CSE 421/521 - Operating Systems
Fall 2014 Recitations

Recitation – III

Concurrent Programming

Prof. Tevfik Kosar

Presented by Sharath Chandrashekhara

University at Buffalo

September 16-18, 2014

Threads

- In certain cases, a single application may need to run several tasks at the same time
  - Creating a new process for each task is time consuming
  - Use a single process with multiple threads
    - faster
    - less overhead for creation, switching, and termination
    - share the same address space

Thread Creation

- **pthread_create**

```
// creates a new thread executing start_routine
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void
  *arg);
```

- **pthread_join**

```
// suspends execution of the calling thread until the target
// thread terminates
int pthread_join(pthread_t thread, void **value_ptr);
```

Thread Example

```
main()
{
pthread_t thread1, thread2;  /* thread variables */

pthread_create(&thread1, NULL, (void *) &print_message_function,(void*)"hello ");
pthread_create(&thread2, NULL, (void *) &print_message_function,(void*)"world!");

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

printf("\n");
exit(0);
}
```

Why use pthread_join?

To force main block to wait for both threads to terminate, before it exits.
If main block exits, both threads exit, even if the threads have not finished their work.

Thread Example *(cont.)*

```
void print_message_function ( void *ptr )
{
    char *cp = (char*)ptr;
    int i;
    for (i=0;i<3;i++){
        printf("%s \n", cp);
        fflush(stdout);
        sleep(1);
    }

    pthread_exit(0); /* exit */
}
```

Example: Interthread Cooperation

```
void* print_count ( void *ptr );
void* increment_count ( void *ptr );

int NUM=5;
int counter =0;

int main()
{
   pthread_t thread1, thread2;

   pthread_create (&thread1, NULL, increment_count, NULL);
   pthread_create (&thread2, NULL,  print_count, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   exit(0);
}
```

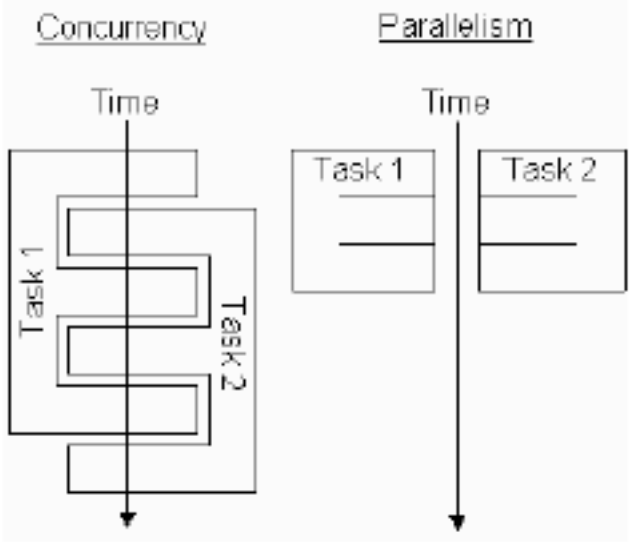
Interthread Cooperation *(cont.)*

```
void* print_count ( void *ptr )
{
    int i;
    for (i=0;i<NUM;i++){
        printf("counter = %d \n", counter);
        //sleep(1);
    }
    pthread_exit(0);
}

void* increment_count ( void *ptr )
{
    int i;
    for (i=0;i<NUM;i++){
        counter++;
        //sleep(1);
    }
    pthread_exit(0);
}
```

Concurrency Issues

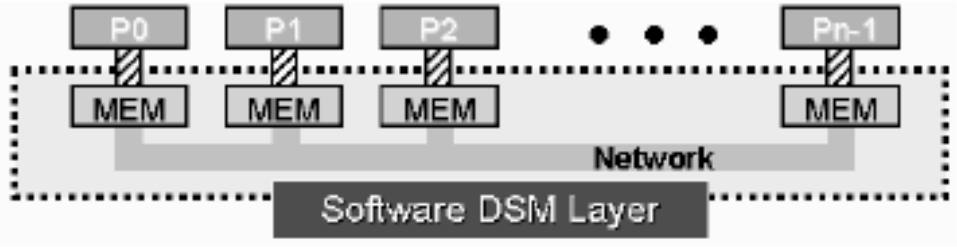
P1:	`X:=1`
P2:	`X:=0; X:=X+1`



- If programs are independent, the results are the same (X=1)
- If programs are executed concurrently and one program is `X:=1`, are results of P1 and P2 different
- **"interleaving" makes it difficult to deal with global properties from the local analysis!**
- assumption: access to the memory is atomic

Concurrency Issues

- Shared variables are an effective way to communicate between processes
- X:=X+1 is implemented as 3 different instructions
  - load the value of X to the register
  - increment the register
  - store the value of register to X
- Two processes updating same variable concurrently causes erroneous results
- Correctivity of the program needs that this updating will be indivisible (or atomic)
- Reading a variable can also be a critical section
  - e.g. reading four bytes that are not volatile




```
LD      AX, CARS
INC     AX
LD      CARS, AX
```

POSIX Threads: MUTEX

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
                                                          *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- a new data type named pthread_mutex_t is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the attribute of a mutex can be controlled by using the pthread_mutex_init() function
- the lock/unlock functions work in tandem

MUTEX Example

```
#include <pthread.h>
pthread_mutex_t my_mutex;     // should be of global scope
int main()
{
        int tmp;
        // initialize the mutex
        tmp = pthread_mutex_init( &my_mutex, NULL );
        // create threads
        ...
        pthread_mutex_lock( &my_mutex );
            do_something_private();
        pthread_mutex_unlock( &my_mutex );
        ...
        return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

POSIX: Semaphores

- creating a semaphore:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

initializes a semaphore object pointed to by sem

pshared is a sharing option; a value of 0 means the semaphore is local to the calling process

gives an initial value value to the semaphore

- terminating a semaphore:

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

frees the resources allocated to the semaphore sem

usually called after pthread_join()

an error will occur if a semaphore is destroyed for which a thread is waiting

POSIX: Semaphores *(cont.)*

- semaphore control:

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

sem_post atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Semaphore: Example

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
sem_t semaphore;        // also a global variable just like mutexes
int main()
{
        int tmp;
        // initialize the semaphore
        tmp = sem_init( &semaphore, 0, 0 );
        // create threads
        pthread_create( &thread[i], NULL, thread_function, NULL );
        ...
        while ( still_has_something_to_do() )
        {
                sem_post( &semaphore );
                }
        ...
        pthread_join( thread[i], NULL );
        sem_destroy( &semaphore );
        return 0;
}
```

Semaphore: Example *(cont.)*

```
void *thread_function( void *arg )
{
        sem_wait( &semaphore );
        perform_task_when_sem_open();
        ...
        pthread_exit( NULL );
}
```

Exercises

Threads (True or False Questions):

- A thread cannot see the variables on another thread's stack.
- ***False*** *-- they can since they share memory*

- In a non-preemptive thread system, you do not have to worry about race conditions.
- ***False*** *-- as threads block and unblock, they may do so in unspecified orders, so you can still have race race conditions.*

- A thread may only call **pthread_join()** on threads which it has created with **pthread_create()**
- ***False*** **--** *Any thread can join with any other*

- With mutexes, you may have a thread execute instructions atomically with respect to other threads that lock the mutex.
- ***True*** *-- That's most often how mutexes are used.*

Exercises

Threads (True or False Questions):

- pthread_create() always returns to the caller
- True.

- pthread_mutex_lock() never returns
- False -- It may block, but it when it unblocks, it will return.

- pthread_exit() returns if there is no error
- False -- never returns.