Operating systems and concurrency (B08)

David Kendall

Northumbria University

Introduction

- Semaphores provide an unstructured synchronisation primitive
- Can lead to problems:
  - Accidental release
  - Deadlock
  - Starvation
  - ...more on this next week
- Can be difficult to detect and debug
- Monitors and condition variables offer one approach to a more structured synchronisation mechanism
- Support widely available, e.g. POSIX threads (Pthreads), Java, C#, etc.

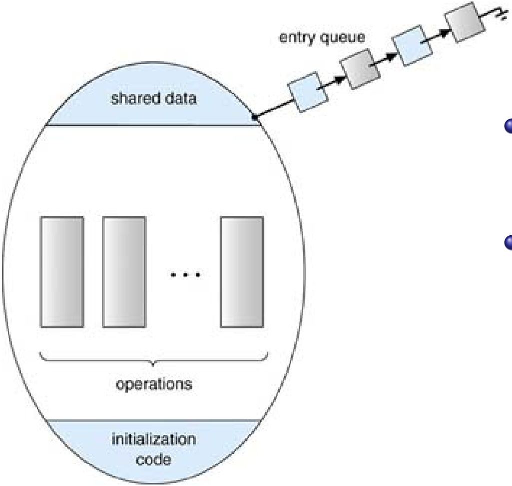
Monitor

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time – mutual exclusion

```
monitor monitor-name {
  // shared variable declarations

  procedure P1 (...) { ... }

                ...

  procedure Pn (...) { ... }

  Initialization code (...) { ... }
}
```

- Proposed independently by Per Brinch Hansen and Tony Hoare in 1973/74

Schematic view of a monitor



(from [SGG13, p.226])

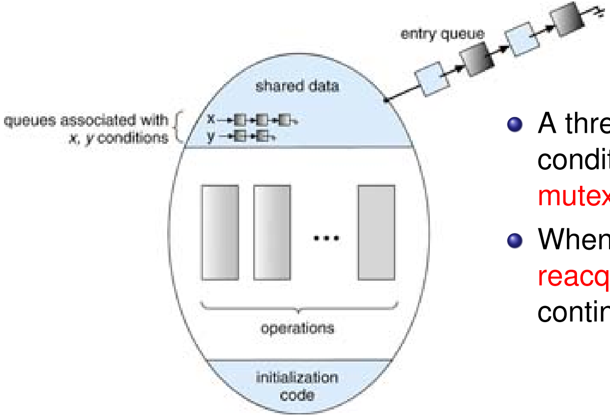
- Thread executing a monitor operation must hold the monitor mutex
- Other threads wanting to execute a monitor operation are queued on the mutex until the executing thread finishes and releases the mutex

Condition variables

- So a monitor provides mutual exclusion
- Condition variables introduced to allow signalling
- Three operations allowed on condition variable, cv:
  - wait(cv) – block until another thread calls signal or broadcast on cv
  - signal(cv) – wake up *one* thread waiting on cv
  - broadcast(cv) – wake up *all* threads waiting on cv
- In Pthreads the CV type is `pthread_cond_t`
  - Use `pthread_cond_init()` to initialise
  - `pthread_cond_wait(&cv, &mutex);`
  - `pthread_cond_signal(&cv);`
  - `pthread_cond_broadcast(&cv);`

(adapted from slides by Matt Welsh, Harvard University, 2009)

Monitor with condition variables



(from [SGG13, p.227])

- A thread that waits on a condition releases the monitor mutex
- When it is signalled, it must reacquire the mutex before continuing

Hoare vs Mesa Monitor Semantics

- The monitor `signal()` operation can have two different meanings:
- Hoare monitors (1974)
  - `signal(cv)` means to run the waiting thread immediately
  - Effectively "hands the lock" to the thread just signaled.
  - Causes the signalling thread to block
- Mesa monitors (Lampson and Redell, Xerox PARC, 1980)
  - `signal(cv)` puts waiting thread back onto the "ready queue" for the monitor
  - But, signaling thread keeps running.
  - Signaled thread doesn't get to run until it can acquire the lock.
    - This is what we almost always use, eg Pthreads, Java, C#, etc. because it's much easier for the OS to implement efficiently.
- What's the practical difference?
  - In Hoare-style semantics, the "condition" that triggered the `signal()` will always be true when the awoken thread runs
    - For example, that the buffer is now no longer empty
  - In Mesa-style semantics, awoken thread has to recheck the condition
    - Since another thread might have beaten it to the punch

Safe bounded buffer using a monitor

```
static pthread_mutex_t bufMutex;
static pthread_cond_t fullSlot;
static pthread_cond_t freeSlot;
static uint8_t nFull = 0;

void safeBufferInit(void) {
  pthread_mutex_init(&bufMutex, NULL);
  pthread_cond_init(&fullSlot, NULL);
  pthread_cond_init(&freeSlot, NULL);
}
```

- `bufMutex` is the monitor mutex
- `fullSlot` is the condition variable used to wait for a full slot
- `freeSlot` is the condition variable used to wait for an free slot
- `nFull` used to keep track of the number of messages in the buffer
- See https://github.com/DavidKendall/bb_with_monitor for the complete program.

