Lab 2

Help Session

Shailja Pandey
Shubhani

Aim

- Assign1 -- Implemented a shell with
  - Long computation task as subroutine
  - Echo
  - Number of Key Pressed
- Assign2
  - Stackless Coroutines
  - Fibers
  - Non Preemptive or Cooperative Scheduling
  - Preemptive Scheduling

Subroutines

- Subroutines are spl cases of coroutines.
- When invoked, execution begins at the start and once a subroutine exits, it is finished.
- An Instance of a subroutine only returns(yields) once and doesn’t hold/save state between invocations.

Coroutines/ Cooperative tasks/ Resumable functions

- Coroutines are computer program that allow multiple entry points for suspending and resuming execution.
- The values of data local to a coroutine persist between successive invocations.
- The execution of a coroutine is suspended as control leaves it and resumption of that coroutine starts from where it left off.

Classification

- **Symmetric coroutine** -- control-transfer operation allows coroutines to explicitly pass control among themselves.

- **Asymmetric coroutine** -- two control-transfer operations: one for invoking a coroutine and one for suspending it, the latter returning control to the coroutine invoker.

Classification

- Stackless
  - heap-allocated data structure to contain arguments and local variables for the coroutine
  - Scalable
  - Fast Context Switch

- Stackfull/Fiber
  - giving to each coroutine its own stack
  - Allows Nested coroutine calls

Labs

2.1 -- Asymmetric Stackless Coroutine

2.2 -- Asymmetric Stacklfull Coroutine -- Fiber

2.1 -- Asymmetric Stackless Coroutine

- We don't have native C/C++ language support yet for coroutine
- Libraries like Boost.Coroutine, CO2 etc to support

- We have built a custom coroutine library -- ***util/coroutine.h***

- **f_t** -- Structure to store values of "data local to a coroutine between successive calls"
- **coroutine_t** -- store PC from where the execution has to resume
- **coroutine_reset()** -- Intialize PC=0 inside coroutine_t structure.
- **h_begin()** -- Control transfer to saved PC
- **h_yield()** -- stores PC of next instruction in coroutine_t and returns.
- **h_end()** -- resets the value of PC to zero and infinitely call yield.

Example

- 3*3 Matrix Generation

*[1*1, 1*2, 1*3*

*2*1, 2*2, 2*3*

*3*1, 3*2, 3*3]*

```
for(i=1;i<=3;i++){
  for(j=1;j<=3;j++){
     ret=i*j; done=false;  }   }
done = true;
```

Example

```
// state of function f to be preserved across multiple calls.
//
struct f_t{
 int i;
 int j;
};
```

```
void f(coroutine_t* pf_coro,f_t* pf_locals,int* pret,bool* pdone){
  coroutine_t& f_coro = *pf_coro; // boilerplate: to ease the transit
  int& ret             = *pret;
  bool& done           = *pdone;

  int& i               = pf_locals->i;
  int& j               = pf_locals->j;

  h_begin(f_coro);

  for(i=1;i<=3;i++){
    for(j=1;j<=3;j++){
      ret=i*j; done=false; h_yield(f_coro); // yield (i*j, false)
    }
  }

  ret=0; done=true; h_end(f_coro); // yield (0,true)
}
```

Example

```
coroutine_t f_coro;
coroutine_reset(f_coro);
f_t f_locals;

f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=1*1  f_done=false
f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=1*2  f_done=false
f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=1*3  f_done=false
f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=2*1  f_done=false
f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=2*2  f_done=false
...
f(f_coro,f_locals,shell.f_ret,shell.f_done); //post cond: f_ret=0    f_done=true
```

2.2 -- Fiber

- Implement a stack for each coroutine, and let local variables stored on stack instead of a data structure.
- ***Results in 2 stacks when fiber is running -- main_stack, f_stack***
- We have built a custom fiber library -- ***util/fiber.h***
- **stack_initN**(f_stack, f_array, f_arraysize, f_start, f_args…): creates a function stack at beginning of fiber and pushes variable number of arguments(N in this case)
- **stack_saverestore**(from_stack,to_stack) : saves the context to **from_stack**, restore the context from **to_stack**.

2.2 -- Fiber

GCC Extended Asm

- To read and write C variables from assembly and to perform jumps from assembler code to C labels.
- Extended asm syntax uses colons (‘:’) to delimit the operand parameters after the assembler template.
- asm [volatile] (
  AssemblerTemplate
  : *OutputOperands*
  [: *InputOperands* [: *Clobbers*]])

2.2 -- Fiber

GCC Extended Asm

- To read and write C variables from assembly and to perform jumps from assembler code to C labels.

- Extended asm syntax uses colons (‘:’) to delimit the operand parameters after the assembler template.

