CIS 307: Threads: Implementing Monitors using locks and condition variables

[Condition Variables], [Producer-Consumer with Protected Buffer], [Dining Philosophers], [Copying a file]

We have already studied monitors. Here we see how to implement monitors for threads using locks and condition variables. We will look at a producer thread communicating with a consumer thread through a protected buffer. We look also at a solution of the Dining Philosophers problem using threads and condition variables.
First we examine the commands we need for using condition variables.

Condition Variables

Condition Variables, as defined in the Pthreads package, can be used to implement simple forms of monitors. As you recall, a condition variable is used within a monitor to allow a process to sleep when unable to carry out an intended monitor operation. For example if a producer tries to put a new element in the buffer while the buffer is full, it will have to wait until some space becomes available. The wait takes place on a condition variable. When the wait is executed the buffer's mutex is opened to allow some other operation to get into the monitor.

Here are the basic commands for using conditions.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t * cond, const pthread_cond_attr *attr);
     Initialization of cond. Usually attr is initialized to
     pthread_condattr_default

#include <pthread.h>
int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);
     When this command is executed the executing thread goes to sleep
     on cond and simultaneously mutex is unlocked, thus allowing another
     thread to execute past a lock on mutex. When a thread is released
     from waiting on a condition variable, its mutex is implicitly locked.

#include <pthread.h>
int pthread_cond_signal(pthread_cond_t * cond);
     This command is null when no thread is asleep on cond. Otherwise
     a thread is released from cond.
```

We will use these commands to implement **Conditional Critical Regions**. These are atomic operation of the form

WHEN predicate DO action

which means that the action will be executed only when the predicate is true (a predicate is an expression that is either true or false). Assuming that **mutex** is a lock variable and **condition** is a condition variable, this can be done with

```
pthread_mutex_lock(mutex);   // We enter and lock
while (not predicate)      // while the predicate is not true
   pthread_cond_wait(condition, mutex); // we sleep and open the lock.
// At this point the predicate is true.
action;
pthread_cond_signal(condition); // After the action we signal to see
                     // if some other operation needs to enter
pthread_mutex_unlock(mutex);  // We unlock and exit. This unlock
                     // is null if the lock is currently held
                     // by the thread released by signal.
```

Note that if 5 threads are waiting on a condition at the time we signal that condition, then only one of these threads is released. It will check its predicate and, if false, it will go back to sleep opening the lock but without waking up another waiting thread. If we want all 5 threads to have a chance to check their predicates then we have to insert the appropriate signal commands or we have to use the **pthread_cond_broadcast** command which wakes up all threads currently waiting on a condition.

Producer and Consumer communicating through a Protected Buffer

Here we see a solution to the problem of implementing a Producer and Consumer communicating through a protected Buffer. [Main Program], [Protected Buffer], [Buffer],

The main program: Producer, Consumer, and Protected Buffer

```
/* pqueuepmain.c  -- Driver to test the pqueue       */

#include  <sys/types.h>
#include  <pthread.h>
#include  "qelem.h"
#include  "pqueuep.h"

void producer(void * a);
void consumer(void * a);

int main(void)
```

```
{
  pthread_t t1, t2;
  void *  pb;

  pb = pqueueinit(10);
  if (pthread_create(&t1, pthread_attr_default, (void *)producer, pb)!=0) {
    perror("pthread_create");
    exit(1);
  }
  if (pthread_create(&t2, pthread_attr_default, (void *)consumer, pb)!=0) {
    perror("pthread_create");
    exit(1);
  }

  /* Wait a while then exit: threads will die */
  sleep(60);
  printf("WE ARE DONE\n");
}

void producer(void * a) {
  while (1){
    pput(a, 'I');
    printf("I am thread producer\n");
    sleep(1);}
}

void consumer(void * a) {
  while (1){
    printf("I am thread consumer with %c\n", pget(a));
    sleep(3);}
}
```

where qelem.h is a file that defines the kind of item kept in the buffer [note that by just changing in this file char to int, or float, or any other elementary type, the program still works. But it will not, without additional precautions, if we use as element an array, or a complex structure]

