238P Operating Systems, Fall 2018

Threads and Locks

30 November 2018

Aftab Hussain
University of California, Irvine

Threads fundamentals
Creating threads
Waiting for threads
The need for locks
Spin locks
Mutexes
Condition variables
Semaphores

Thread fundamentals

Threads are similar to processes...

Can have **similar states** as processes
e.g. ready/waiting/terminated/blocked

Have **PCs** to point to the location of current instruction

Have their private set of **registers**

Thread fundamentals

Threads are similar to processes...

Can have **child threads**, just as processes can have child processes.

Thread fundamentals

Threads are similar to processes...

Can have **child threads**, just as processes can have child processes.

When switching between threads, save the state of the thread to a **thread control block**, similar to saving **process control blocks** while switching between processes.

Thread fundamentals

Threads are similar to processes...

Can have **child threads**, just as processes can have child processes.

When switching between threads, save the state of the thread to a **thread control block**, similar to saving **process control blocks** while switching between processes.

A thread can thus be viewed as a separate process.

Thread fundamentals

however....

They **share the same address space** of the process that created them.

Thread fundamentals

however....

they **share the address space** of the process that created them

Threads of the same process can thus read and update all variables in the process's address space in parallel

Thread fundamentals

however....

they **share the address space** of the process that created them

Threads of the same process can thus read and update all variables in the process's address space in parallel

This is one advantage of using threads: ***parallelizing a single task***

Thread fundamentals

Each thread maintains its own stack

They might execute different code,
call different function
use different arguments.

