Introduction to Concurrency

03/09/2021

Professor Amanda Bienz

Textbook pages 159-171

Thread

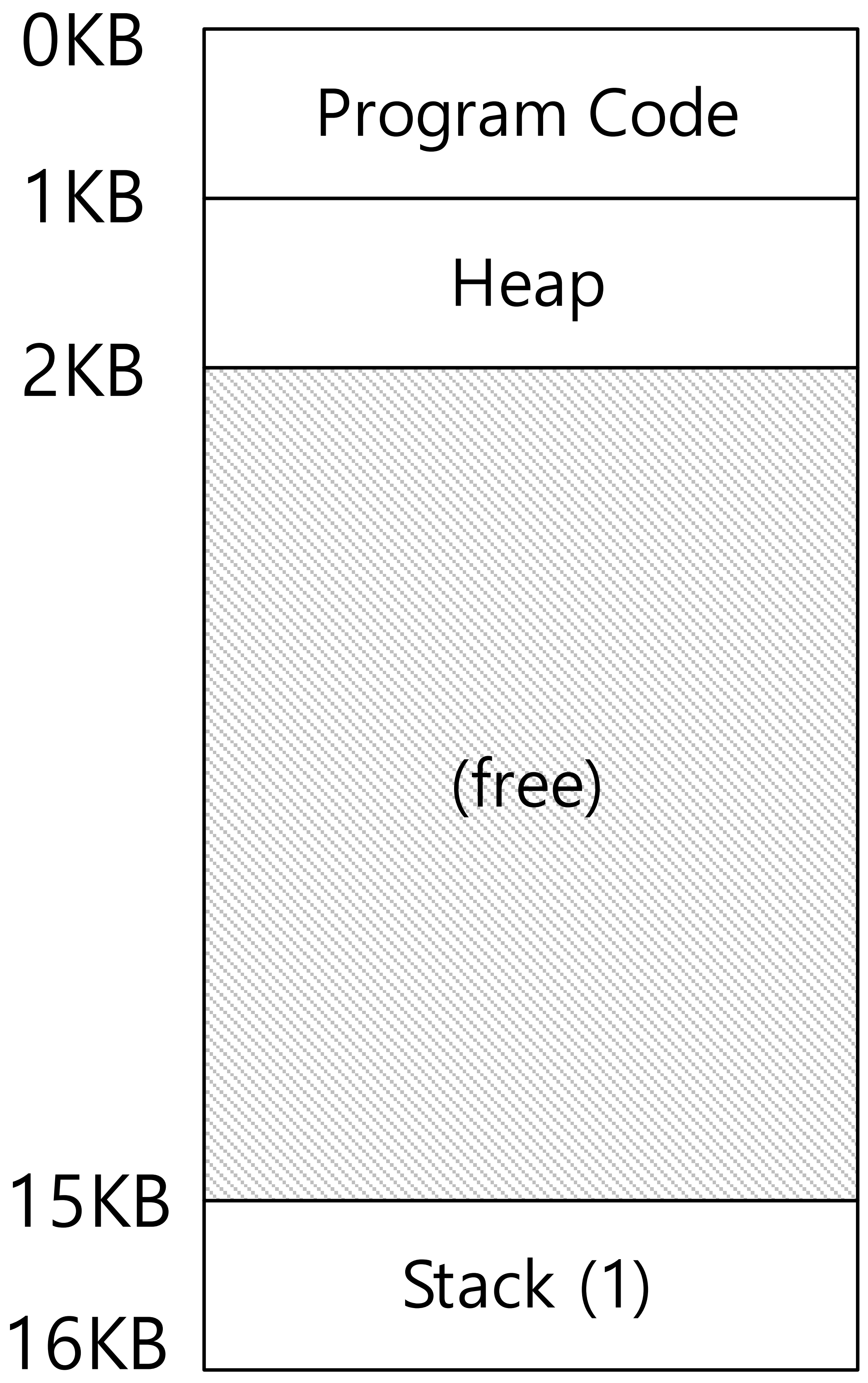
- New abstraction for **single running process**
- Multi-threaded program:
  - Multi-threaded program has more than one point of execution
  - Multiple program counters
  - They **share** the same **address space**

Context switch between threads

- Each thread has its own **program counter** and **set of registers**
  - One or more **thread control blocks (TCBs)** are needed to store the state of each thread
- When switching from running one (T1) to running the other (T2):
  - Save the register state of T1
  - Restore the register state of T2
  - Address space remains the same