```
/* qelem.h  -- Header file used to store type of elements of queues */
#define elemtype char
```

and pqueuep.h is a file that defines the interface to the protected buffer

```
/* pqueuep.h  -- Header file for protected circular buffer */

void * pqueueinit(int size);
void pput(void * fifo, elemtype v);
elemtype pget(void * fifo);
int pqueueempty(void * fifo);
int pqueuefull();
```

In this simple example the producer will rush ahead inserting items into the buffer while the consumer takes out a few items. Then the buffer becomes full and the producer and consumer will take turns inserting one item and

extracting one item.

The Protected Buffer

Here is how we can implement a protected buffer as a monitor using an unprotected buffer, a lock and a condition variable.

```
/* pqueuep.c  -- Code file for protected circular buffer */

#include <sys/types.h>
#include <pthread.h>
#include "qelem.h"
#include "queuep.h"

typedef struct pqueuestruct {
  void * q;
  pthread_mutex_t mutex;
  pthread_cond_t condition;
  } pqueue;

pqueue * pqueueinit(int size){
  pqueue * fifo = (pqueue *)malloc(sizeof(pqueue));

  fifo->q = queueinit(size);
  if(pthread_mutex_init(&(fifo->mutex), pthread_mutexattr_default) != 0) {
    perror("pthread_mutex_init");
    exit(1);}
  if(pthread_cond_init(&(fifo->condition), pthread_condattr_default) != 0) {
    perror("pthread_cond_init");
    exit(1);}
  return fifo;
}

void pput(pqueue * fifo, elemtype v) {
  pthread_mutex_lock(&(fifo->mutex));
  while (queuefull(fifo->q)) {
    pthread_cond_wait(&(fifo->condition), &(fifo->mutex));}
  put(fifo->q, v);
  pthread_cond_signal(&(fifo->condition));
  pthread_mutex_unlock(&(fifo->mutex));
}

elemtype pget(pqueue * fifo) {
  elemtype t;

  pthread_mutex_lock(&(fifo->mutex));
  while (queueempty(fifo->q)) {
    pthread_cond_wait(&(fifo->condition), &(fifo->mutex));}
  t = get(fifo->q);
  pthread_cond_signal(&(fifo->condition));
  pthread_mutex_unlock(&(fifo->mutex));
  return t;
}

int pqueueempty(pqueue * fifo) {
  int t;

  pthread_mutex_lock(&(fifo->mutex));
```

```
  t = queueempty(fifo->q);
  pthread_mutex_unlock(&(fifo->mutex));
  return t;
}

int pqueuefull(pqueue * fifo) {
  int t;

  pthread_mutex_lock(&(fifo->mutex));
  t = queuefull(&(fifo->q));
  pthread_mutex_unlock(&(fifo->mutex));
  return t;
}
```

where queuep.h specifies the interface to the circular buffer

```
/* queuep.h  -- Header file for circular buffer */

void * queueinit(int size);
void put(void * fifo, elemtype v);
elemtype get(void * fifo);
int queueempty(void * fifo);
int queuefull(void * fifo);
```

Note that in the protected buffer case we have one lock, one condition, and two predicates, pqueuefull and pqueueempty. In all other monitor problems we will have exactly one lock, any number of conditions, and any number of predicates. For example in the case of the Dining Philosophers we could use a condition per philosopher and a single predicate: **The current Philosopher is hungry and its neighbors are not eating**.

The unprotected Buffer

Of course you know how to implement an old fashioned circular buffer. Here is a possible body for it:

```
/* queuep.c  -- Code file for circular buffer */

#include     "qelem.h"

typedef struct queuestruct {
  int maxsize, head, tail, count;
  elemtype q[1000];} queue;
typedef queue * queuep;

queuep queueinit(int size){
  queuep fifo = (queuep)malloc(4*sizeof(int)+(size+1)*(sizeof(elemtype)));
  (fifo->maxsize) = size;
  (fifo->head) = 0; (fifo->tail) = 0; (fifo->count) = 0;}

void put(queuep fifo, elemtype v) {
  if ((fifo->count) <= (fifo->maxsize)) {
```

