Lecture 11: Context Switching Inside the User Space

Vivek Kumar

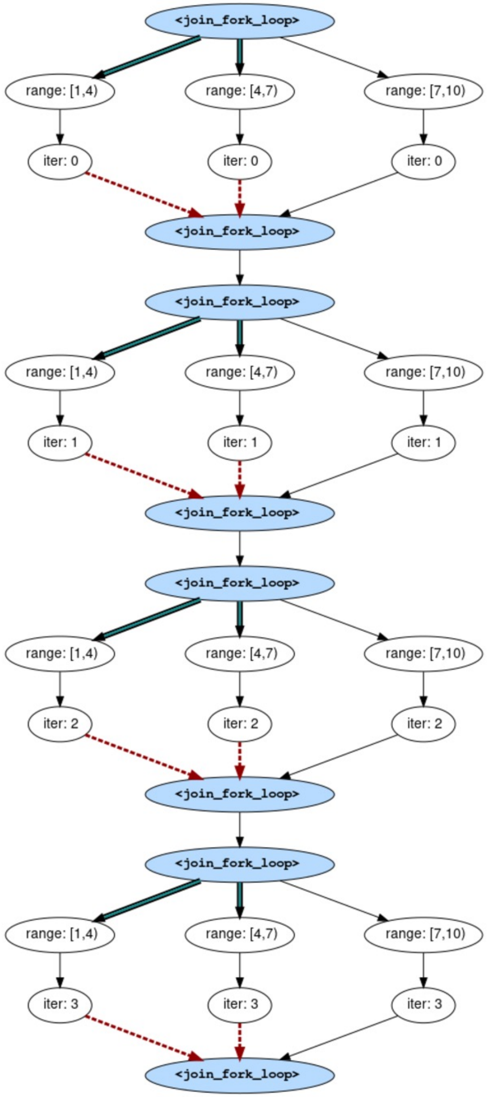
Computer Science and Engineering

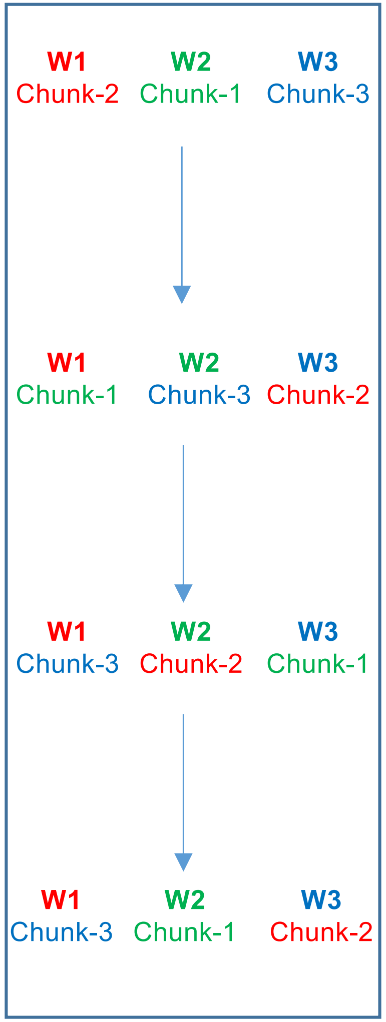
IIIT Delhi

vivekk@iiitd.ac.in

Trace/Replay

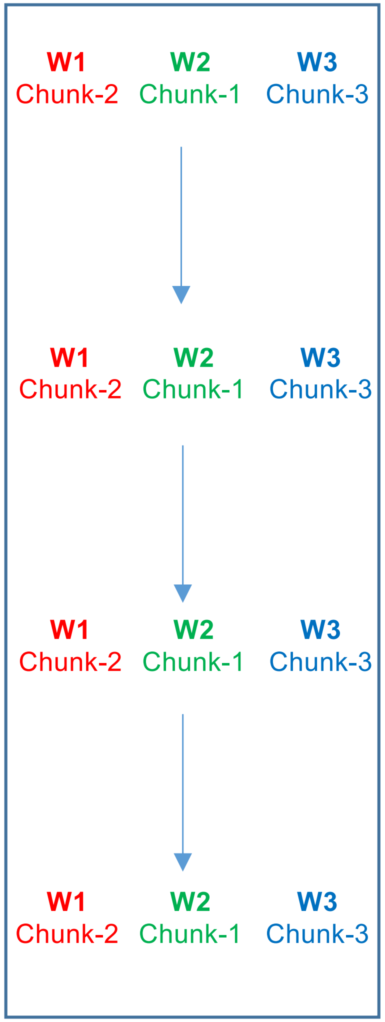
- Improved locality if each workers executes the exact same set of tasks in each for loop iteration of compute
- Trace/Replay for improving locality
  - ○ Trace (i.e., record) the tasks executed by each worker during the first iteration of for loop inside compute
  - ○ For the rest of iterations of the above for loop of compute, disable random work-stealing and use the information gathered during the Trace (i.e., record) phase to replay the exact set of tasks at each worker





W1	W2	W3
Chunk-2	Chunk-1	Chunk-3
W1	W2	W3
Chunk-1	Chunk-3	Chunk-2
W1	W2	W3
Chunk-3	Chunk-2	Chunk-1
W1	W2	W3
Chunk-3	Chunk-1	Chunk-2

Random work-stealing



W1	W2	W3
Chunk-2	Chunk-1	Chunk-3
W1	W2	W3
Chunk-2	Chunk-1	Chunk-3
W1	W2	W3
Chunk-2	Chunk-1	Chunk-3
W1	W2	W3
Chunk-2	Chunk-1	Chunk-3

Trace/Replay

Today’s Class

- More about thread
  - Stack
  - Scheduling
- Boost C++ libraries for concurrency
  - Context

