Real-Time Programming

Lecture 5

Farhang Nemati

Spring 2016

Repetition

- volatile …
- Semaphores
- Mutexes

Semaphores/Mutexes; Exercise

- Improve the Producer-Consumer so that:
  - There is a maximum size for the buffer:
    `int buffer[10];`
  - The producer waits (is blocked) if buffer is full
  - The consumer waits (is blocked) if the buffer is empty

Semaphores/Mutexes; Solution

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

const int MAX_BUF_SIZE = 10;
int buffer[MAX_BUF_SIZE];
void start()
{
   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
   sem_t bufFullSem, bufEmptySem;
   pthread_t prodThrd, consThrd;
   sem_init(&bufFullSem, 0, MAX_BUF_SIZE);
   sem_init(&bufEmptySem, 0, 0);
   pthread_create(&prodThrd, NULL, producer, NULL);
   pthread_create(&consThrd, NULL, consumer, NULL);
}
```

Semaphores/Mutexes; Solution

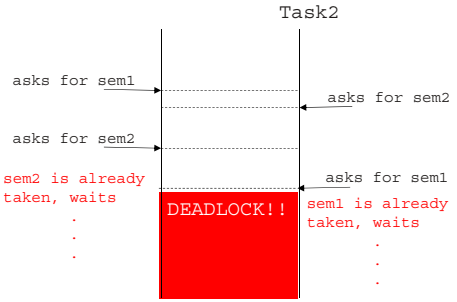
```
void *producer(void *arg)
{
    while(1)
    {
       x = calculate();
       sem_wait(&bufFullSem);
       pthread_mutex_lock(&mutex);
       buffer[index++] = x;
       pthread_mutex_unlock(& mutex);
       sem_post(&bufEmptySem);
    }
}
```

```
void *consumer(void *arg)
{
    while(1)
    {
       sem_wait(&bufEmptySem);
       pthread_mutex_lock(&mutex);
       y = buffer[--index];
       pthread_mutex_unlock(& mutex);
       use(y);
       sem_post(&bufFullSem);
    }
}
```

Problems with Semaphores and Mutexes

- Deadlock

```
//Task1                  //Task2
...                      ...
sem_wait(&sem1);         sem_wait(&sem2);
...                      ...
sem_wait(&sem2);         sem_wait(&sem1);
...                      ...
sem_post(&sem2);         sem_post(&sem1);
...                      ...
sem_post(&sem1);         sem_post(&sem2);
```



- Task Starvation (Indefinite Postponement)
- Error prone: One mistake can take the whole system down.

Condition Variables

- In combination with a mutex
- Two Operations
  - Wait: Always waits (suspends) for a condition. Releases the associated mutex while suspending on the condition.
  - Signal: wakes up a waiting task. The waiting task is resumed and checks the condition again. Has no effect if no task is waiting for the condition.

POSIX Condition Variables

```
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;

pthread_cond_wait(pthread_cond_t*,pthread_mutex_t *);

pthread_cond_signal(pthread_cond_t*,pthread_mutex_t *);
```

Condition Variables

```
void start()
{
   pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;
   pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

   pthread_t prodId, oddConsId, evenConsId;
   pthread_create(&prodId, NULL, producer, NULL);
   pthread_create(&oddConsId, NULL, oddConsumer, NULL);
   pthread_create(&evenConsId, NULL, evenConsumer, NULL);
}
```

Condition Variables

```
void *producer(void *arg)
{
    while(1)
    {
        x = calculate();
        pthread_mutex_lock(&mutex);
        buffer[index++] = x;
        pthread_cond_signal(&cond, &mutex);
        pthread_mutex_unlock(&mutex);
    }
}
```

Condition Variables

```
void *evenConsumer(void *arg)
{
    while(1)
    {
       pthread_mutex_lock(&mutex);
       while(index == 0 || buffer[index]%2 != 0)
       {
           pthread_cond_wait(&cond, &mutex);
       }
       y = buffer[--index];
       pthread_mutex_unlock(&mutex);

       //Another consumer might be waiting ...
       pthread_cond_signal(&cond, &mutex);

       addEven(y);
    }
}
```

```
void *oddConsumer(void *arg)
{
    while(1)
    {
       pthread_mutex_lock(&mutex);
       while(index == 0 || buffer[index]%2 == 0)
       {
           pthread_cond_wait(&cond, &mutex);
       }
       y = buffer[--index];
       pthread_mutex_unlock(&mutex);

       //Another consumer might be waiting ...
       pthread_cond_signal(&cond, &mutex);

       addOdd(y);
    }
}
```

Monitors

- All critical sections and operations needed for managing synchronization are encapsulated in modules that are called Monitors.
- Critical sections and variables to be accessed under mutual exclusion are hidden from the caller.

Monitor Example

```
const int MAX_SIZE = 100;
typedef struct Monitor
{
    pthread_mutex_t mutex;
    pthread_cond_t bufFullSem;
    pthread_cond_t bufEmptySem;
    int buffer[MAX_SIZE];
    int index = 0;
} MyMonitor;

void initMonitor(myMonitor* monitor)
{
     monitor->mutex = PTHREAD_MUTEX_INITIALIZER;
     monitor->bufFullSem = PTHREAD_COND_INITIALIZER;
     monitor->bufEmptySem = PTHREAD_COND_INITIALIZER;
}
```

Monitor Example

```
void takeValue(myMonitor* monitor, int* x)
{
    pthread_mutex_lock(&(monitor->mutex));
    while(monitor->index == 0)
    {
      pthread_cond_wait(&(monitor-> bufEmptySem), &(monitor->mutex));
    }
    *x = monitor->buffer[--monitor->index];
    pthread_mutex_unlock(&(monitor->mutex));
    pthread_cond_signal(&(monitor-> bufFullSem), &(monitor->mutex));
}

void addValue(MyMonitor* monitor, int x)
{
    pthread_mutex_lock(&(monitor->mutex));
    while(monitor->index == MAX_SIZE)
    {
        pthread_cond_wait(&(monitor->bufFullSem), &(monitor->mutex));
    }
    monitor->buffer[monitor->index++] = x;
    pthread_mutex_unlock(&(monitor->mutex));
    pthread_cond_signal(&(monitor->bufEmptySem), &(monitor->mutex));
}
```

Barriers

- A synchronization point for a group of tasks
- Every task in the group stops at the assigned barrier until all other tasks from the group are reached the barrier

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
       const pthread_barrierattr_t *restrict attr, unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```