26. Concurrency: An Introduction

Operating System: Three Easy Pieces

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 same address space.

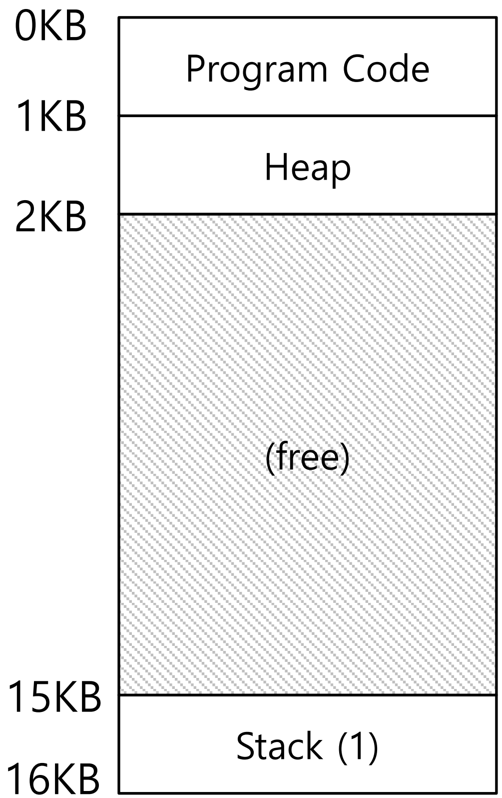
Context switch between threads

- 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.

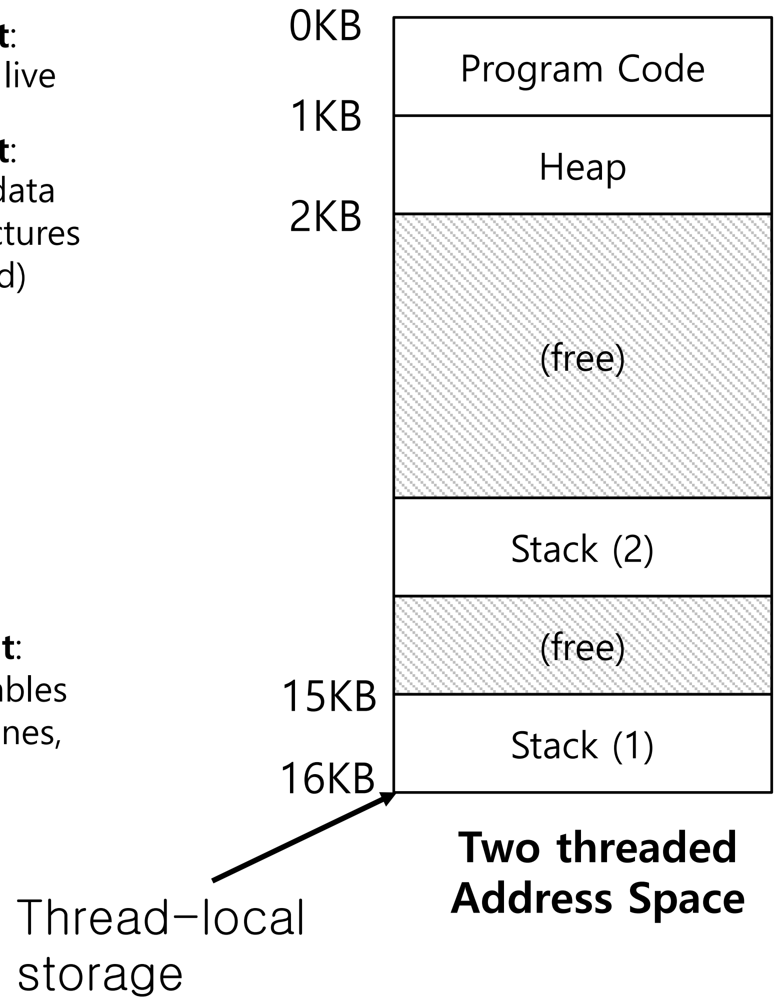


A Single-Threaded Address Space

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.



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..

Example

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.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");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

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

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.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;
}
```

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	PC	%eax	counter
			(after instruction)		
	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	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.
  - 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**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- Disclaimer: Disclaimer: 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)