Callee and Caller

```
L1: main () {

L2:   bar(100);

L3: }
```

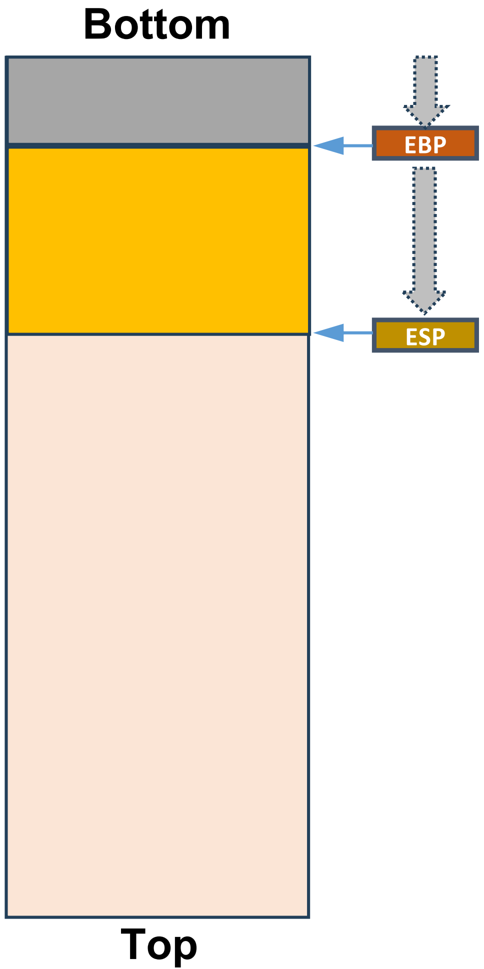
```
L4: bar(int x) {

L5:   int b1=x1;

L6: }
```

- Call chain
  - main → bar
- Callee and Caller
  - main (caller) → bar (callee)

Thread Stack Frames in X86

```
L1: main () {

L2:   bar(100);

L3: }
```

```
L4: bar(int x) {

L5:   int b1=x1;

L6: }
```

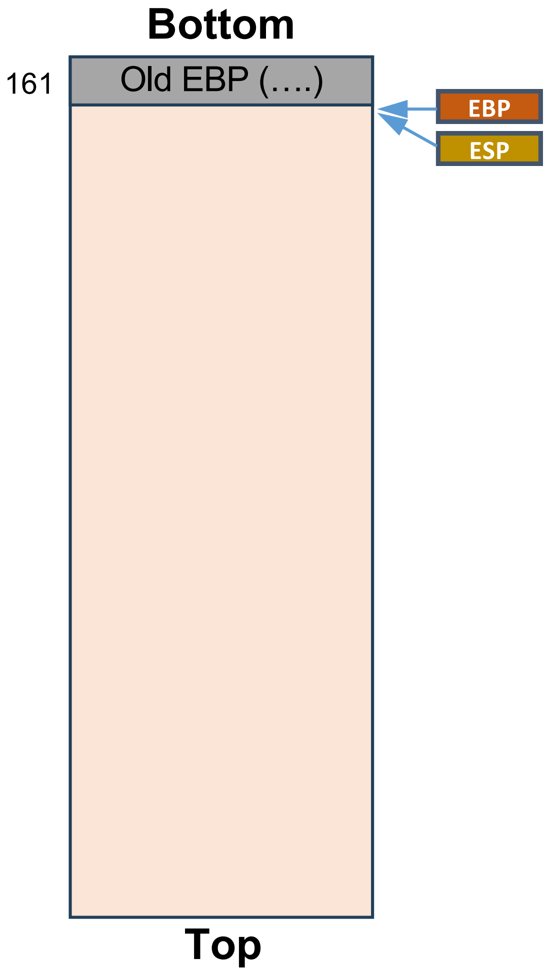


- **EBP** register points to **bottom** of the stack (first item pushed on the stack) and **ESP** register points to the **top** of the stack (last item pushed on stack)
- Area of the stack between the location pointed by EBP and ESP is called **stack frames** for that method
- The **topmost frame** corresponds to the currently executing method

Thread Stack in X86 (1/7)

```
L1: main () {
        <save_regs>

L2:   bar(100);

L3: }
```

```
L4: bar(int x) {

L5:   int b1=x1;

L6: }
```

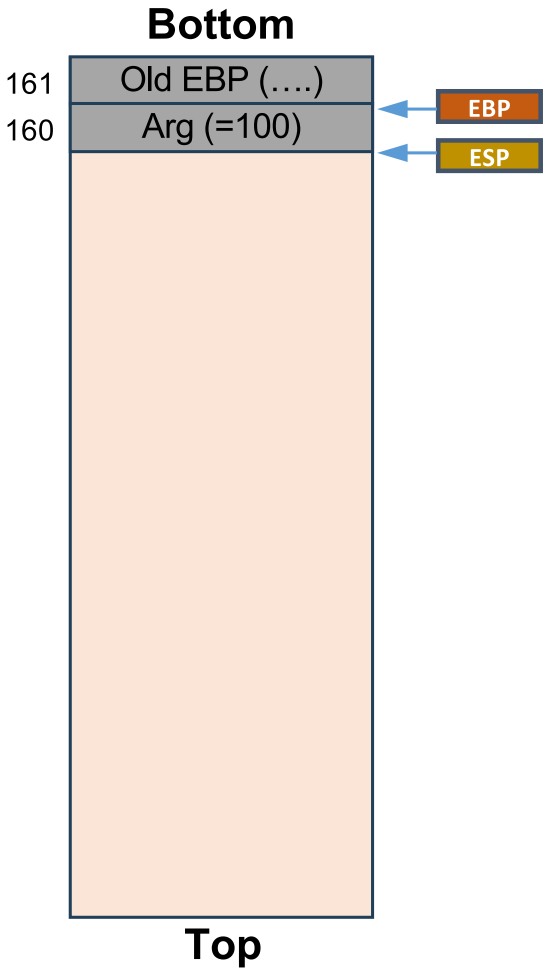


- <save regs>
  - Current content of EBP register is saved in stack
    - Carried out before every method call
    - Also known as **method prologue**
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

Thread Stack in X86 (2/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>

L2:   bar(100);



L3: }
```

```
L4: bar(int x) {


L5:   int b1=x1;