The stack of the relevant thread

- There will be one stack per thread

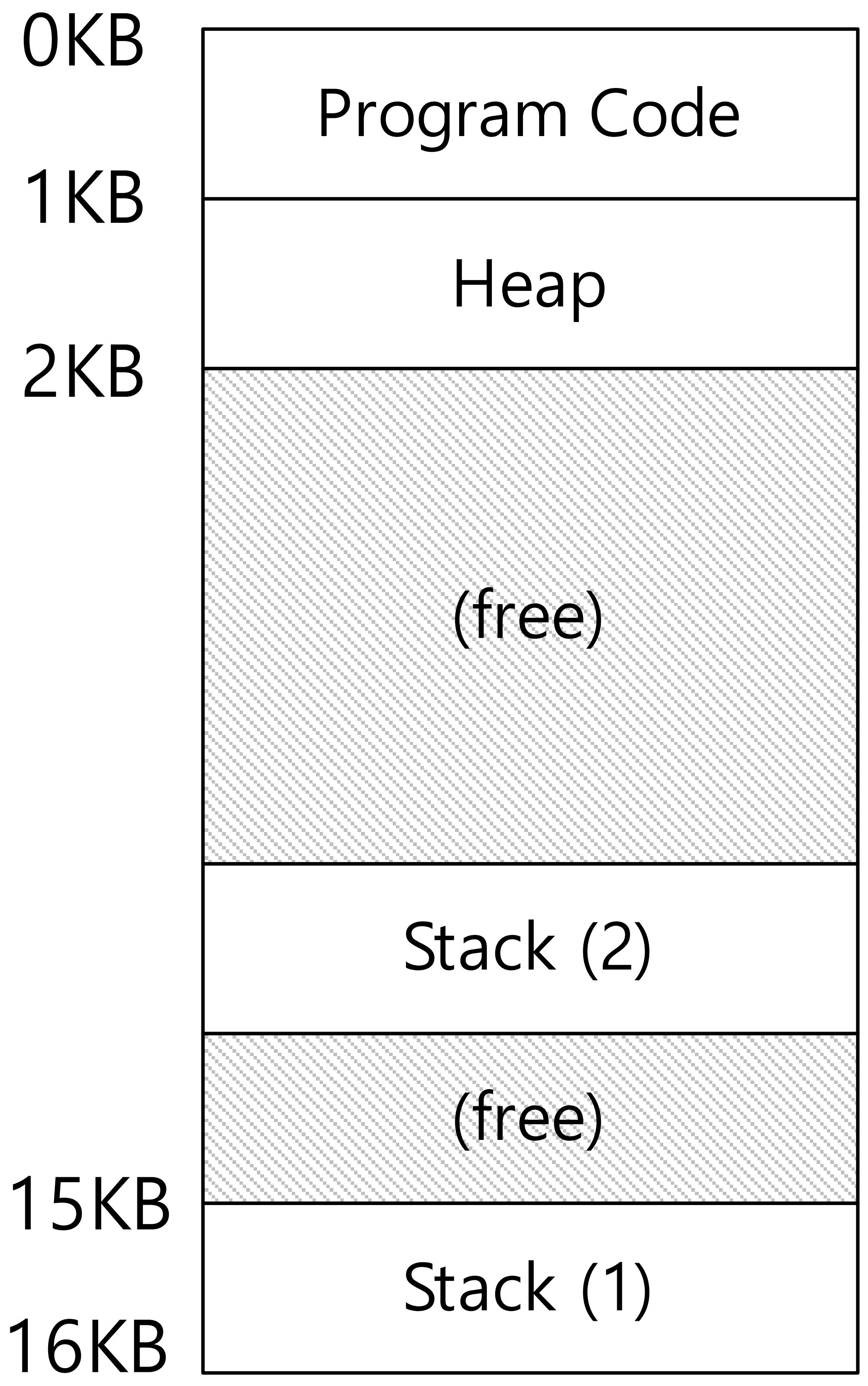


The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.

A Single-Threaded Address Space



Two threaded Address Space

Race Condition

- Example with two threads
  - Assume we have a counter variable that holds the number 50
  - Each thread executes the line : counter = counter + 1
  - What do you think the result will be? Why?

Critical Section Problem

- A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread
  - Multiple threads executing critical section can result in a race condition
  - Need to support atomicity for critical sections (mutual exclusion)
- **Goal 1 : carefully specify what a good solution to the critical section problem looks like**
- **Goal 2 : provide programming abstractions that let the user solve this problem**

Example Solution : Locks

- Ensure that any such critical section executes as if it were a single atomic instruction
  - Execute a series of instructions atomically
- lock_t mutex
  lock(&mutex)
  counter += 1
  unlock(&mutex)

Thread Creation

- How to create an control threads?

```
#include <pthread.h>

int
pthread_create(      pthread_t*         thread,
               const pthread_attr_t*    attr,
                     void*              (*start_routine)(void*),
                     void*              arg);
```

- `thread`: Used to interact with this thread.
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, …
- `start_routine`: the function this thread start running in.
- `arg`: the argument to be passed to the function (`start routine`)
  - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

- If start_routine instead required another type of argument, the declaration would look like this:
  - An integer argument:

```
int
pthread_create(…, // first two args are the same
                  void*  (*start_routine)(int),
                  int     arg);
```

  - Return an integer:

```
int
pthread_create(…, // first two args are the same
                  int  (*start_routine)(void*),
                  void*    arg);
```

Example : Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    …
}
```

Wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Example : Waiting for Thread Completion

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}
```

Example : Waiting for Thread Completion (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                  // waiting inside of the
  // pthread_join() routine.
    printf(“returned %d %d\n”, m->x, m->y);
    return 0;
}
```

Example : Dangerous Code

- Be careful with **how values are returned** from a thread

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

Example : Simpler Argument Passing to a Thread

- Just passing a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf(“%d\n”, m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf(“returned %d\n”, m);
    return 0;
}
```

Locks

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- All locks must be properly initialized
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- **These two calls are used in lock acquisition**

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - trylock: return failure if the lock is already held
  - timelock: return after a timeout

Compiling and Running

- **To compile them, you must include the header `pthread.h`**

```
prompt> gcc -o main main.c -Wall -pthread
```

  - Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
man -k pthread
```

- **There are a lot more pieces to pthreads. We'll see more abstractions and useful examples later**

Reading

- For Thursday, read pages 257-275
  - Synchronization (critical sections, atomic operations, locks)