```
    (fifo->count)++;
    fifo->q[(fifo->tail)++] = v;
    if ((fifo->tail) > (fifo->maxsize)) (fifo->tail) = 0;}
}

elemtype get(queuep fifo) {
  elemtype t;
  if ((fifo->count) > 0) {
    (fifo->count)--;
    t = (fifo->q)[(fifo->head)++];
    if ((fifo->head) > (fifo->maxsize))(fifo->head)=0;
    return t;}
}

int queueempty(queuep fifo) {
  return ((fifo->count) == 0);
}

int queuefull(queuep fifo) {
  return ((fifo->count) == (fifo->maxsize));
}
```

Dining Philosophers

You all have seen the Dining Philosopher problem. Here you see a solution using threads and condition variables. It does not suffer of deadlocks, but there is the danger of livelocks. It is divided into three small files: [philtable.h], [philmain.c], [philtable.c]

[philtable.h]

```
/* philtable.h   -- Here are the calls we can make on the monitor
 *                  representing the dining philosophers
 */

void * tableinit(void *(*)(int *)); // argument is the function
                                    // representing the philosopher
void printstate(void);
void pickup(int k);
void putdown(int k);
```

[philmain.c]

```
/* philmain.c    */

#include  "philtable.h"

void * philosopher(int * a);

int main(void) {
  void * tab = tableinit(philosopher);
  sleep(60);  // Wait a while then exit
```

```
  printf("WE ARE DONE\n");}

void * philosopher(int * who) {
  /* For simplicity, all philosophers eat for the same amount */
  /* of time and think for a time that is simply related */
  /* to their position at the table. The parameter who identifies*/
  /* the philosopher: 0, 1, 2, .. */
  while (1){
    sleep((*who)+1);
    pickup((*who));
    sleep(1);
    putdown((*who));}}
```

[philtable.c]

```
/* philtable.c    */

#include  <sys/types.h>
#include  <pthread.h>

#define PHILNUM 5

typedef enum {thinking, hungry, eating} philstat;

typedef struct tablestruct {
  pthread_t t[PHILNUM];
  int self[PHILNUM];
  pthread_mutex_t mutex;
  pthread_cond_t condition[PHILNUM];
  philstat status[PHILNUM];
  } table;

table * tab;

void printstate(void){
  /* Prints out state of philosophers as, say, TEHHE, meaning */
  /* that philosopher 0 is thinking, philosophers 1 and 4 are eating, and*/
  /* philosophers 2 and 3 are hungry.*/
  static char stat[] = "THE";
  int i;
  for (i=0; istatus)[i]]);}
  printf("\n");
}

int test (int i) {
  if (
      ((tab->status)[i] == hungry) &&
      ((tab->status)[(i+1)% PHILNUM] != eating) &&
      ((tab->status)[(i-1+PHILNUM)% PHILNUM] != eating)) {
    (tab->status)[i] = eating;
    pthread_cond_signal(&((tab->condition)[i]));
    return 1;
  }
  return 0;
}

void pickup(int k) {
  pthread_mutex_lock(&(tab->mutex));
```

```
  (tab->status)[k] = hungry;
  printstate();
  if (!test(k)) {
    pthread_cond_wait(&((tab->condition)[k]), &(tab->mutex));}
  printstate();
  pthread_mutex_unlock(&(tab->mutex));
}

void putdown(int k) {
  pthread_mutex_lock(&(tab->mutex));
  (tab->status)[k] = thinking;
  printstate();
  test((k+1)%PHILNUM);
  test((k-1+PHILNUM)%PHILNUM);
  pthread_mutex_unlock(&(tab->mutex));
}

table * tableinit(void *(* philosopher)(void *)) {
  int i;

