Real-Time Programming

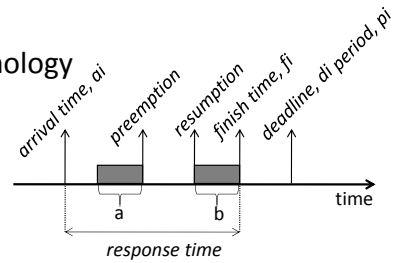
Lecture 4

Farhang Nemati

Spring 2016

Repetition

- Task Terminology



execution time = a + b

Worst Case Execution Time (e_i) = max(execution time)

- Preemptive, Non-preemptive Tasks
- Periodic, Aperiodic, Sporadic Tasks

Repetition

- POSIX Clocks and Timing Facilities
  - `int clock_getres(clockid_t, struct timespec *)`
  - `int clock_gettime(clockid_t, struct timespec *)`
  - `int clock_settime(clockid_t, const struct timespec *)`
  - `CLOCK_REALTIME`
- Task States in RTOS
  - Ready, Running, Waiting, Idle
- Extract Timing Restrictions
  - Application requirements, Physical rules
- Task Synchronization and Communication
  - Shared Variable Based,
    - Atomic Operation, Mutual Exclusion, Condition Synchronization, Busy Waiting, …
  - Message Passing Based

Busy Waiting

```
while (flag != 1)
{
    //Do nothing ...
}
```

- Livelock: The tasks may stuck in a loop for condition checking and never exit the loop
- Inefficient: wasting processor with not useful work
- Developing and testing is difficult

Exercise; Two Threads

- Create two threads. The first thread is supposed to increment a global integer and the second thread will put the odd values of the counter in a buffer. The program then checks if the buffer only contains odd numbers. Does the program have the results as expected?
- See the attached code (*twothread.c*)

Semaphores

- Used for synchronization
- Wraps a non-negative integer value
  - The value can only be accessed through 2 function calls
- It may have an internal queue in which tasks waiting for the semaphore are put

Semaphores

- Three operation are possible:
  - To create a semaphore with a value
  - To take/wait for the semaphore:
    - If the value is 0 the caller task suspends
    - If the value is not 0, the value is decremented by 1 and the caller proceeds
  - To give/signal the semaphore
    - The value is incremented by 1
    - Signals the tasks waiting (suspended) on the semaphore

Semaphores

- No need for busy waiting
- Binary Semaphores
  - Its value has only two different values; 0, 1
  - Used for cases with two cases, e.g., on/off, busy/free
- General (Counting) Semaphores
  - Its value can have any non-negative number
  - Suitable for handling resource sharing

Mutexes

- Similar to semaphore with a binary value
- Used for protecting mutual exclusion critical sections
- Suits for handling mutually exclusive resources shared among multiple tasks
- The task that takes a mutex, owns it

POSIX Semaphores

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

- Initialize/open a semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned value); //unnamed
sem_t *sem_open(const char *name, int oflag, ...); //named
```

- Take/wait for a semaphore

```
int sem_wait(sem_t *sem);//the caller task(thread) suspends if the semaphore is occupied
int sem_trywait(sem_t *sem); //the caller task(thread) continues even if the semaphore is occupied
```

- Give/Signal a semaphore

```
int sem_post(sem_t *sem);
```

- Delete a semaphore

```
int sem_destroy(sem_t *sem); //unnamed
int sem_unlink(const char *name); //named
```

POSIX Mutexes

- Like POSIX threads the attributes of a mutex are passed as an attributes object (pthread_mutexattr_t)
- There are functions to get/set each attribute of the attributes object

POSIX Mutexes

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

- Initialize a mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Or

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

- Destroy a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

POSIX Mutexes

- Lock a mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);//the caller task suspends if the mutex is locked

int pthread_mutex_trylock(pthread_mutex_t *mutex);//the caller task continues even if the mutex is locked
```

- Unlock a mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exercise; Two Threads with Mutex

- Remove the inconsistency in the results
- See the attached code (*twothread_sync.c*)

Semaphores/Mutexes

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

//Task1                              //Task2
while(1)                             while(1)
{                                    {
   pthread_mutex_lock(&mtx);            pthread_mutex_lock(&mtx);
   x = x + 1;                           x = x - 1;
   pthread_mutex_unlock(&mtx);          pthread_mutex_unlock(&mtx);

   y = y + 1;                           z = z + 1;
}                                    }
```

Semaphores/Mutexes

- How many semaphores (or mutexes) are needed?

```
//Task1                //Task2                //Task3
while(1)               while(1)               while(1)
{                      {                      {
   x = x + 1;             x = x - 1;             x = x * 2;
   y = y + 1;             z = z + 1;             z = 0;
}                      }                      }
```

Semaphores/Mutexes

```
//Task1
while(1)
{
   sem_wait(&sem1);
   x = x + 1;
   sem_post(&sem1);

   y = y + 1;
}
```

```
//Task2
while(1)
{
   sem_wait(&sem1);
   x = x - 1;
   sem_post(&sem1);

   sem_wait(&sem2);
   z = z + 1;
   sem_post(&sem2);
}
```

```
//Task3
while(1)
{
    sem_wait(&sem1);
    x = x * 2;
    semGive(&sem1);

    sem_wait(&sem2);
    z = 0;
    sem_post(&sem2);
}
```

- Make critical sections as short as possible

Semaphores/Mutexes

- Producer-Consumer reading from/writing to a buffer

```
//Producer
while(1)
{
   x = calculate();

   pthread_mutex_lock(&mtx);
   buffer[index++] = x;
   pthread_mutex_unlock(&mtx);
}
```

```
//Consumer
while(1)
{
   pthread_mutex_lock(&mtx);
   y = buffer[--index];
   pthread_mutex_unlock(&mtx);

   use(y);
}
```

- What happens if Producer writes to the buffer when it is full? Or Consumer reads from the buffer when it is empty?

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