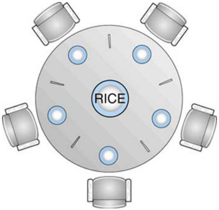
Safe bounded buffer using a monitor (ctd.)

```
void safeBufferPut(message_t const * const msg) {
  pthread_mutex_lock(&bufMutex);
  while (nFull == BUF_SIZE) {
    pthread_cond_wait(&freeSlot, &bufMutex);
  }
  putBuffer(msg);
  nFull += 1;
  pthread_cond_signal(&fullSlot);
  pthread_mutex_unlock(&bufMutex);
}
void safeBufferGet(message_t * const msg) {
  pthread_mutex_lock(&bufMutex);
  while (nFull == 0) {
    pthread_cond_wait(&fullSlot, &bufMutex);
  }
  getBuffer(msg);
  nFull -= 1;
  pthread_cond_signal(&freeSlot);
  pthread_mutex_unlock(&bufMutex);
}
```

Monitors – concurrency good practice

- Organise all shared state into one or more monitors
- Each monitor should have a single monitor mutex
- Every public method (function, procedure) should acquire the monitor mutex at the very beginning and release it at the very end of the method
- Use condition variables to synchronise (wait and signal for the state to satisfy some condition)
  - Notice that you have much more flexibility in expressing the condition to be waited for than when using a semaphore - you can use any boolean expression on the available state
  - Condition variables do not have an associated ‘counter’, just a queue of waiting threads
- Always wait for a condition variable in a loop (assume Mesa semantics)

Dining philosophers



- Another classic synchronisation problem
- Introduced here to illustrate how to build a monitor using Pthreads
- Philosophers think, get hungry, then eat, . . . repeatedly, . . . that's all
- To eat, a philosopher must have two chopsticks
- Spot the deadlock possibility . . . what about starvation? . . . oh how we laughed!
- See https://github.com/DavidKendall/dp_with_monitor

dpmonitor.c

```c
#include <assert.h>
#include <sys/types.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dpmonitor.h"

/************ Type declarations ***********************/

typedef enum {THINKING, HUNGRY, EATING} state_t;


/************ Local function prototypes **************/

static void eatIfOk(int i);
static int leftNghbr(int i);
static int rightNghbr(int i);
```

dpmonitor.c

```
/*********** Monitor variables ************************/

static pthread_mutex_t dpMutex;

static pthread_cond_t okToEat[N_PHIL];

static state_t state[N_PHIL];
```

dpmonitor.c

```
/*********** Monitor function definitions ***********/

void dpInit(void) {
  int rc;

  // Initialise the monitor mutex
  rc = pthread_mutex_init(&dpMutex, NULL);
  assert(rc == 0);

  // Initialise the state and the condition variables
  for (int i=0; i<N_PHIL; i+=1) {
    state[i] = THINKING;
    rc = pthread_cond_init(&okToEat[i], NULL);
    assert(rc == 0);
  }
}
```

dpmonitor.c

```c
void dpPickup(int i) {
  pthread_mutex_lock(&dpMutex);
  state[i] = HUNGRY;
  eatIfOk(i);
  while (state[i] != EATING) {
    pthread_cond_wait(&okToEat[i], &dpMutex);
  }
  pthread_mutex_unlock(&dpMutex);
}
```

dpmonitor.c

```
void dpPutdown(int i) {
  pthread_mutex_lock(&dpMutex);
  state[i] = THINKING;
  eatIfOk(rightNghbr(i));
  eatIfOk(leftNghbr(i));
  pthread_mutex_unlock(&dpMutex);
}
```

dpmonitor.c

```
/**************** Local function definitions ***********/

static void eatIfOk (int i) {
  if ((state[i] == HUNGRY) &&
      (state[rightNghbr(i)] != EATING) &&
      (state[leftNghbr(i)] != EATING)) {
    state[i] = EATING;
    pthread_cond_signal(&okToEat[i]);
  }
}

static int leftNghbr(int i) {
  return ((i+(N_PHIL-1)) % N_PHIL);
}

static int rightNghbr(int i) {
  return ((i+1) % N_PHIL);
}
```

Acknowledgements

- **[SGG10]** Silberschatz,A., Galvin, P., Gagne,G., *Operating System Concepts* (8th edition), Wiley, 2010
- Lawrence Livermore Pthreads tutorial
- Welsh, M., *Semaphores, Condition Variables and Monitors*, Lecture slides, Harvard University, 2009 (local copy)