.
         filp->f_op = &rdwr_pipefifo_fops;
         pipe->readers++;
         pipe->writers++;
         pipe->r_counter++;
         pipe->w_counter++;
         if (pipe->readers == 1 || pipe->writers == 1)
             wake_up_partner(inode);
         break;
    default:
         ret = -EINVAL;
         goto err;
    /* Ok! */
    mutex_unlock(&inode->i_mutex);
    return 0;
err_rd:
    if (!--pipe->readers)
         wake_up_interruptible(&pipe->wait);
    ret = -ERESTARTSYS;
    goto err;
err wr:
    if (!--pipe->writers)
         wake_up_interruptible(&pipe->wait);
    ret = -ERESTARTSYS;
    goto err;
err:
    if (!pipe->readers && !pipe->writers)
         free_pipe_info(inode);
err_nocleanup:
    mutex_unlock(&inode->i_mutex);
    return ret;
\ensuremath{^{*}} Dummy default file-operations: the only thing this does
* is contain the open that then fills in the correct operations
* depending on the access mode of the file...
const struct file_operations def_fifo_fops = {
                = fifo_open, /* will set read_ or write_pipefifo_fops */
};
```

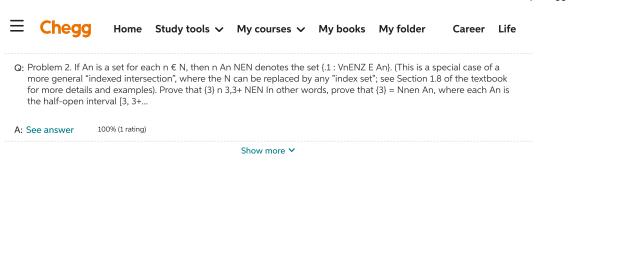
Hope you got some explanation, still having doubts, mention in comment section, happy to resolve them, Thankyou.

Thankyou. Happy Learning

View comments (1) >

Questions viewed by other students

Q: A condensate line 152 mm nominal size made of schedule 40 carbon steel pipe is supported by threaded rod hangers spaced at 2.5 m center-to-center. The hangers are carbon steel, 50 cm long, with a root diameter of 12 mm.



COMPANY~

LEGAL & POLICIES

CHEGG PRODUCTS AND SERVICES

CHEGG NETWORK

CUSTOMER SERVICE~





© 2003-2022 Chegg Inc. All rights reserved.