L6: }
```

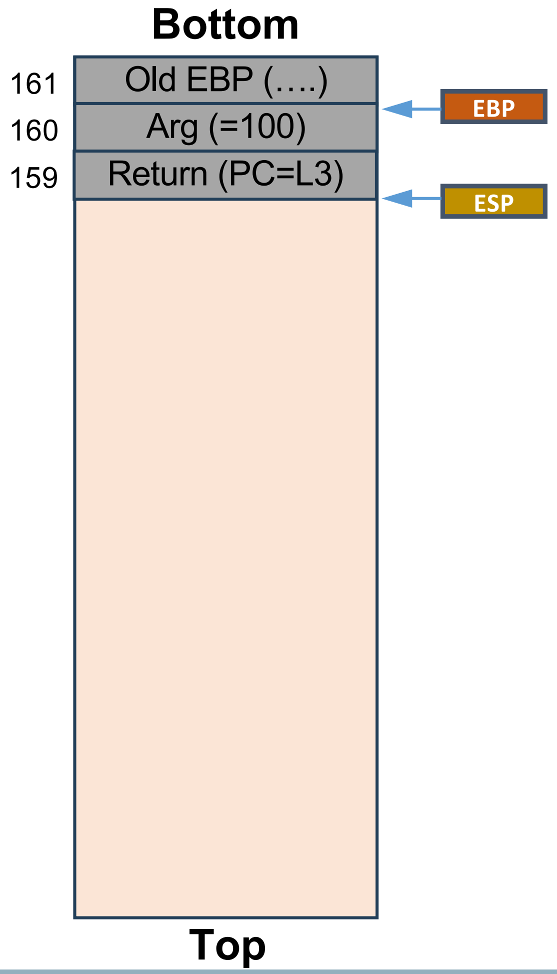


- <alloc(4)>
    - One slot is added on the stack in the current stack frame for the argument to the callee method
    - ESP is updated

Thread Stack in X86 (3/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>
        <save(PC)>
L2:   bar(100);



L3: }
```

```
L4: bar(int x) {


L5:   int b1=x1;


L6: }
```



Bottom

161 Old EBP (….)

160 Arg (=100)

159 Return (PC=L3)

EBP

ESP

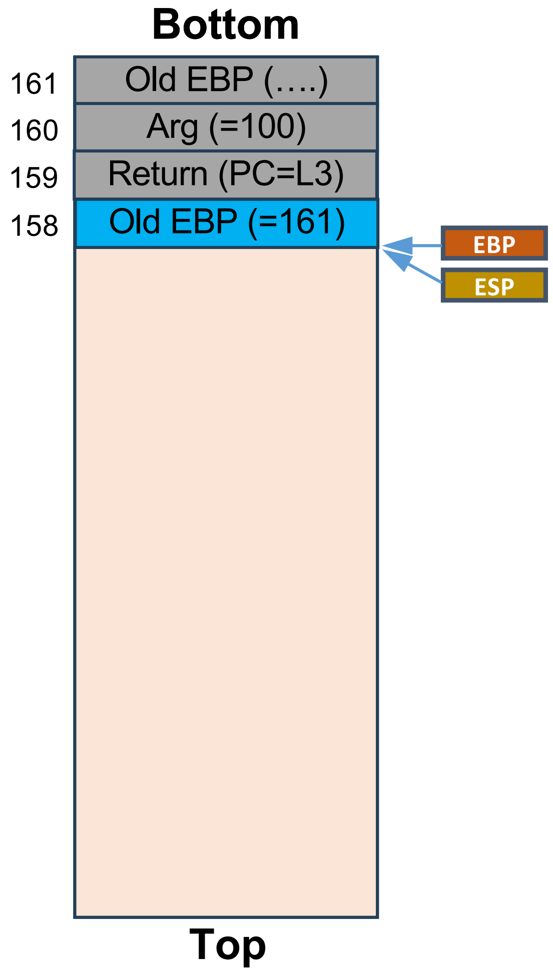
Top

- <save (PC)>
  - Current content of **EIP** register is saved in stack
  - ESP is now pointing to newly added slot

Thread Stack in X86 (4/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>
        <save(PC)>
L2:   bar(100);

L3: }
```

```
L4: bar(int x) {
        <save_regs>

L5:   int b1=x1;

L6: }
```



Bottom

161	Old EBP (….)
160	Arg (=100)
159	Return (PC=L3)
158	Old EBP (=161)

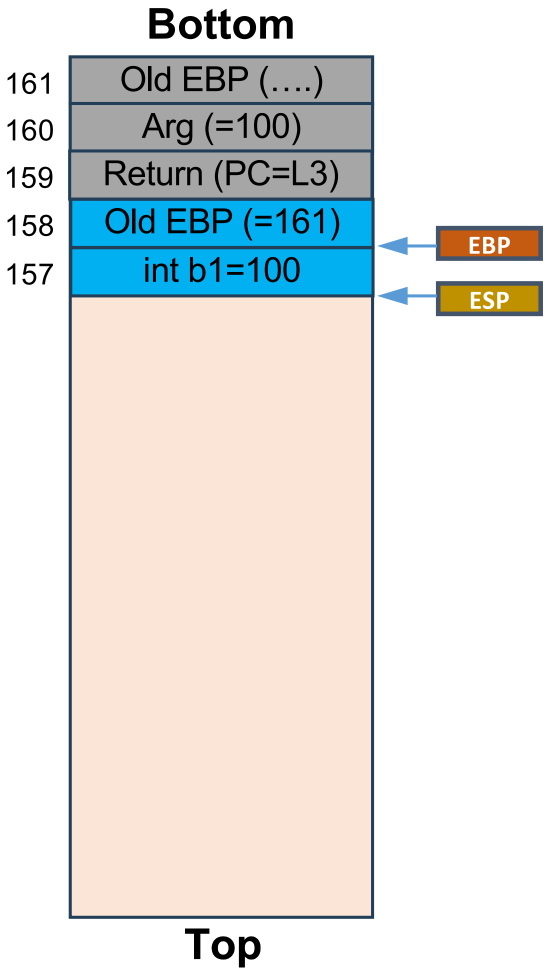
Top

- <save regs>
  - Current content of EBP register is saved in stack
  - EBP now points to the slot containing old value of EBP
  - ESP also points to the same slot

Thread Stack in X86 (5/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>
        <save(PC)>
L2:   bar(100);



L3: }
```

```
L4: bar(int x) {
        <save_regs>
        <alloc(4)>
L5:   int b1=x1;


L6: }
```