Thread fundamentals

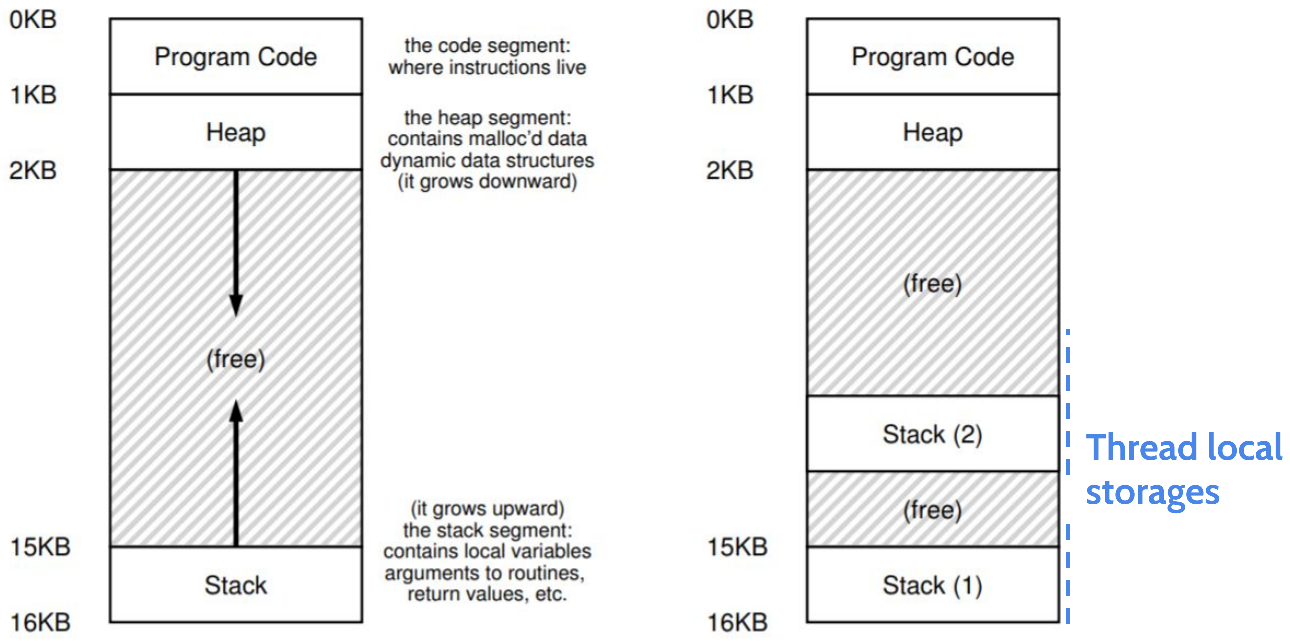

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

Thread fundamentals

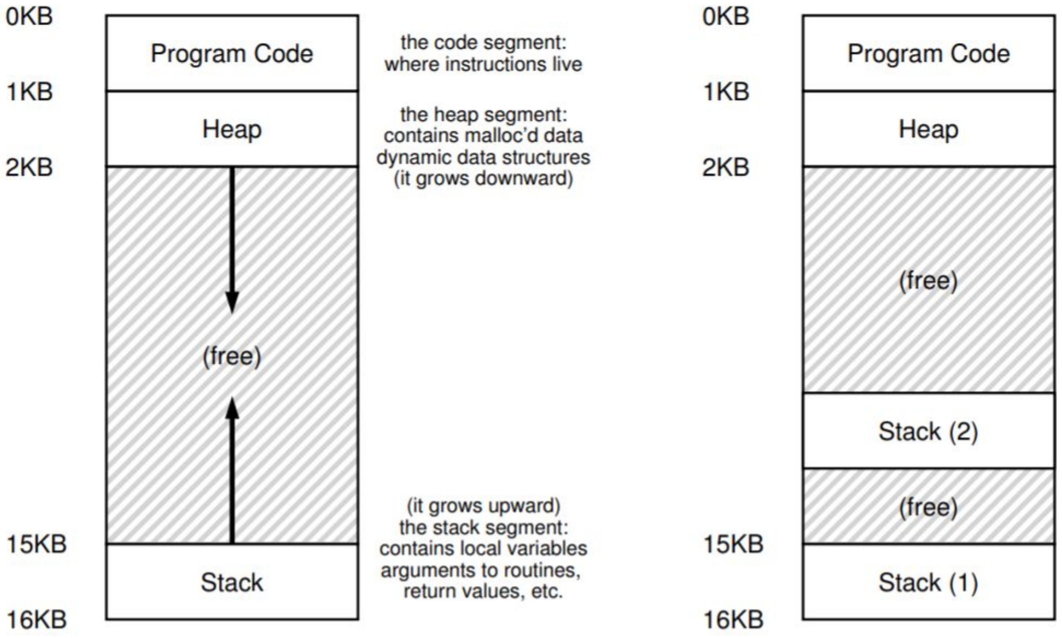

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

On space availability, stacks are small, which is OK. But ***recursion*** *can make things different..*

http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

Creating Threads

Invoke a thread create function supplying:

> the function pointer to the function
that you want the thread to execute

> a pointer to a stack (pre-allocated by the parent process)

> the input argument to the function

It returns the PID of the new thread to the parent.

Creating Threads

Needs to be implemented as a system call in HW4.

Invoke a thread create function supplying:

> the function pointer to the function that you want the thread to execute

> a pointer to a stack (pre-allocated by the parent process)

> the input argument to the function

It returns the PID of the new thread to the parent.

Creating Threads

Let’s see the HW4 input example.

Creating Threads

Once threads are created, the **OS scheduler** decides how and when to run them.

They may be run immediately, or kept in a ready state.

Creating Threads

Once threads are created, the **OS scheduler** decides how and when to run them.
They may be run immediately, or kept in a ready state.

The first thread is always the **main thread**, from which the child threads are created.

Creating Threads

Once threads are created, the **OS scheduler** decides how and when to run them.
They may be run immediately, or kept in a ready state.

The first thread is always the **main thread**, from which the child threads are created.

The main thread can be made to wait for the child threads to finish.
We use a join function - similar to wait() used for processes.

Creating Threads

Once threads are created, the **OS scheduler** decides how and when to run them.
They may be run immediately, or kept in a ready state.

Uncontrolled scheduling can lead to non-deterministic behaviour - hence the need for ***locks.***

d is always the **main thread**, from
d threads are created.

ead can be made to wait for the child threads to finish.
We use a join function - similar to wait() used for processes.

Syncing

Let’s go back to HW 4’s synchronization part.

see the **data race**

Syncing

We need a locking mechanism.

We need some part of the code to be **mutually exclusive**, i.e., only one thread can work on it at a time.

Let’s first look at a lock that doesn’t work,
and then a lock that does.

Syncing

Spinning threads who can’t get access
can be inefficient.

Instead of spinning threads which can’t get access right away, **put them to sleep**.

Goto “Mutexes section” paragraph 2 in HW4.

Syncing

When to wake them up?

Syncing

When to wake them up?

Use **condition variables**

Goto “Condition variables section” in HW4.

Producer consumer problem

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Counting Semaphore
[Whiteboard Explanation]

References:

producer consumer problem
https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

volatile keyword
https://www.youtube.com/watch?v=W3pFxSBkeJ8

OSSTEP Remzi