CS415 Module 4 Part B - Problems and Pitfalls of Concurrency

Athens State University

Outline

Contents

1 The Reader-Writer Problem

The Reader-Writer Problem

- A data area is shared among many processes
  - Some processes only read the data area (readers) and some only write to the data area (writers)
- Conditions that must be satisfied
  1. Any number of readers may simultaneously read the file
  2. Only one writer at a time my write to the file
  3. If a writer is writing to the file, no reader may read it

Priority Polices

- **Read-preferring reader-writer** Give priority to readers
- **Write-preferring reader-writer** Give priority to writers
- **Unspecified policy reader-writer** Make no call about who gets to read or write. makes things easier on the implmentation

Read-preferring RW locks allow for maximum concurrency, but can lead to write-starvation if contention is high. This is because writer threads will not be able to acquire the lock as long as at least one reading thread holds it. Since multiple reader threads may hold the lock at once, this means that a writer thread may continue waiting for the lock while new reader threads are able to acquire the lock, even to the point where the writer may still be waiting after all of the readers which were holding the lock when it first attempted to acquire it have released the lock. Priority to readers may be weak, as just described, or strong, meaning that whenever a writer releases the lock, any blocking readers always acquire it next.

Write-preferring RW locks avoid the problem of writer starvation by preventing any new readers from acquiring the lock if there is a writer queued and waiting for the lock; the writer will acquire the lock as soon as all readers which were already holding the lock have completed.[4] The downside is that write-preferring locks allows for less concurrency in the presence of writer threads, compared to read-preferring RW locks. Also the lock is less performant because each operation, taking or releasing the lock for either read or write, is more complex, internally requiring taking and releasing two mutexes instead of one.[citation needed] This variation is sometimes also known as "write-biased" readers–writer lock.

Unspecified priority RW locks does not provide any guarantees with regards read vs. write access. Unspecified priority can in some situations be preferable if it allows for a more efficient implementation.

`Pthreads` Read-write Locks

```c
pthread_rwlock_t  plock;

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        pthread_rwlockattr_t *attr);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

- A RW Lock can be one of three states: Unlocked, Read Locked, and Write Locked
- Don't try to do a read lock if you already have a write lock
- Don't try to do a write lock if you already have a read lock

1.1 Case Study: Using `Pthreads` RW Locks In a Job Queue

Case Study: Using RW Locks to protect a job queue

- In this example, we do a raw implementation of a job queue
- We permit multiple threads to use this queue
- So, need to protect adding or removing a job from the queue
- We do this using a `Pthread` Read/Write Lock

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

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t   j_id;   /* tells which thread handles this job */
    /* ... more stuff here ... */
};

struct queue {
    struct job      *q_head;
    struct job      *q_tail;
    pthread_rwlock_t q_lock;
};
```

```c

/*
 * Initialize a queue.
 */
int
queue_init(struct queue *qp)
{
    int err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);
    /* ... continue initialization ... */
    return(0);
}
```

```c
//Insert a job at the head of the queue.
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp;    /* list was empty */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}
```

```c
// Append a job on the tail of the queue.
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp;    /* list was empty */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}
```

```c
// Remove the given job from a queue.
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
        else
            jp->j_next->j_prev = jp->j_prev;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        jp->j_prev->j_next = jp->j_next;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}
```

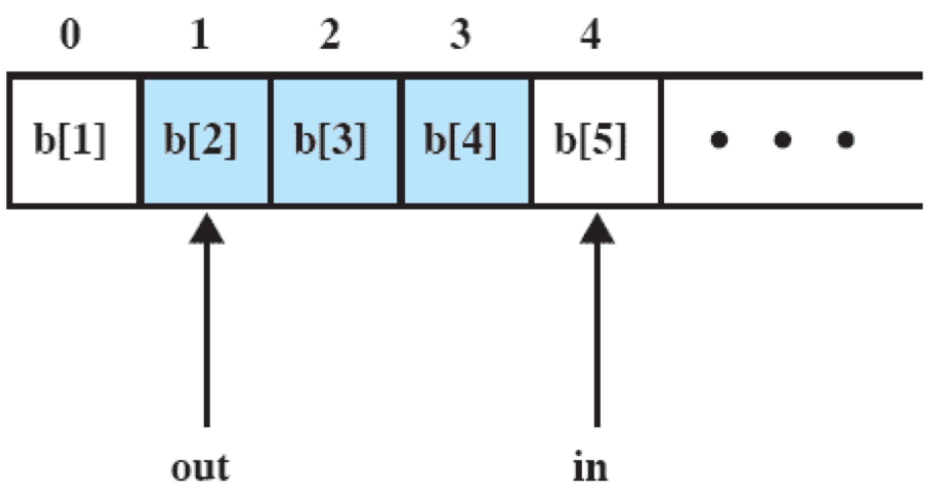
```
// Find a job for the given thread ID.
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

2 The Producer-Consumer Problem

The Producer-Consumer Problem

- General Situation:
  - One or more producers are generating data items and pumping them into a buffer
  - A single consumer is taking items out of the buffer one at a time
  - Only one producer or consumer may access the buffer at any one time
- The Problem:
  - Need to ensure that the producer can't add data into a full buffer and the consumer can't remove data from an empty buffer

Buffer Structure



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the consumer makes sure that the producer has advanced beyond it (in $>$ out) before proceeding.

Condition Variables

- **Condition Variable**: A data types that is used to block a process or thread until a particular condition is true
- Contrast with a mutex:
  - A mutex lets threads synchronize by controlling access to data
  - A condition variable lets threads synchronize on the value of data
- One must use mutexes and condition variables in concert. This avoids a deadlock situation

A `Pthreads` Solution to the Producer/Consumer Problem