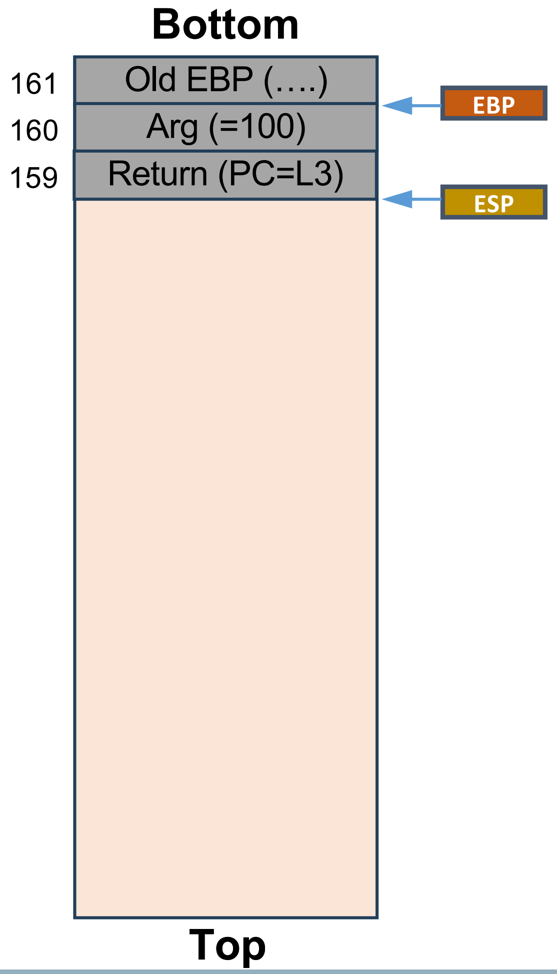


- <alloc(4)>
  - One slot is added on the stack in the current stack frame for the local variable
  - ESP is updated

Thread Stack in X86 (6/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>
        <save(PC)>
L2:     bar(100);



L3: }
```

```
L4: bar(int x) {
        <save_regs>
        <alloc(4)>
L5:     int b1=x1;
        <dealloc(4)>
        <restore_regs>
L6: }
```

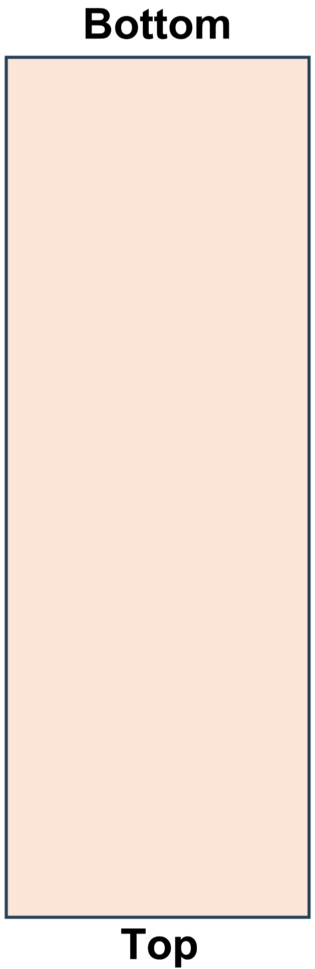


- <dealloc(4)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

Thread Stack in X86 (7/7)

```
L1: main () {
        <save_regs>
        <alloc(4)>
        <save(PC)>
L2:     bar(100);
        <restore(PC)>
        <dealloc(4)>
        <restore_regs>
L3: }
```

