CS 241 Section Week #5
(10/01/09)

Topics This Section

- MP3 Review
- MP4 Forward
- Semaphores
- Problems

MP3

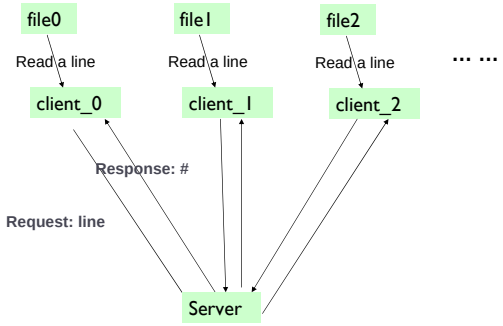
MP3 Review

- 3 scheduling functions:
  - new_job()
  - job_finished()
  - quantum_expired()
- Job queue
- Statistic functions
  - initial_wait_time(), wait_time(), response_time()
- Note about memory
  - valgrind

MP4

MP4 Forward

- The scenario:



MP4 Forward

- Communicate using shared memories
  - queue
  - client_result[client_id]
- Deal with three semaphores
  - queue_semaphore
    - To protect access to the queue
  - server_notify_semaphore
    - Client notifies the server when data is added to the queue
  - client_notify_semaphore
    - Array of semaphores
    - Client ID is used to access the semaphores
    - Server notify individual client a response is ready

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

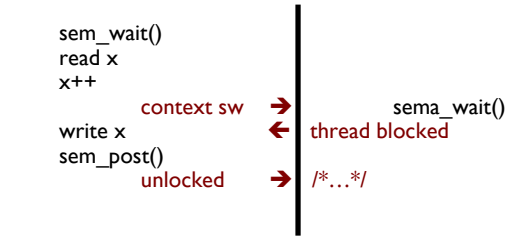
- Three steps to starting them:
  - Include: #include <semaphore.h>
  - Define: sem_t sema;
  - Init: sem_init(&sema, 0, 1);

Semaphores

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

Semaphores

- Example Revisited:



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, NUL
    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