Sistemas Operativos: Concurrency

Pedro F. Souto (pfs@fe.up.pt)

March 30, 2016

Roadmap

Threads

Review

- A multi-threaded program has several execution points
- Threads of the same process:
  - Share the same address space

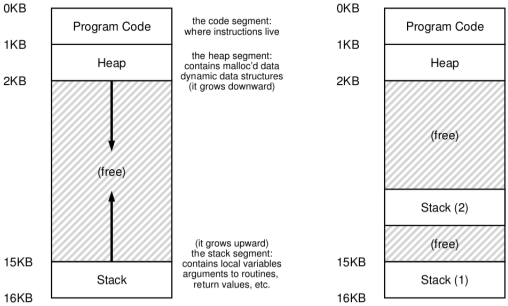

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

- However each thread has its own:
  - Stack (Stack Pointer/Base Pointer)
  - Register set, including
    - Program Counter/Instruction Pointer
  - State (Ready, Running, Waiting)

Thread Creation

```
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Interleavings (1/3)

- Thread "A" runs when "main" thread blocks

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Interleavings (2/3)

- Thread "A" runs as soon as it is created

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Interleavings (3/3)

- Thread "B" runs before thread "A"

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

- Many more interleavings are possible:
  - They are determined by the scheduler decisions

Data Sharing in Multi-Threaded Programs (1/2)

```
int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg)
{
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n",
                letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Data Sharing in Multi-Threaded Programs (2/2)

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n"
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n",
                counter, (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
                counter, max*2);
    return 0;
}
```

Pthread_create() is not pthread_create()

```
void
Pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void*), void *arg)
{
    int rc = pthread_create(thread, attr,
                            start_routine, arg);
    assert(rc == 0);
}
```

- This is a pattern used by Richard Stevens in his books
- It is useful for illustrating simple programs
  - There is no need for explicitly handling failure of the system call

Data Sharing: What is going on

- Let's run this for different values of `max`:
  - 1.000.0000
  - 10.000.0000
  - 100.000.0000

Data Sharing: What is going on?

- `objdump -d t1`

```
4009b4: mov    0x2008d2(%rip),%eax # 60128c <counter>
4009ba: add    $0x1,%eax
4009bd: mov    %eax,0x2008c9(%rip) # 60128c <counter>
```

- Or, from the book:

```
100: mov 0x8049a1c, %eax
105: add $0x1, %eax
108: mov %eax, 0x8049a1c
```

Data Sharing: a race condition

OS	Thread 1	Thread 2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

- Actually, this needs not happen that often
  - Usually a thread executes several loop iterations, not one only
- The issue is that the increment is not done **atomically**, i.e. indivisibly
  - If the x86 had an instruction that increments a value in memory atomically:

```
100: addl 0x8049a1c,0x01
```

Data Sharing: critical sections and mutual exclusion

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

- If the execution of a code segment may lead to a race condition, then we say that that segment is a **critical section**
- A simple way to ensure correct execution is to ensure that critical sections are executed in **mutual exclusion**
  - This can be done with the help of some **synchronization primitives**

Lock

- A **lock** is a synchronization variable that is used to ensure mutual exclusion in the execution of critical sections that may **interfere** with one another
- Locks support two operations (primitives):

  `lock` which **locks**/**acquires** a lock

  - Upon return, the `lock` is **locked/aquired/held** by the calling thread
    - Depending on the implementation, the calling thread may block, i.e. move to the `WAIT` state, if the lock has been locked already

  `unlock()` which **unlocks**/textbfreleases a lock

  - Invoking `unlock` on a **locked/aquired/held** lock, allows another thread to return from the `lock` primitive
- So the **protocol** used to prevent race conditions with locks is:

```
lock_t mutex;    // some globally-allocated lock
...
lock(mutex);
...              // critical section
unlock(mutex)
```

Locks (Mutexes) em *libpthread*

- A mutex is a variable whose type is **`pthread_mutex_t`**

```
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- List of functions that operate on mutexes:

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

- `pthread_mutex_trylock()` tries to *lock*, i.e. it always returns immediately, even if the mutex is already locked.
  - The return value indicates whether the lock is held by the calling thread or by a different thread.
- A *mutex* must be initialized before use

Eliminating race-conditions with locks

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

int max;
volatile int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *mythread(void *arg)
{
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
          Pthread_mutex_lock(&lock);
          counter = counter + 1; // shared: only one
          Pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```