```
L4: bar(int x) {
        <save_regs>
        <alloc(4)>
L5:     int b1=x1;
        <dealloc(4)>
        <restore_regs>
L6: }
```



- <restore(PC)>
  - EIP register is now restored with the old value of EIP before the call went inside the callee
- <dealloc(4)>
  - ESP is updated to point to the slot just before the slot(s) added for storing local variable(s)
- <restore_regs>
  - Old value of EBP is popped and stored in EBP
  - ESP adjusted to point to the last slot in the current stack frame

CPU Scheduling

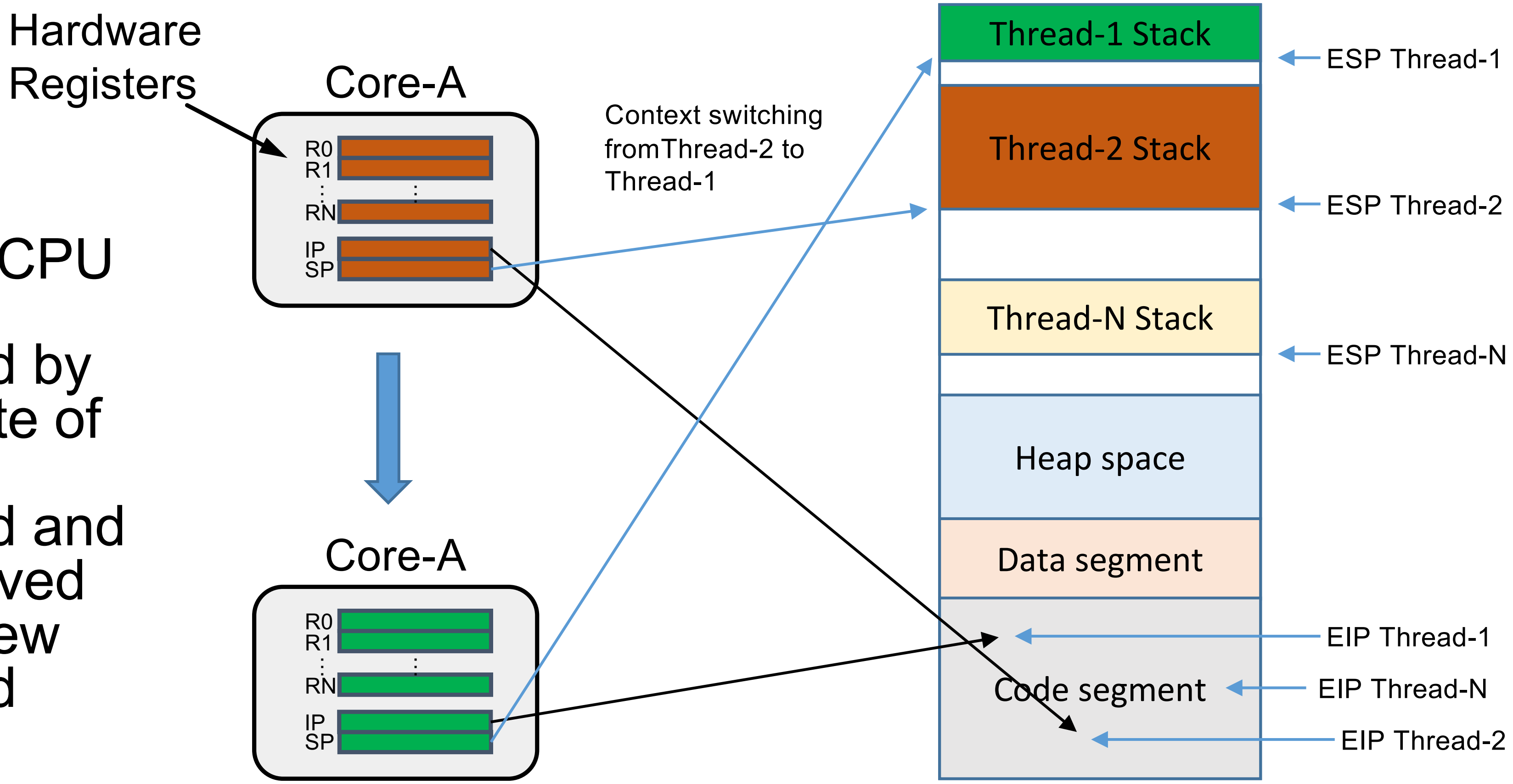
- Cooperative
  - Processes/threads decide when to yield the CPU
- Preemptive (e.g., used by Linux kernel scheduler)
  - Processes/threads preempted at blocking points
    - Blocking calls
      - IO
      - Sleep
      - Wait (locking)
    - Interrupts
- Context switch required in each case!

Context Switch: Why?

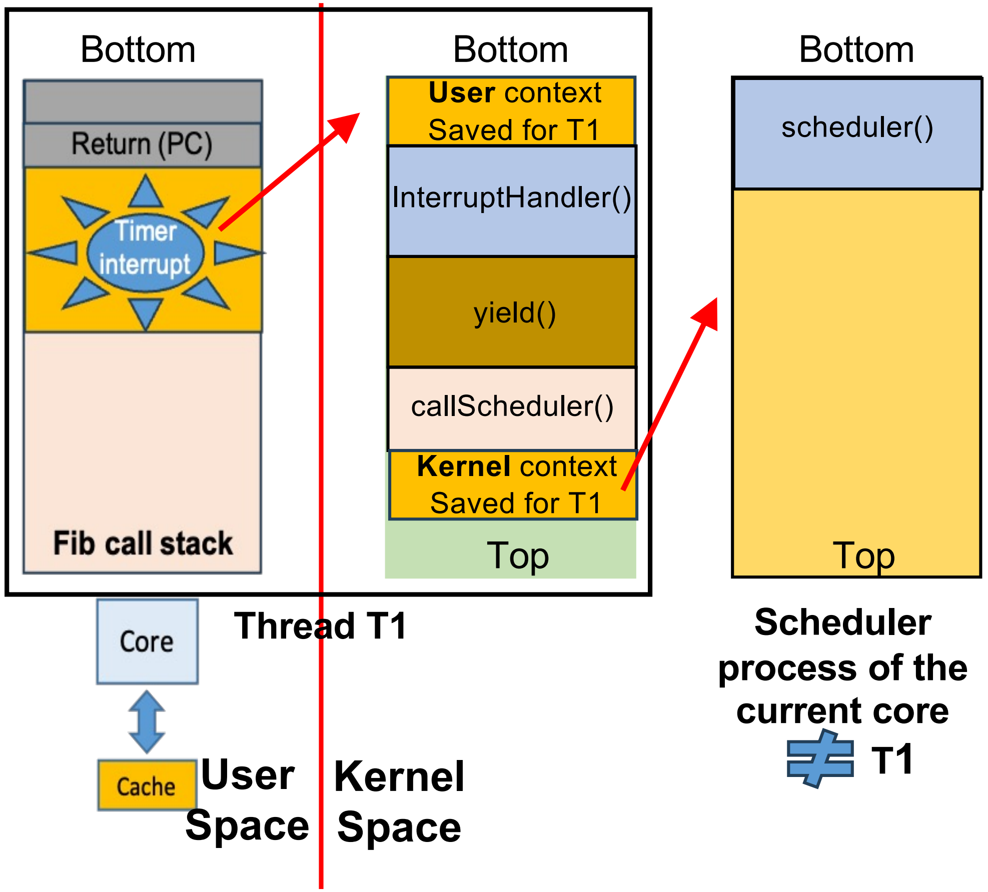
- Could happen due to several reasons
  - Blocking operations (IO, synchronizations, etc.)
  - Arrival of a high priority process
  - Process terminating
  - Process has exhausted its allotted CPU slice
    - It would be the primary reason when several processes/threads are being used for running parallel program(s)

Context Switch: What?

Switching the CPU to another process/thread by saving the state of the old process/thread and loading the saved state for the new process/thread

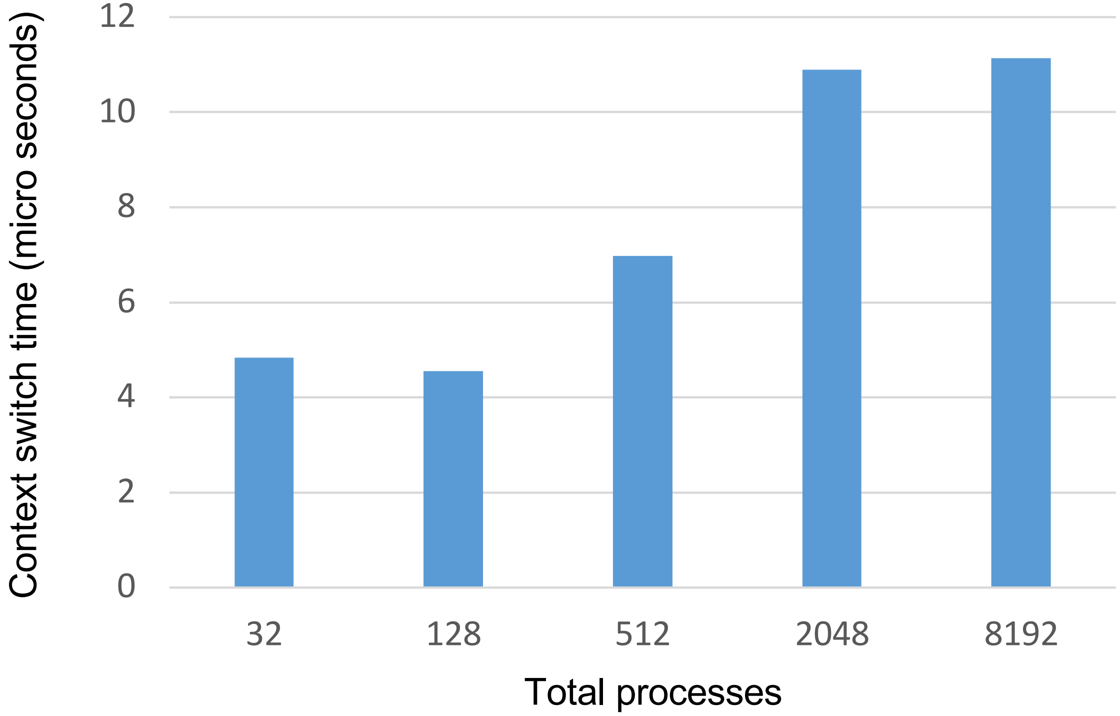


Context Switch: How?



- Timer interrupt moves the process/thread execution from user stack to its kernel stack (each user process/thread has its own kernel stack)
- User context is saved on the bottom of the kernel stack and interrupt handler is invoked
- The interrupt handler will call yield() method
