Operating Systems

KAIST

26. Concurrency: An Introduction

Thread

- A new abstraction for <u>a single running process</u>

- Multi-threaded program
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They share the share the same address space.

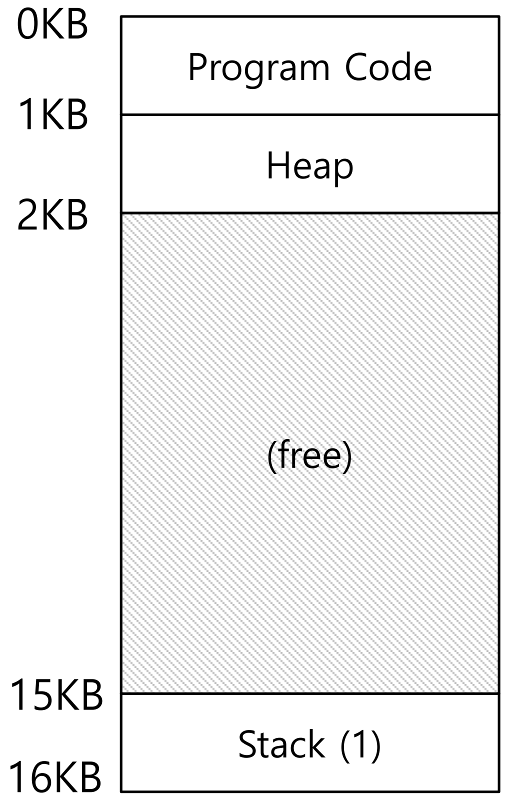
Context switch between threads

- 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),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The address space remains the same.

The stack of the relevant thread

- There will be one stack per thread.

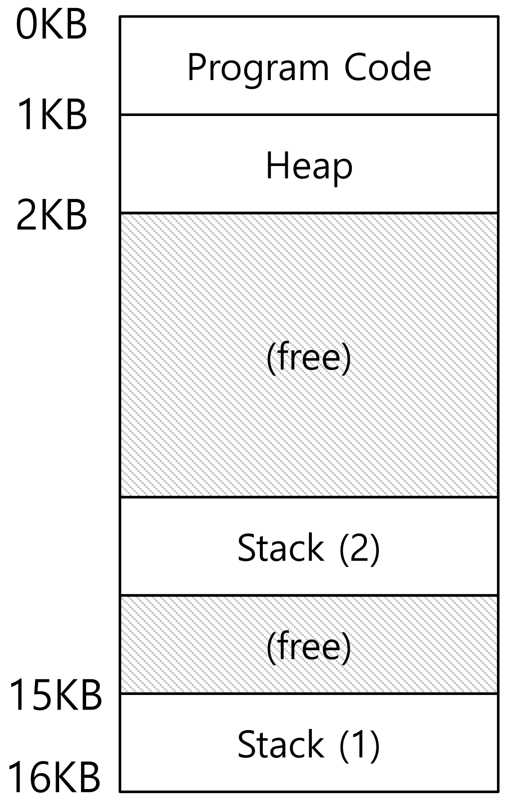


The code segment: where instructions live

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

Why Use Threads?

- Parallelism
  - Single-threaded program: the task is straightforward, but slow.
  - Multi-threaded program: natural and typical way to make programs run faster on modern hardware.
  - **Parallelization**: The task of transforming standard **single-threaded** program into a program that does this sort of work on multiple CPUs.
- Avoid blocking program progress due to slow I/O.
  - Threading enables **overlap** of I/O with other activities within a single program.
  - It is much like **multiprogramming** did for processes across programs.

An Example: Thread Creation

- Simple Thread Creation Code (`t0.c`)

```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 thre threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf(“main: end\n”);
    return 0;
}
```

Thread Trace (1)

main	Thread 1	Thread 2
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"		

Thread Trace (2)

main	**Thread 1**	**Thread 2**
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"		

Thread Trace (3)

main	Thread 1	Thread 2
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"		

Race condition

- Increasing a value of a variable
  - counter = counter + 1

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

Race condition

- Example with two threads
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

			(after instruction)		
OS	Thread1	Thread2	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	50
	`mov %eax, 0x8049a1c`		113	$\boxed{51}$	**51**

A few terminologies

- Race condition:
  - the results depend on the timing execution of the code.
  - Result is indeterminate.
- Critical section
  - 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**)

The wish for atomicity

- Ideal approach; make the increment as a single assembly instruction

```
memory-add 0x8049a1c, $0x1
```

- In general, we do not have such instruction. Instead, we use lock.
- 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);
balance = balance + 1;
unlock(&mutex);
```

Critical section