

ITP 30002 Operating System

# Multithreading

OSTEP Chapters 26 and 28

Shin Hong

# Thread

2

- 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

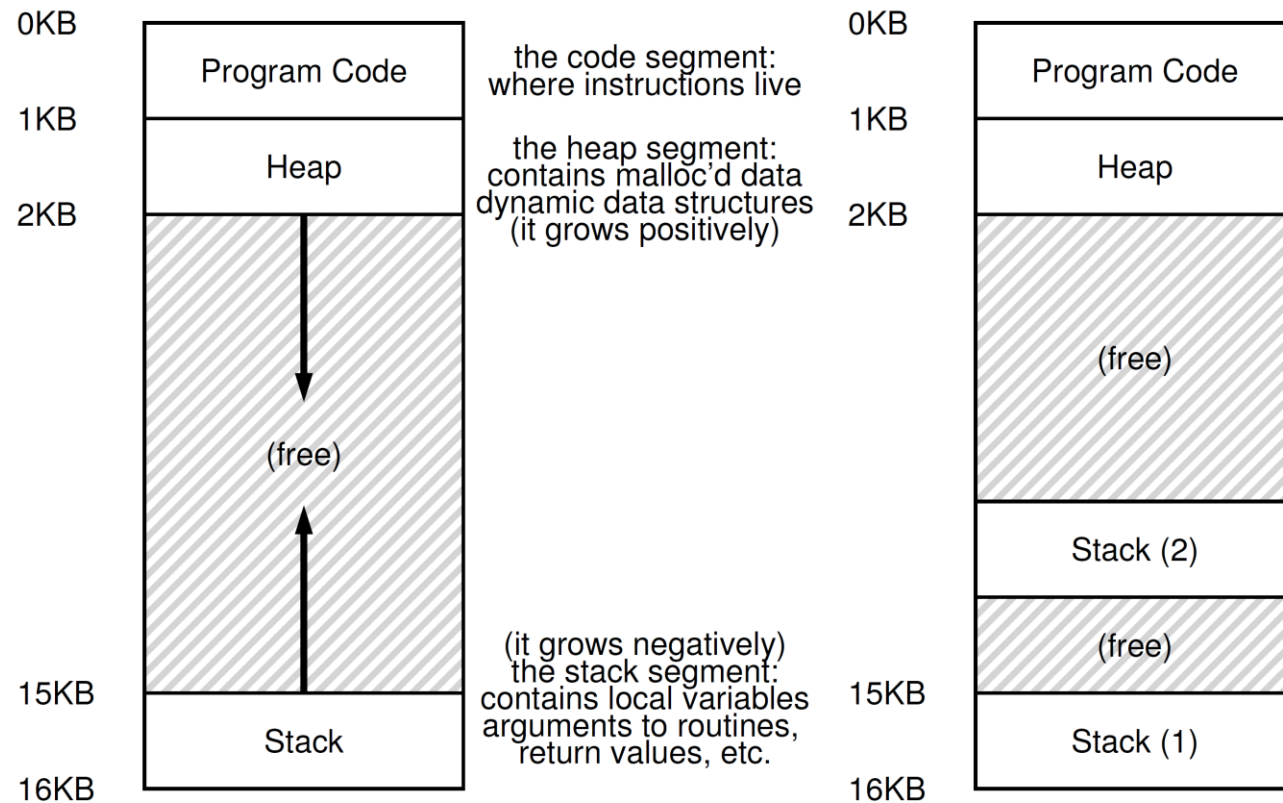
ITP 30002  
Operating System

2023-05-02

# Multithreading Mechanism

- Context switching
  - kernel threads
  - user-level threads

- Address space



# Example

4

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4  #include "common.h"
5  #include "common_threads.h"
6
7  void *mythread(void *arg) {
8      printf("%s\n", (char *) arg);
9      return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Multithreading

ITP 30002  
Operating System

2023-05-02

# Non-deterministic Scheduling

5

- 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

Multithreading

ITP 30002  
Operating System

2023-05-02

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  static volatile int counter = 0;
7
8  // mythread()
9  //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter);
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n",
39           counter);
40     return 0;
41 }

```