- The **yield** will invoke a call to the CPU scheduler, which will: a) save the kernel context of the user process at the top of the kernel stack, b) load the context of the CPU scheduler, and c) jump to the CPU scheduler stack where it will call the schedule() routine of the OS
- Each CPU has its own CPU scheduler process that runs on a separate kernel stack than that of the process/thread it was originally executing

Context Switch: Cost?

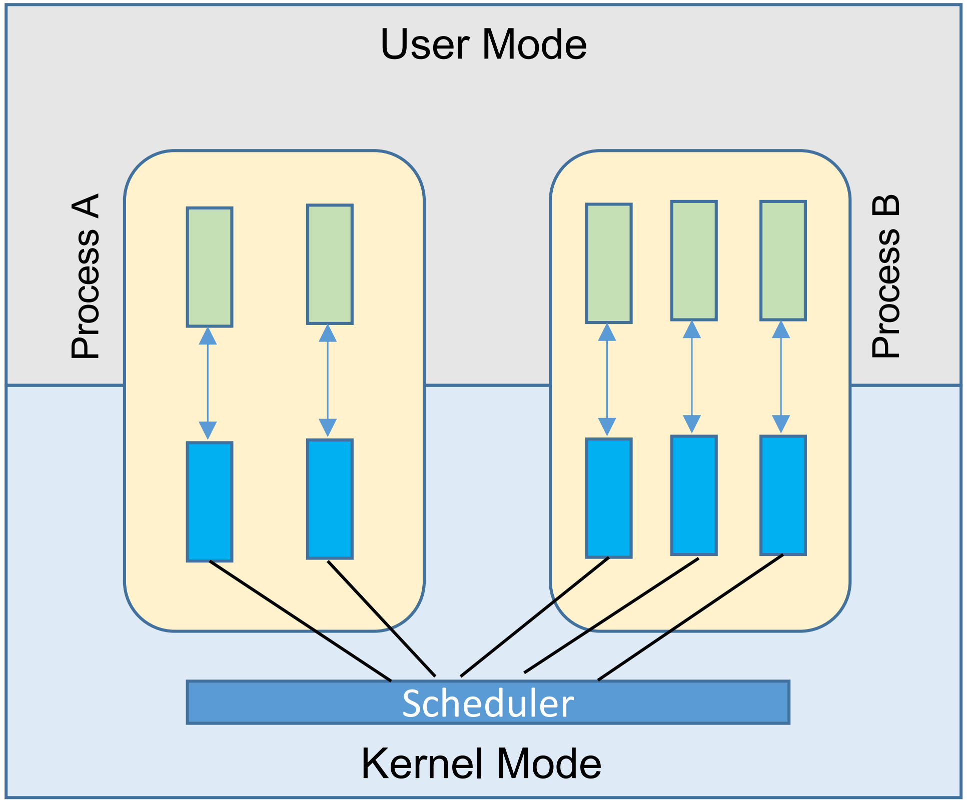


- Context switch overhead measured on an AMD EPYC 32-core processor running Ubuntu 18.04.3 LTS
  - Data generated using lmbench benchmark (./lat_ctx –s 0 32 128 512 2048 8192)
- Overheads
  - Timer interrupt latency
  - Saving restoring context
  - Process scheduling
  - Reloading TLB
  - Loss in cache locality

Threading Model

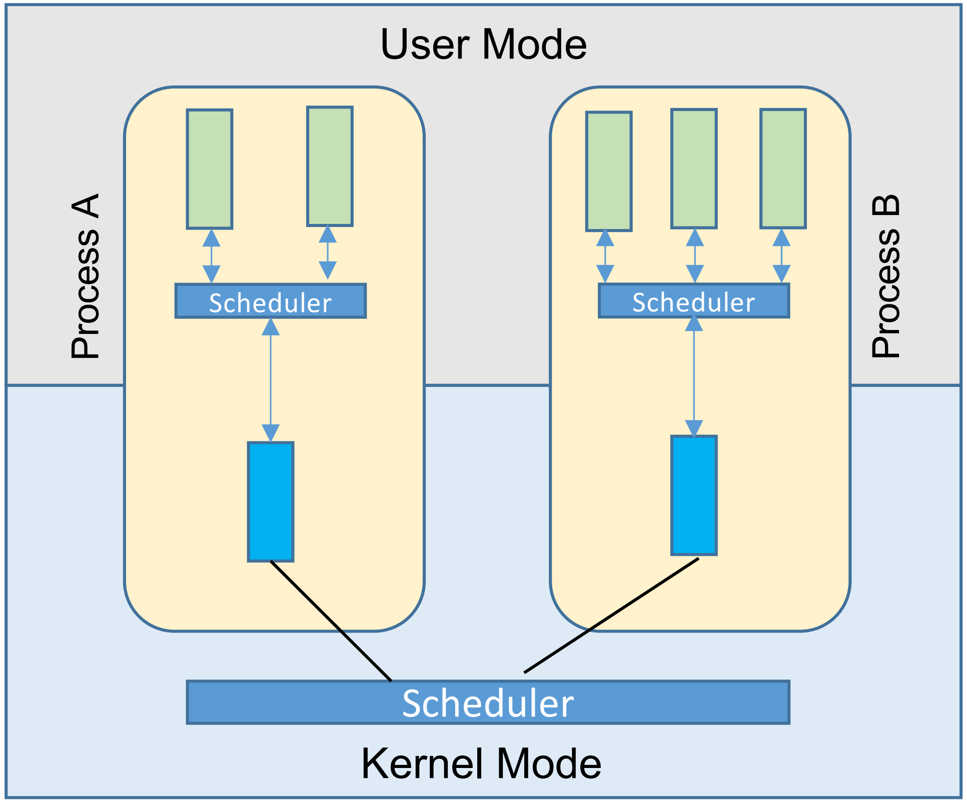
- **1x1** threading Model (Kernel Level Threads)
- **MxN** threading model (User Level Threads)

1x1 Threading Model



- Every thread created by the user has 1x1 mapping with the kernel thread
  - E.g., pthread library on Linux
- OS manages all thread operations
  - Heavyweight operations
    - Thread creation
    - Context switches
  - Scheduling policy solely managed by the kernel

MxN Threading Model



- User gets to create several threads, but each of these threads can be mapped to a single kernel level thread
  - Some JVMs have been doing it
- Runtime library (in user space) manages all thread operations
  - Lightweight operations (OS is totally unaware of user level thread operations)
    - Thread creation
    - Context switches
  - Flexible scheduling policies can be implemented

Today's Class

- Threading models
- Boost C++ libraries for concurrency
  - ➡ Context

Is it Possible in Plain C/C++?

