CS 241 Section Week #5
(2/25/10)

Topics This Section

- MP3 Review
- Synchronization
- Problems
- Deadlocks
- MP4 Forward

MP3

MP3 Review

- 3 scheduling functions:
  - new_job()
  - job_finished()
  - quantum_expired()
- Job queue
  - Distinguish the running job and all other jobs in the queue
  - How to maintain the queue?
    - Priority queue based implementation: the head is the next job to be run
    - Normal queue: traverse the queue to find the right job to be run next

MP3 Review

- Statistic functions
  - turnaround_time = finishing_time – arrival_time
  - wait_time = turnaround_time – running_time
  - response_time = first_run_time – arrival_time
- It is tricky to keep track of the first_run_time
  - Several events may happen at the same time unit t: new_job(), job_finish(), etc
  - new_job() returns job m
  - job_finish() returns job n
  - job n is the next job to be run by the CPU
  - Assuming both m and n are never run by CPU before, only job n's first_run_time should be updated to time t

MP3 Review

- C header files
  - A header file (.h file) is a file containing C declarations (functions or variables) and macro definitions to be shared between several source files
  - Two variants:
    - #include <file>
    - #include "file"
  - You may use include guards to avoid illegal multiple definitions of the same variable or the same function

```
#ifndef SOME_GUARD
#define SOME_GUARD
int global_variable;
#endif
```

Semaphores

Example (machex1.c)

```
int N = 1000000;
int x = 0;

int main(int argc, char** argv)
{
    pthread_t threadCountUp, threadCountDown;

    pthread_create(&threadCountUp, NULL,countUp,NULL);
    pthread_create(&threadCountDown,NULL, countDown,
    NULL);
    pthread_join(threadCountUp, NULL);
    pthread_join(threadCountDown, NULL);
    printf("%d\n", x);
}
```

Example

```
void* countUp()
{
    int i;

    for (i = 0; i < N; i++)
    {
     int c = x;
     c++;
     x = c;
    }
}
```

```
void* countDown()
{
     int i;

     for (i = 0; i < N; i++)
     {
          int c = x;
          c--;
          x = c;
     }
}
```

Semaphores

- Thread1 did ‘x++’ N times.
- Thread2 did ‘x--’ N times.
- Ideal result: ‘x’ is at its initial value.
- Please try to compile machex1.c and run it with different N values: N= 1000, N = 1000000, etc…
- Actual result?

Semaphores

- To fix this:
  - A thread must read, update, and write ‘x’ back to memory without any other threads interacting with ‘x’.
  - This concept is an atomic operation.

Semaphores

- Conceptually, we would want an ‘atomic’ scope:
  - ```
    void* countUp() {
        atomic {
            int c = x;
            c++;
            x = c;
        }       // But this doesn't exist…
    }
    ```
    ```

Semaphores

- Semaphores provide a locking mechanism to give us atomicity.
  - void* countUp() {
    sem_wait(&sema);
    int c = x;
    c++;
    x = c;
    sem_post(&sema);
    }

Semaphores

- Semaphores provide a locking mechanism to give us atomicity.
  - void* countUp() {
    sem_wait(&sema); LOCKS
    int c = x;
    c++;
    x = c;
    sem_post(&sema); UNLOCKS
    }

Semaphores

- To use a semaphore, you have to define it. Three steps to defining a semaphore:
- 1. Include the header file:
  - #include <semaphore.h>

Semaphores

- To use a semaphore, you have to define it. Three steps to defining a semaphore:
- 2. Declare the semaphore:
  - sem_t sema;
    (Declare this in a global scope.)

Semaphores

- To use a semaphore, you have to define it. Three steps to defining a semaphore:

- 3. Initialize the semaphore:
  - sem_init(&sema, 0, 1);

Semaphores

- sem_init(&sema, 0, 1);
  - &sema: Your declared sem_t.
  - 0 : 0 := Thread Sync
  - 1 : Total of one thread inside a 'locked' section of code.

Semaphores

- Three steps to starting them:
  - Include: #include <semaphore.h>
  - Define: sem_t sema;
  - Init: sem_init(&sema, 0, 1);

Semaphores

- Two functions to use them:
  - Acquiring the lock:
    sem_wait(&sema);
  - Releasing the lock:
    sem_post(&sema);

Semaphores

- Example Revisited:

sem_wait()
read x
x++
context sw → sema_wait()
write x ← thread blocked
sem_post()
unlocked → /*…*/

Mutexes

- Mutexes are binary semaphores
- Simple and efficient
- Use of a mutex
  - pthread_mutex_init(): unlike semaphores, no initial value is needed
  - pthread_mutex_lock()
  - pthread_mutex_trylock()
  - pthread_mutex_unlock()
  - pthread_mutex_destroy()
- We focus on mutexes in MP4

Problems

Problem 1 - Use semaphores to ensure order

- machex2.c creates two threads to print out "Hello World".
- Use semaphores to ensure that "World\nHello" is never printed instead of "Hello World".

