5118020-03 Operating System

Multithreading

OSTEP Chapters 26 and 28

Shin Hong

Thread

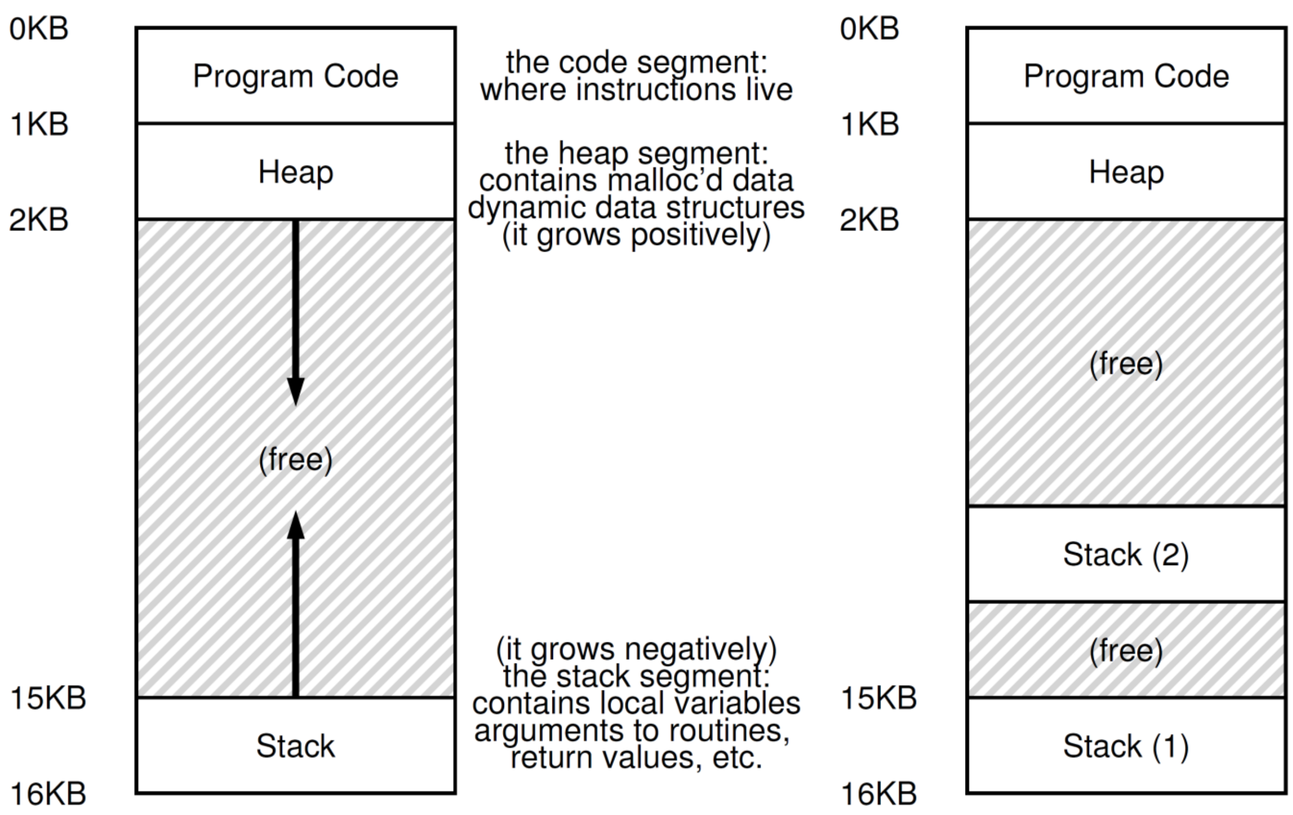
- Thread is an abstraction of a computation flow (or CPU)
  - thread vs. process
    - a process comprises abstractions of CPU, memory and other system resources, while a thread abstracts only CPU
    - a process may contain multiple threads that share the memory space and all other resources of the process

- Programs are asked to be *multithreaded* to concurrently run multiple collaborating logics
  - for interactive communication with multiple entities (e.g., GUI, network communication)
  - to exploit multi-core processors

Multithreading Mechanism

- Context switching
  - kernel threads
  - user-level threads

- Address space



Example


```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

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");
    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: end\n");
    return 0;
}
```

Non-deterministic Scheduling

- Concurrent threads can run in different order depending on how the scheduler decide to run them

- Accesses on a shared data structure by concurrent threads must be synchronized, or they may result erroneous states
  - without synchronization, there is no guarantee that instructions of a thread are executed consecutively

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", 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 with both (counter = %d)\n",
            counter);
    return 0;
}
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Race Condition

- A multithreaded program has a **race condition** if the result of a program execution with the same input depends on how thread scheduling went (i.e., nondeterministic)
  - unintended race condition is considered harmful

- Example

```
1  int balance;
   //must be non-negative
2  mutex m ;

3  withdraw (int x){
4    if(balance >= x){
5      balance=balance-x ;
6    }
7  }
```

Initially, `balance`:10

-t1: `withdraw(10)` –	-t2: `withdraw(5)` –
4 `if(balance>=10)` 5 `t = balance`	
	3 `if(balance>=5)`
5 `balance=t-10`	
	5 `t = balance` 5 `balance=t-5`

Enforcing Atomicity

- A **critical section** is a code block that must be executed by at most one thread at a time to guarantee its sequential, non-interfered execution for updating shared data structures

- A critical section must be executed **atomically** such that no interference happens in a middle of a critical section execution (i.e., mutual execution)
    - disabling timer interrupt
    - using synchronization primitive

Lock

- A programmer needs a mechanism to enforce a series of instructions to be executed atomically despite non-deterministic scheduling interrupts

- A lock is a special variable that only one thread can hold at a time
  - also called as mutex (mutual exclusion)
  - status
    - available (or unlocked, or free)
    - acquired(or locked, or held): owner
  - operations
    - lock (or acquisition)
    - unlock (or release)

Lock Implementations

- Why naïve spin-wait does not work?

  - checking and setting of a flag are not guaranteed to be atomic

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Lock Implementations: Disabling Interrupt

- Approach 1. disabling interrupt
  - turning off timer interrupt before entering a critical section

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

  - limitations
    - dangerous
    - reduces concurrency
    - does not work on multiprocessor architectures
    - inefficient

Lock Implementations: Software Lock

- There exist software lock algorithms (e.g., Dekker's algorithm) that consist of atomic load and store instructions such that two threads never enter a critical section at the same time
  - Limitations
    - Less scalable with respect to the number of threads
    - Does not work on multiprocessor architectures

  - Ex. Peterson's algorithm

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

Lock Implementation: Atomic Instructions

- Hardware supports executing checking and setting at one instruction
  - e.g., test-and-set

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

- Spin-lock with test-and-set

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Enforcing Ordering Between Threads

- There would be a case where a thread must wait for another thread to complete an action before it continues
  - conditional variable (wait-notify)

- Example

== Thread 1 ==

```
shared_var = alloc(...) ;
```

== Thread 2 ==

```
shared_var->data = ... ;
```