```
void A() {
  cout<< “IN-A” << endl;
  /* Do something */
  cout<< “OUT-A” << endl;
}
void B() {
  cout<< “IN-B” << endl;
  /* Do something */
  cout<< “OUT-B” << endl;
}
void C() {
  cout<< “IN-C” << endl;
  /* Do something */
  cout<< “OUT-C” << endl;
}
int main() {
  A();
  B();
  C();
}
```

Figure-1

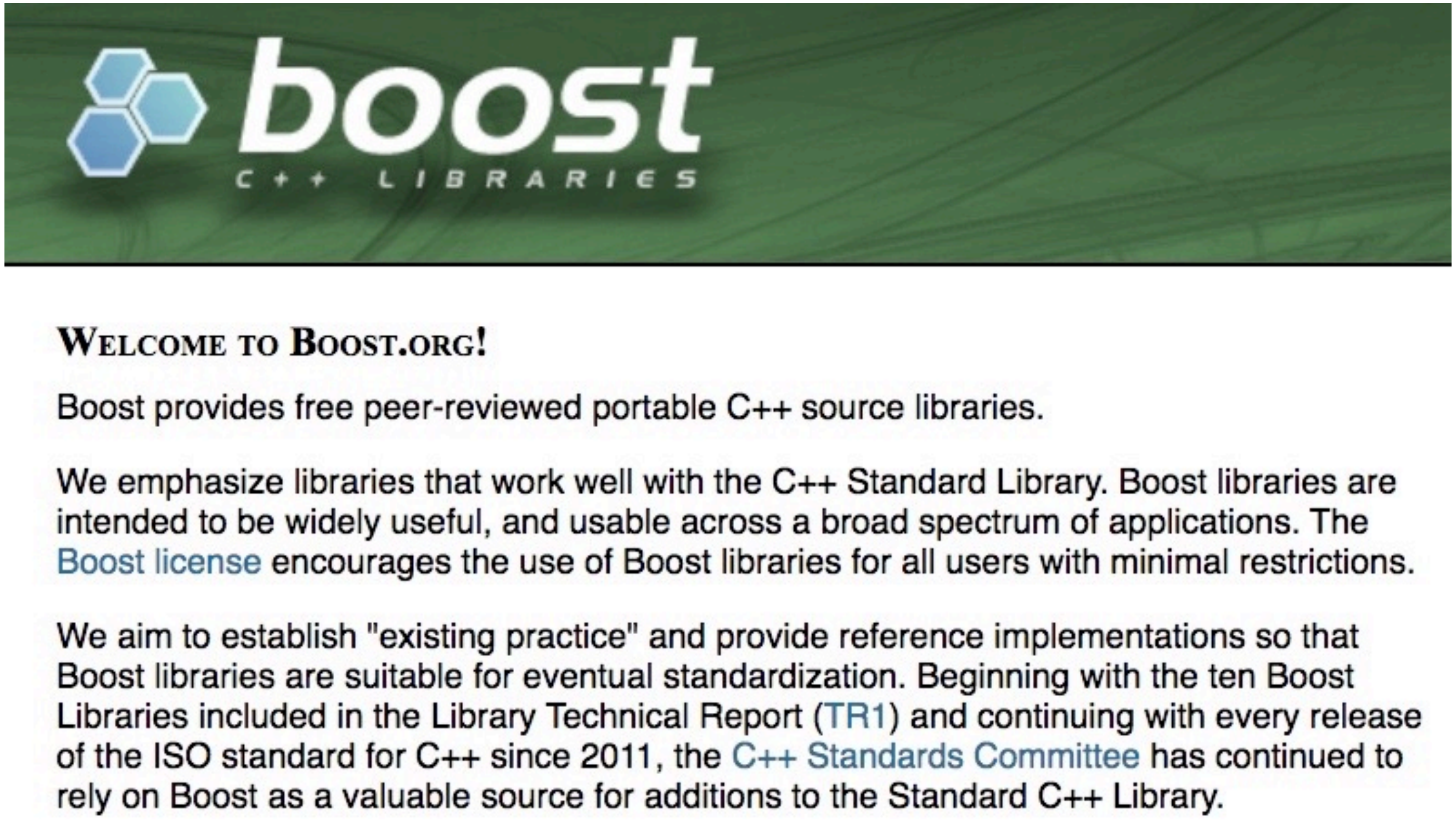
- Output in Figure-1?

IN-A
OUT-A
IN-B
OUT-B
IN-C
OUT-C

- How to get this?

IN-A
IN-B
IN-C
OUT-A
OUT-B
OUT-C

Boost C++ Libraries



Welcome to Boost.org!

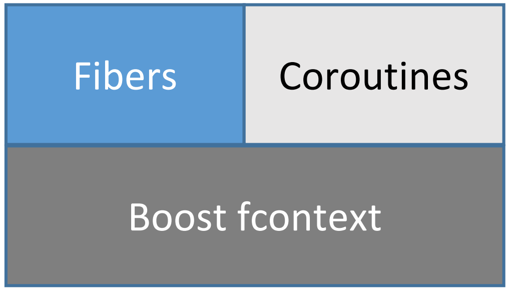
Boost provides free peer-reviewed portable C++ source libraries.

We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages the use of Boost libraries for all users with minimal restrictions.

We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Beginning with the ten Boost Libraries included in the Library Technical Report (TR1) and continuing with every release of the ISO standard for C++ since 2011, the C++ Standards Committee has continued to rely on Boost as a valuable source for additions to the Standard C++ Library.

Boost Context Library

- Provides a sort of cooperative multitasking on a single thread
- By providing an abstraction of the current execution state in the current thread, a *fcontext_t* instance represents a specific point in the application's execution path
  - stack (with local variables)
  - stack pointer
  - all registers and CPU flags
  - instruction pointer
- Provides the means to suspend the current execution path and to transfer execution control, thereby permitting another *fcontext_t* to run on the current thread
  - Helps in extremely low latency context switching of execution inside userspace (around 19 CPU cycles on x86_64 platform [1])
- Disadvantage
  - Not supported on all platforms as based on assembly code



Documentation: https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/index.html

[1] https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/context/performance.html#performance

Boost Context: Only Two Low-level Core APIs

- Create new context

