26. Concurrency: An Introduction

Operating System: Three Easy Pieces

Thread

- A new abstraction for a single running process

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

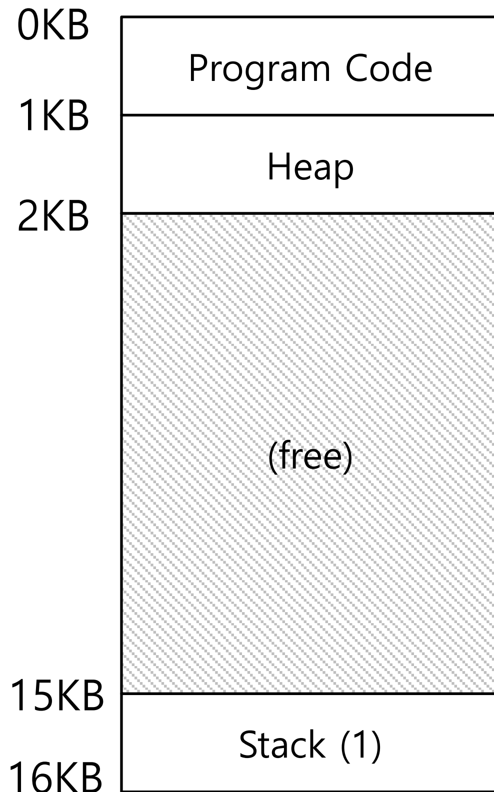
Context switch between threads

- 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.
  - All of then within a common PCB

- 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**.

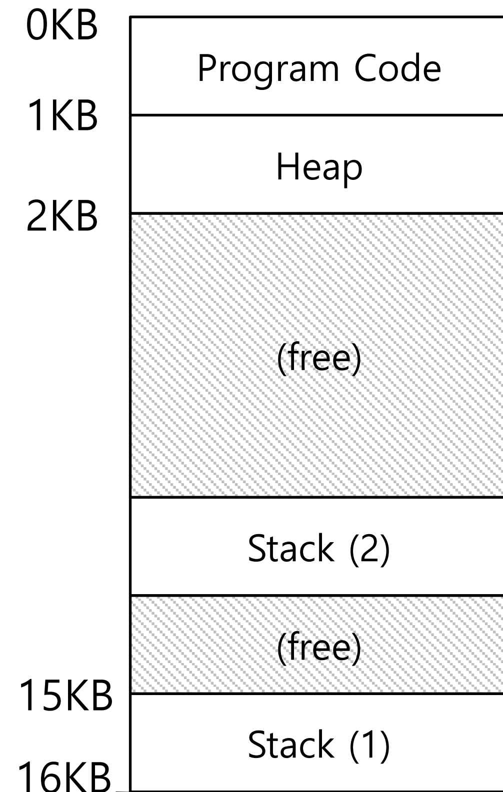


The code segment: where instructions live

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



Thread-local storage

Two threaded Address Space

Why threads?

- Performance
  - **Parallelism** is the only way to use translate multiple cores into performance
  - Parallelization: from single-threaded programs to multi-threaded
- Convenience
  - Way to **overlap** I/O with useful work: approach of server-base applications such as web-servers, DBMS, etc..
- Why threads and not processes?
  - In threads is much **easier** to **share data**
  - Less pressure over the memory
  - Processes when the task are separated with little (to none) sharing

Example

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

Figure 26.2: **Simple Thread Creation Code (t0.c)**

Possible outcomes

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"		

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"		

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"		

Uh Oh

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
6
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 *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
26 //
27 // main()
28 //
29 // Just launches two threads (pthread_create)
30 // and then waits for them (pthread_join)
31 //
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

Possible outcomes

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
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)
```

The heart of the problem: : Uncontrolled Scheduling

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

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	mov 0x8049a1c, %eax		100	0	50
	add $0x1, %eax		105	50	50
			108	51	50
interrupt save T1's state (TCB) restore T2's state (TCB)			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	51	**51**

The wish for atomicity

- Do the read and modification of the memory in a single step
  - i.e. "**all or nothing**"!
- How ho handle complex data? (v.gr. a b-tree)
  - Use some atomic hardware support (called **synchronization primitives**) to construct OS support
- A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread (mixing R and W).
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
1    lock_t mutex;
2    . . .
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

Critical section

One more problem: Waiting for another/s

- Sometimes the thread interaction is wait for another thread
  - V.gr. When a thread should wait to another that had issued a I/O
  - Need to be **slept** until the other thread receives the I/O end
- Sometimes the action of multiple threads should be synchronous
  - V.gr. Many threads are performing in parallel an iteration in a numerical problem
  - All threads should start the next iteration at once (**barrier**)
- This sleeping/waking cycle will be controlled by **condition variables**

- This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)