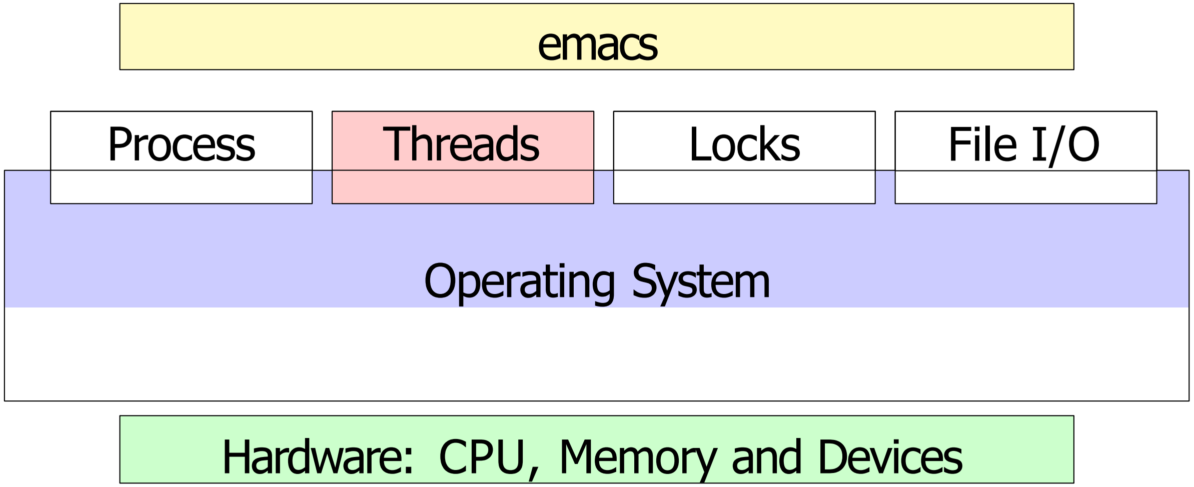
Operating Systems

Threads

These slides include content from the work of:
Youjip Won, KIAT OS Lab

Today: Threads

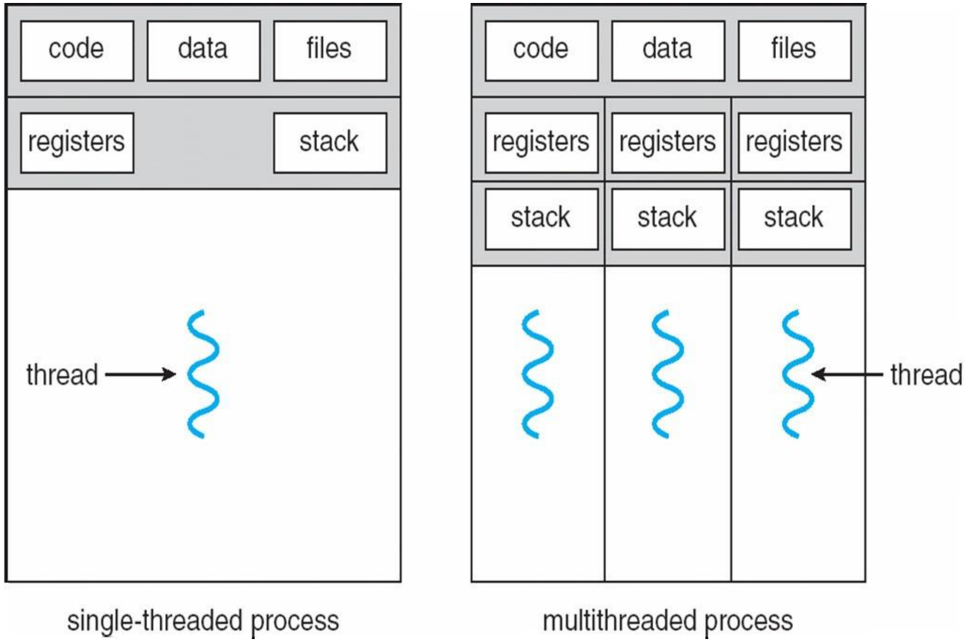


Thread

- A new abstraction for a single running process
- A thread is a schedulable execution context
- It comprises a thread ID, a program counter (PC), a register set, and a stack.

- Multi-threaded program
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They share the same address space.
  - If a process has multiple threads of control, it can perform more than one task at a time.

Threads



- A thread is a schedulable execution context
  - Program counter, registers, stack (local variables) . . .
- Multi-threaded programs share the address space (global variables, heap, . . .)