```
fcontext_t make_context( /* pointer to top of new stack */,
                         /* size of the new stack */,
                         /* function to call when starting new context */);
```

- Jump to new context

```
void* jump_fcontext( /*current context */,
                     /* new context */,
                     /* some more arguments … */);
```

Boost Context C++11 Library

- Two primary operations
  - callcc
    - Call with current **continuation**
    - Captures current continuation and triggers a context switch
  - Resuming a saved continuation
    - resume()
      - Can be used to switch across different continuations

```
1. void foo() {
2.   S1;
3.    //continuation of S1 (L4+)
4.    S2;
5.    //continuation of S2 (L6+)
6.   S3;
7. }
```

Boost Context C++11 Library: Example

```
void A() {
  cout<< “IN-A” << endl;
  /* Do something */
  cout<< “OUT-A” << endl;
}
void B() {
  cout<< “IN-B” << endl;
  /* Do something */
  cout<< “OUT-B” << endl;
}
void C() {
  cout<< “IN-C” << endl;
  /* Do something */
  cout<< “OUT-C” << endl;
}
int main() {
  A();
  B();
  C();
}
```

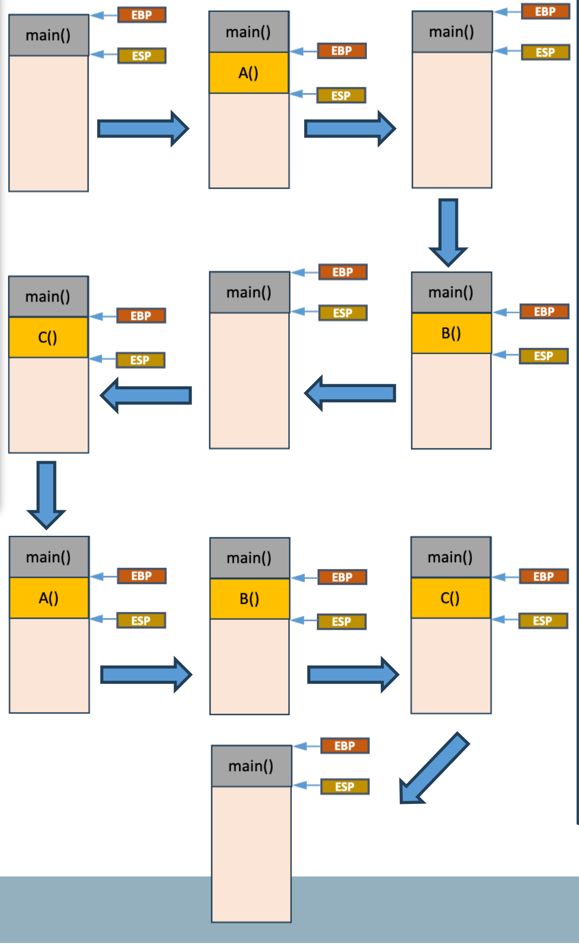
Figure-1

```
#include <boost/context/all.hpp>
ctx::continuation A(ctx::continuation cont) {
  cout<< “IN-A” << endl;
  cont = cont.resume();
  /* Do something */
  cout<< “OUT-A” << endl;
  return std::move(cont);
}
/* Methods B & C rewritten as A above */
int main() {
  ctx::continuation a = ctx::callcc(A);
  ctx::continuation b = ctx::callcc(B);
  ctx::continuation c = ctx::callcc(C);
  a.resume();
  b.resume();
  c.resume();
}
```

Figure-2

Used to switch across different continuations

Call with current continuation. Captures current continuation and triggers a context switch



● Figure-1

IN-A
OUT-A
IN-B
OUT-B
IN-C
OUT-C

● Figure-2

IN-A
IN-B
IN-C
OUT-A
OUT-B
OUT-C

Another Example, Output?

```
int main() {
  int counter = 1;
  ctx::continuation c = ctx::callcc([&](ctx::continuation && c) {
    counter += 1;
    c = c.resume();
    counter -= 1;
    return std::move(c);
  });

  counter += 1;
  c = c.resume();
  std::cout <<”Counter = “ << counter;
  return 0;
}
```

Handling Blocking Task in plain C++

```
/* User application (C++11) */
.......
void main() {
  future_t* future = async([=]() {
      foo();
  });
  future.get();
}
```

```
/* Runtime implementation */
get() {
  if(this_future->not_ready()) {
    /* create/save current continuation and context
       switch to do something else */
  }
  else return this_future->value;
}
```

- Solution
  - Use boost context library (C++11) to avoid thread creation for any situation where the current thread cannot proceed due to blocking condition
  - Save the current context (continuation) and swap it with another ready context on the same thread

Installing Boost Context and Fiber Library

- Install Boost
  - `wget https://boostorg.jfrog.io/artifactory/main/release/1.80.0/source/boost_1_80_0.tar.gz`
  - `tar xvfz boost_1_80_0.tar.gz`
  - `cd ~/boost_1_80_0/`
  - `./bootstrap.sh --prefix=/absolute/path/to/boost-install --with-libraries=fiber,context`
  - `./b2 install`
- Compile programs
  - `g++ -O3 -I/absolute/path/to/boost-install/include -L/absolute/path/to/boost-install/lib Program.cpp -lboost_fiber -lboost_context -lpthread`
- Execute programs
  - `export LD_LIBRARY_PATH=/absolute/path/to/boost-install/lib:$LD_LIBRARY_PATH`
  - `./a.out`

Reading Materials

- Context
  - https://www.boost.org/doc/libs/1_80_0/libs/context/doc/html/index.html

Next Lecture (#12)

- User level threads
- Quiz-3
  - Syllabus Lecture 08-11