```
void *hello_thread() {
   sleep(2);
   fprintf(stderr, "Hello ");
}

void *world_thread() {
   sleep(1);
   fprintf(stderr, "World!\n");
}
```

```
int main() {
   pthread_t hello, world;

   pthread_create(&hello, NULL, hello_thread, NULL);
   pthread_create(&world, NULL, world_thread, NULL);

   pthread_join(hello, NULL);
   pthread_join(world, NULL);

   return 0;
}
```

Problem 1 - Use semaphores to ensure order

```
void *hello_thread() {
    sleep(2);
    fprintf(stderr, "Hello ");
    sem_post(&sem);
}

void *world_thread() {
    sleep(1);
    sem_wait(&sem);
    fprintf(stderr, "World!\n");
    sem_post(&sem);
}
```

```
sem_t sem;

int main(){
    pthread_t hello, world;

    sem_init(&sem, 0, 0);

    pthread_create(&hello, NULL, hello_thread,
NULL);
    pthread_create(&world, NULL, world_thread,
NULL);

    pthread_join(hello, NULL);
    pthread_join(world, NULL);

    return 0;
}
```

Problem 2 – Two semaphores

- machex3.c creates two threads.
  - Both want access to two semaphores.
- If you run this program, the program appears to stop running after a bit. Why?

```
sem_t sem1, sem2;
int main() {
    pthread_t t1, t2;

    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 1);

    pthread_create(&t1, NULL, p1, NULL);
    pthread_create(&t2, NULL, p2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

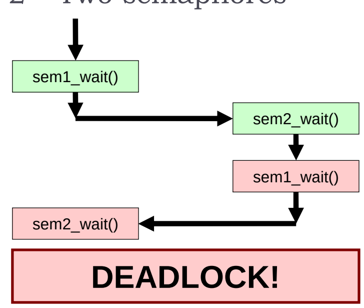
    return 0;
}
```

Problem 2 – Two semaphores

```
void *p1() {
   while (1) {
      printf("p1: sem_wait(&sem1);\n");
       sem_wait(&sem1);
       printf("p1: sem_wait(&sem2);\n");
       sem_wait(&sem2);
       printf("p1: locked\n");
       printf("p1: sem_post(&sem2);\n");
       sem_post(&sem2);
       printf("p1: sem_post(&sem1);\n");
       sem_post(&sem1);
       printf("p1: unlocked\n");
   }
   return NULL;
}
```

```
void *p2(){
   while (1) {
       printf("p2: sem_wait(&sem2);\n");
       sem_wait(&sem2);
       printf("p2: sem_wait(&sem1);\n");
       sem_wait(&sem1);
       printf("p2: locked\n");
       printf("p2:sem_post(&sem1);\n");
       sem_post(&sem1);
       printf("p2: sem_post(&sem2);\n");
       sem_post(&sem2);
       printf("p2: unlocked\n");
       }
       return NULL;
}
```

Problem 2 – Two semaphores



Requirements for Deadlock

- Mutual exclusion
  - Processes claim **exclusive** control of the resources they require
- Hold-and-wait (a.k.a. wait-for) condition
  - Processes hold resources already allocated to them while waiting for additional resources
- No preemption condition
  - Resources cannot be removed from the processes holding them until used to completion
- Circular wait condition: ***deadlock has occurred***
  - A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain

Dealing with Deadlocks

- Prevention
  - Break one of the four deadlock conditions
- Avoidance
  - Impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- Detection
  - determine if deadlock has occurred, and which processes and resources are involved.

MP4

MP4 Overview

- This is your first long MP. You have two weeks to complete it.
- You need to implement two parts:
  - The “deadlock resilient mutex” library: libdrm
  - The library for cycle detection and cycle-related functions: libwfg
- A compiled libwfg library is provided for you to implement the first part of the MP
- libwfg is not thread-safe.Therefore, you will need to have a lock to control access to calls to libwfg to ensure two processes do not make a call at the same time to libwfg.
- After completing the first part, you should write your own libwfg libary

Part 1: Deadlock Resilient Mutex

- Deadlock prevention
  - Enforce a global ordering on all locks
  - Locks should be acquired in descending order
- Deadlock avoidance
  - No cycle exist in a wait-for graph
- Deadlock detection
  - Periodically incur the cycle detection algorithm

Part 2: The library for cycle detection and cycle-related functions

- You are to implement the wait-for graph (a resource allocation graph in fact)
  - **wfg_init()**
  - **wfg_add_wait_edge(): a thread request a resource**
  - **wfg_add_hold_edge(): a resource is acquired by a thread**
  - **wfg_remove_edge()**
- You are to implement the cycle detection algorithm
- The given test cases are far from complete. You should derive your own test cases.