```


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

```

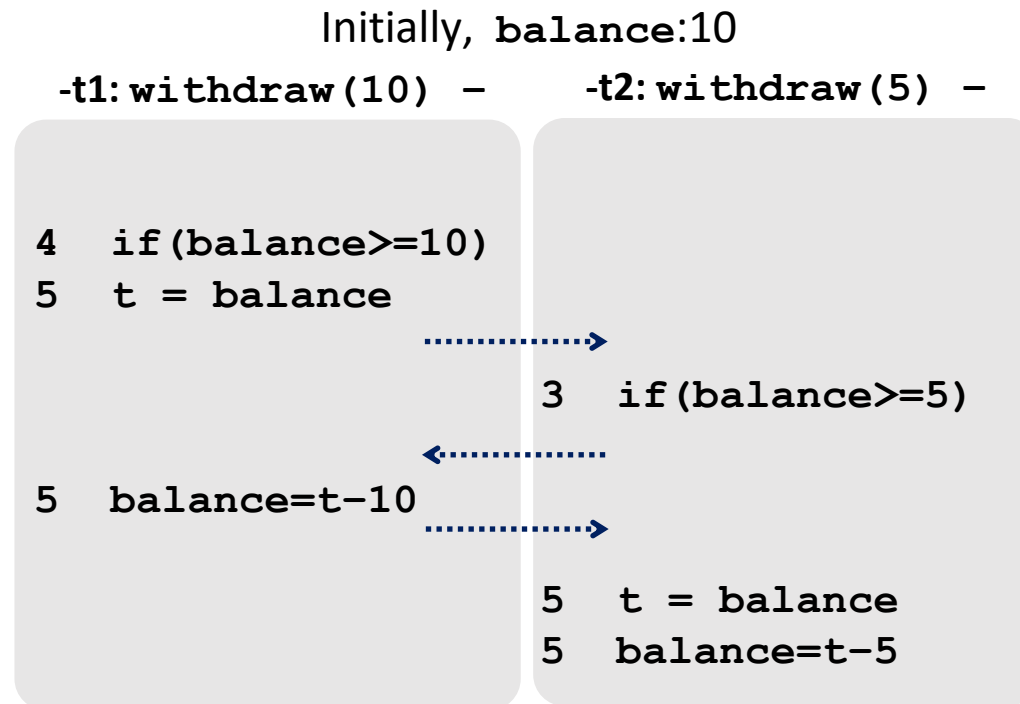
# Race Condition

7

- 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  }
```



Multithreading

ITP 30002  
Operating System

2023-05-02

# Enforcing Atomicity

8

- 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



Multithreading

ITP 30002  
Operating System

2023-05-02



# Lock

9

- 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)

Multithreading

ITP 30002  
Operating System

2023-05-02

# Lock Implementations

10

- Why naïve spin-wait does not work?
  - checking and setting of a flag are not guaranteed to be atomic

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Multithreading

ITP 30002  
Operating System

2023-05-02

# Lock Implementations: Disabling Interrupt

11

- Approach 1. disabling interrupt

- turning off timer interrupt before entering a critical section

```
1 void lock() {  
2     DisableInterrupts();  
3 }  
4 void unlock() {  
5     EnableInterrupts();  
6 }
```

- limitations

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

Multithreading

ITP 30002  
Operating System

2023-05-02

# Lock Implementations: Software Lock

12

- 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;
}
```

Multithreading

ITP 30002  
Operating System

2023-05-02

# Lock Implementation: Atomic Instructions

13

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

```
1 int TestAndSet(int *old_ptr, int new) {  
2     int old = *old_ptr; // fetch old value at old_ptr  
3     *old_ptr = new;     // store 'new' into old_ptr  
4     return old;         // return the old value  
5 }
```

- Spin-lock with test-and-set

```
1 typedef struct __lock_t {  
2     int flag;  
3 } lock_t;  
4  
5 void init(lock_t *lock) {  
6     // 0: lock is available, 1: lock is held  
7     lock->flag = 0;  
8 }  
9  
10 void lock(lock_t *lock) {  
11     while (TestAndSet(&lock->flag, 1) == 1)  
12         ; // spin-wait (do nothing)  
13 }  
14  
15 void unlock(lock_t *lock) {  
16     lock->flag = 0;  
17 }
```

Multithreading

ITP 30002  
Operating System

2023-05-02

# Enforcing Ordering Between Threads

14

- 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 = ...;
```

Multithreading

ITP 30002  
Operating System

2023-05-02