```
#define MAX 10000000000      /* Numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;

void* producer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex); /* protect buffer */
    while (buffer != 0)           /* If there is something
            in the buffer then wait */
      pthread_cond_wait(&condp, &the_mutex);
    buffer = i;
    pthread_cond_signal(&condc);   /* wake up consumer */
    pthread_mutex_unlock(&the_mutex); /* release the buffer */
  }
  pthread_exit(0);
}
```

A `Pthreads` Solution to the Producer/Consumer Problem

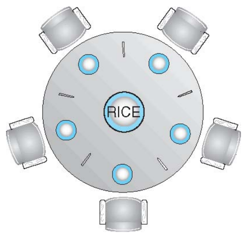
```
void* consumer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex); /* protect buffer */
    while (buffer == 0)      /* If there is nothing in
             the buffer then wait */
      pthread_cond_wait(&condc, &the_mutex);
    buffer = 0;
    pthread_cond_signal(&condp);   /* wake up consumer */
    pthread_mutex_unlock(&the_mutex); /* release the buffer */
  }
  pthread_exit(0);
}
```

A `Pthreads` Solution to the Producer/Consumer Problem

```
int main(int argc, char **argv) {
  pthread_t pro, con;
  // Initialize the mutex and condition variables
  /* What's the NULL for ??? */
  pthread_mutex_init(&the_mutex, NULL);
  /* Initialize consumer condition variable */
  pthread_cond_init(&condc, NULL);
  /* Initialize producer condition variable */
  pthread_cond_init(&condp, NULL);
  // Create the threads
  pthread_create(&con, NULL, consumer, NULL);
  pthread_create(&pro, NULL, producer, NULL);
  // Wait for the threads to finish
  // Otherwise main might run to the end
  // and kill the entire process when it exits.
  pthread_join(&con, NULL);
  pthread_join(&pro, NULL);
  // Cleanup -- would happen automatically at end of program
  pthread_mutex_destroy(&the_mutex);
  pthread_cond_destroy(&condc);
  pthread_cond_destroy(&condp);
}
```

3 The Dining Philosophers Problem

The Dining Philosophers Problem



- Philosophers spend their lives thinking and eating
- And they don't like their neighbors, so will try to pick up 2 chopsticks (one at a time) to eat from a bowl

The dining philosophers problem is a classic thought experiment dealing with synchronization of processes. It is a rather meaningless problem at face value but is typical of the type of synchronization problems common when allocating resources in operating systems.

3.1 Case Study: Dining Philosophers in `Pthreads`

Case Study: Using `Pthreads` To Examine Dining Philosophers

- The code in this case study, adapted from work done for the University of Tennessee's course in operating systems, uses pthreads to implement a general driver and several possible "solutions" for the dining philosopher's problem.
- The driver is in the file `dphil_skeleton.c`. The philosophers are represented in a `Phil_struct` structure. The skeleton calls the `initialize_v()` function, which is undefined in the skeleton. This functions returns a `void *` function which is stored as part of the `Phil_struct` structure.
- The driver skeleton forks off a set philosopher threads (the total number of which is a command-line argument). After doing so, the main thread sleeps for 10 seconds and prints out information about how long each philosopher has been blocked waiting to eat.

Case Study: How the philosophers operate

Each philosopher thread implements the following algorithm:

```
while(1) {
    think(randomNumberOfSeconds);
    pickup(p);
    eat(randomNuberOfSeconds);
    putdown(p);
}
```

Each variation of the problem we will examine will need to implement the `initialize_v()`, `pickup()`, and `putdown()` functions to manage the chopsticks and avoid deadlock

Case Study: Use a mutex per chopstick

Our first attempt at solving this problem will have each philosopher pick up the left hand chopstick, then the right hand chopstick, eat, and put the chopsticks back on the table in reverse order.
We try, without success to avoid deadlocks by having odd-numbered philosophers pick up chopsticks in a left-to-right order and even-numbered philosophers pick up chopsticks in right-to-left order.

Case Study: Use a mutex per chopstick