Context switch between threads

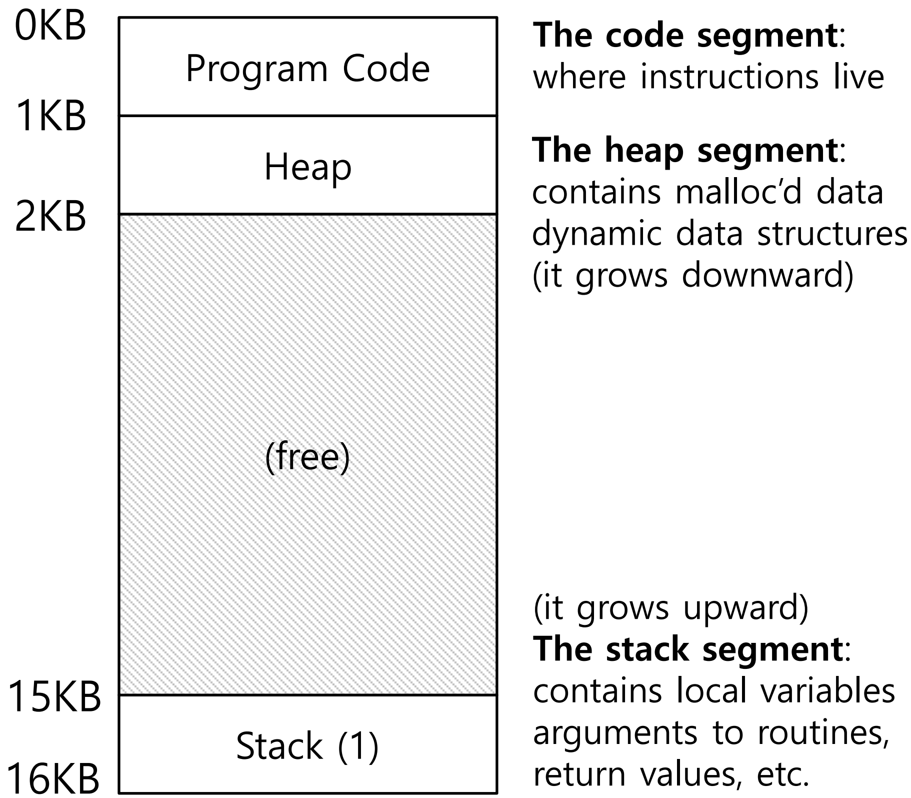
- Each thread has its own program counter and set of registers.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.

- 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.

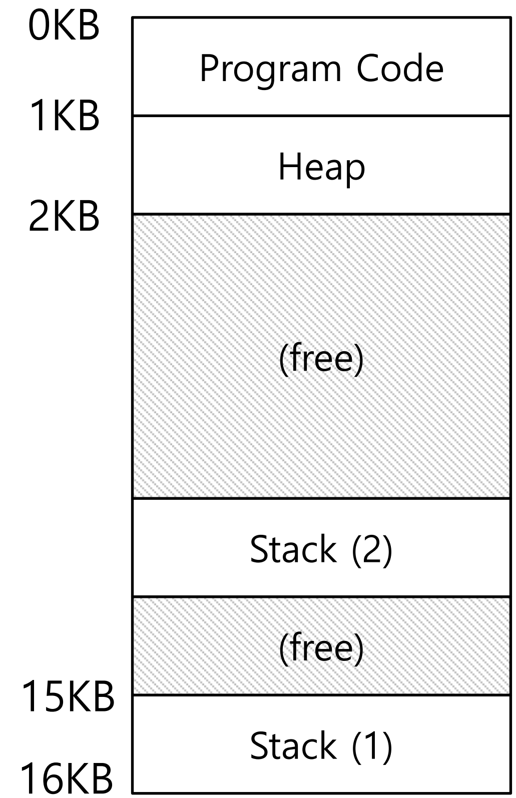
- With processes, we saved state to a PCB; now, we'll need TCBs to store the state of each thread of a process.

The stack of the relevant thread

- There will be one stack per thread.



A Single-Threaded Address Space



Two threaded Address Space

Why Use Threads?

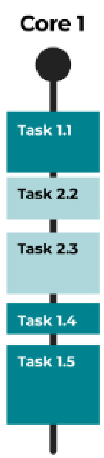
- Most popular abstraction for concurrency
  - Lighter-weight abstraction than processes: **Shared Resources, Context Switching,…**
  - All threads in one process share memory, file descriptors, etc.
- Parallelism
  - Single-threaded program: the task is straightforward, but slow.
  - Multi-threaded 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.

Why Use Threads?

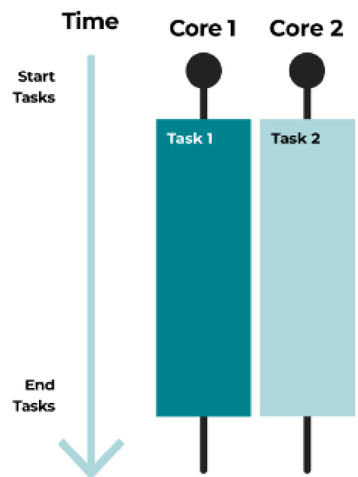
- Of course, in either of the cases mentioned above, you could use multiple processes instead of threads.
- However, threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs.
- Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

Concurrency vs Parallelism

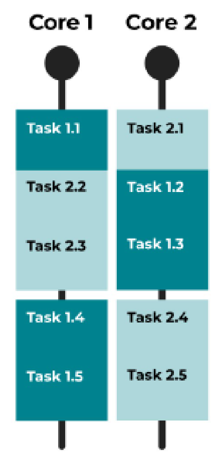
Concurrency



Parallelism



Concurrency& Parallelism



Multi-threading Examples

- Web Servers:
  - use multi-threading to handle multiple client requests simultaneously.
- Web Browsers:
  - use multi-threading to load different elements of a webpage (like text, images, and scripts) at the same time.
- Office Applications:
  - Microsoft Word can use multi-threading to perform spell-checking, auto-saving, and complex calculations in the background while still being responsive to the user.
- Operating Systems:
  - use multi-threading to manage multiple tasks at once, such as running background services, managing user inputs, and updating the user interface.
- Machine Learning Frameworks:
  - Tools like TensorFlow or PyTorch use multi-threading to parallelize data loading and processing.

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;
    printf(“main: begin\n”);
    Pthread_create(&p1, NULL, mythread, “A”);
    Pthread_create(&p2, NULL, mythread, “B”);
    // join waits for their 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"		

Thread Creation

- How to create and 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.

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