  tab = (table *) malloc (sizeof(table));
  if(pthread_mutex_init(&(tab->mutex), pthread_mutexattr_default) != 0) {
    perror("pthread_mutex_init");
    exit(1);}
  for (i=0; iself)[i] = i;
    (tab->status)[i] = thinking;
    if(pthread_cond_init(&((tab->condition)[i]), pthread_condattr_default)
       != 0) {
      perror("pthread_cond_init");
      exit(1);}
  }
  for (i=0; it)[i]),pthread_attr_default,
                   philosopher, &((tab->self)[i]))!= 0) {
      perror("pthread_create");
      exit(1);}
  }
  return tab;
}
```

Copying a file

The obvious way to copy a file "source.dat" to a file "target.dat" is to read a buffer from one and to write it to the other. An alternative way, taking advantage of overlap between read and write operations when using two buffers, is presented below in terms of threads and conditional critical regions.

```
/* cpfile.c  -- Copy a file overlapping read and write  */

#include  <sys/types.h>
#include  <sys/stat.h>
#include  <fcntl.h>
#include  <pthread.h>

#define BUFFERSIZE 8192

typedef enum {empty, full} bstat;
```

```
typedef struct bufferstruct {
  char b[BUFFERSIZE];
  int count;         // Number of characters in b
  bstat stat;        // State of b
  pthread_mutex_t mutex;
  pthread_cond_t condition;
  } buffer;

/* Operations on buffer */
void initbuffer (buffer * p);
void fillbuffer (buffer * p);
void unfillbuffer (buffer * p);

void filler(void); // The procedure executed by the reading thread

buffer buf[2]; // The two buffers used for concurrent reading, writing
int infile, outfile;

int main (int argc, char *argv[]){
  pthread_t t;
  int ubuffer = 0; // It is 0 or 1, indicating buffer to be emptied

  if (argc < 3) {
    printf("Usage: cpfile fromfile tofile\n");
    exit(0);}

  if ((infile = open(argv[1], O_RDONLY)) < 0) {
    printf("Cannot open %s\n", argv[1]); exit(0);}
  if ((outfile = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC,
                   S_IRUSR|S_IRGRP|S_IROTH)) < 0) {
    printf("Cannot open %s\n", argv[2]); exit(1);}
  initbuffer(&buf[0]);
  initbuffer(&buf[1]);

  if (pthread_create(&t, pthread_attr_default,
                  (void *)filler, NULL)!= 0) {
    perror("pthread_create");
    exit(1);}

  do {
    unfillbuffer(&(buf[ubuffer]));
    if ((buf[ubuffer].count) == 0) break;
    ubuffer = 1-ubuffer;} while (1);
  close(infile);
  close(outfile);
}


void filler(void){
  int status = 0;
  int fbuffer = 0;   // It is 0 or 1, undicating buffer to be filled

  do {
    fillbuffer(&(buf[fbuffer]));
    if ((buf[fbuffer].count) == 0) break;
    fbuffer = 1-fbuffer;} while (1);
  pthread_exit (&status);
}

void initbuffer (buffer * p) {
```

```
  if(pthread_mutex_init(&(p->mutex), pthread_mutexattr_default) != 0) {
    perror("pthread_mutex_init");
    exit(1);}
  if(pthread_cond_init(&((p->condition)), pthread_condattr_default)
     != 0) {
    perror("pthread_cond_init");
    exit(1);}
  p->count = 0;
  p->stat = empty;
}

/* It reads from the infile into buffer */
void fillbuffer (buffer * p) {
  pthread_mutex_lock(&(p->mutex));
  while ((p->stat)!=empty) {
    pthread_cond_wait(&((p->condition)), &(p->mutex));}
  (p->count) = read(infile, p->b, BUFFERSIZE);
  p->stat = full;
  pthread_cond_signal(&((p->condition)));
  pthread_mutex_unlock(&(p->mutex));
}

/* It writes into the outfile from buffer */
void unfillbuffer (buffer * p) {
  int nn;
  pthread_mutex_lock(&(p->mutex));
  while ((p->stat)!=full) {
    pthread_cond_wait(&((p->condition)), &(p->mutex));}
  p->stat = empty;
  pthread_cond_signal(&((p->condition)));
  pthread_mutex_unlock(&(p->mutex));
}
```

ingargiola.cis.temple.edu