```c
void *initialize_v(int phil_count) {
  Sticks *pp;
  int i;
  pp = (Sticks *) malloc(sizeof(Sticks));
  pp->phil_count = phil_count;
  pp->lock =
     (pthread_mutex_t **) malloc(sizeof(pthread_mutex_t *)*phil_count);
  if (pp->lock == NULL) { perror("malloc"); exit(1); }
  for (i = 0; i < phil_count; i++) {
    pp->lock[i] = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
    if (pp->lock[i] == NULL) { perror("malloc"); exit(1); }
  }
  for (i = 0; i < phil_count; i++) {
    pthread_mutex_init(pp->lock[i], NULL);
  }
  return (void *) pp;
}
```

Case Study: Use a mutex per chopstick

```c
void pickup(Phil_struct *ps)
{
  Sticks *pp;
  int phil_count;
  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;
  if (ps->id % 2 == 1) {
    /* lock up left stick */
    pthread_mutex_lock(pp->lock[ps->id]);
    /* lock right stick */
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]);
  } else {
    /* lock right stick */
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]);
    /* lock up left stick */
    pthread_mutex_lock(pp->lock[ps->id]);
  }
}
```

Case Study: Use a mutex per chopstick

```c
% void putdown(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  if (ps->id % 2 == 1) {
    /* unlock right stick */
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]);
    /* unlock left stick */
    pthread_mutex_unlock(pp->lock[ps->id]);
  } else {
    /* unlock left stick */
    pthread_mutex_unlock(pp->lock[ps->id]);
    /* unlock right stick */
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]);
  }
}
```

Case Study: Using a mutex per chopstick

If we run our simulation, we find this approach suffers from starvation. One philosopher never gets a chance to eat.
So, what asked the philosophers to check to see if both chopsticks are available when they are ready to eat. Doing so requires the use of condition variables.

Case Study: Using condition variables to prevent starvation

```
% #include <stdio.h>
#include <pthread.h>
#include "dphil.h"


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

typedef struct {
  pthread_mutex_t *mon;
  pthread_cond_t **cv;
  int *state;
  int phil_count;
} Phil;
```

Case Study: Using condition variables to prevent starvation

```
void pickup(Phil_struct *ps)
{
  Phil *pp;
  int phil_count;

  pp = (Phil *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_lock(pp->mon);
  pp->state[ps->id] = HUNGRY;
  while (pp->state[(ps->id + (phil_count-1))%phil_count] == EATING ||
         pp->state[(ps->id + 1)%phil_count] == EATING) {
    pthread_cond_wait(pp->cv[ps->id], pp->mon);
  }
  pp->state[ps->id] = EATING;
  pthread_mutex_unlock(pp->mon);
}
```

Case Study: Using condition variables to prevent starvation

```c
void putdown(Phil_struct *ps)
{
  Phil *pp;
  int phil_count;

  pp = (Phil *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_lock(pp->mon);
  pp->state[ps->id] = THINKING;
  if (pp->state[(ps->id+(phil_count-1))%phil_count] == HUNGRY) {
    pthread_cond_signal(pp->cv[(ps->id+(phil_count-1))%phil_count]);
  }
  if (pp->state[(ps->id+1)%phil_count] == HUNGRY) {
    pthread_cond_signal(pp->cv[(ps->id+1)%phil_count]);
  }
  pthread_mutex_unlock(pp->mon);
}
```

Case Study: Using condition variables to prevent starvation

```c

void *initialize_v(int phil_count)
{
  Phil *pp;
  int i;

  pp = (Phil *) malloc(sizeof(Phil));
  pp->phil_count = phil_count;
  pp->mon = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  pp->cv = (pthread_cond_t **) malloc(sizeof(pthread_cond_t *)*phil_count);
  if (pp->cv == NULL) { perror("malloc"); exit(1); }
  pp->state = (int *) malloc(sizeof(int)*phil_count);
  if (pp->state == NULL) { perror("malloc"); exit(1); }
  pthread_mutex_init(pp->mon, NULL);
  for (i = 0; i < phil_count; i++) {
    pp->cv[i] = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (pp->cv[i] == NULL) { perror("malloc"); exit(1); }
    pthread_cond_init(pp->cv[i], NULL);
    pp->state[i] = THINKING;
  }

  return (void *) pp;
}
```

Case Study: Avoiding starvation

- There are two possible methods to preventing starvation:
    1. Make certain that the thread system will unblock monitors and condition variables in the same order that they are blocked
    2. Implement your own scheme for preventing starvation. For example, have the philosophers queue up when they are hungry and eat only if they are at the head of the queue and if the chopsticks are free

So, why is understanding dining philosophers important?

- This problem is a thought experiment (and exam question) that illustrates some of the ugly issues involved with avoiding deadlock
- It rears it's ugly head in issues where one has to deal with multiple processes that attempt to access circular buffers
- This is a common problem that occurs when having to deal with memory allocation and disk-drive access.

4 Key Points

Key Points

- Three problems: Reader-writer, producer-consumer, and dining philosophers
- And we how use synchronization primitives like those provided in `Pthreads` to solve these problems