- asm [volatile] (
  AssemblerTemplate
  : *OutputOperands*
  [: *InputOperands* [: *Clobbers*]])

2.2 -- Fiber

GCC Extended Asm

- ***Outputvariables***
  - the names of C variables modified by the assembly
  - *asmSymbolicName*
    - position of the operand in the list of operands in the assembler template.
  - *Constraint*
    - must begin with either ‘=’ (a variable overwriting an existing value) or ‘+’ (when reading and writing)
    - describe where the value resides.
    - ‘r’ for register and ‘m’ for memory.

2.2 -- Fiber

GCC Extended Asm

- ***inputvariables***
  - C variables and expressions available to the assembly code
  - *asmSymbolicName*
    - position of the operand in the list of operands in the assembler template.
  - *Constraint*
    - describe where the value resides.
    - ‘r’ for register and ‘m’ for memory.

2.2 -- Fiber

GCC Extended Asm

- ***Clobbers***
  - calculations may require additional registers,
  - or the processor may overwrite a register as a side effect of a particular assembler instruction.
  - In order to inform the compiler of these changes, list them in the clobber list.

2.2 -- Fiber

util/fiber.h has MACRO written in GCC Extended Asm

```
#define stack_inithelper(_teip)  do{
 asm volatile(
   "  movl $1f,%0      \n\t"
   "  jmp  2f          \n\t"
   "1:                 \n\t"
   "  movl $0, %%ebp   \n\t"
   "  jmp *(%%esp)    \n\t"
   "2:                 \n\t"
  :"=m" (_teip)
  :
 );
}while(false)
```

```
#define stack_init2(f_stack,f_array,f_arraysize,f_start,f_arg1,f_arg2) do{
 uintptr_t teip;                                                        \
 stack_inithelper(teip);                                                \
 addr_t  stack=addr_t(f_array)+f_arraysize;                             \
 stack=stack_push(stack,f_arg2);                                        \
 stack=stack_push(stack,f_arg1);                                        \
 stack=stack_push(stack,f_start);                                       \
 stack=stack_push(stack,teip);                                          \
 f_stack=stack;                                                         \
}while(false)
```

2.2 -- Fiber

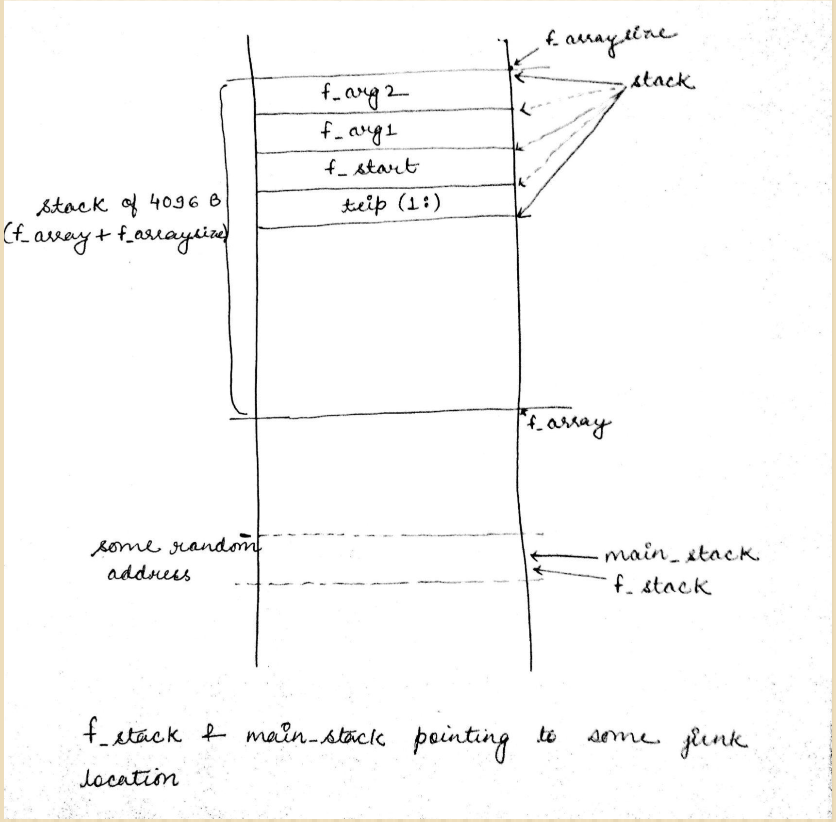
util/fiber.h has MACRO written in GCC Extended Asm

```
#define stack_saverestore(from_stack,to_stack) do {
 asm volatile(
   "  pushl %%eax        \n\t"
   "  pushl %%ecx        \n\t"
   "  pushl %%ebp        \n\t"
   "  pushl $1f          \n\t"
   "                     \n\t"
   "  movl  %%esp,(%0)   \n\t"
   "  movl  (%1),%%esp   \n\t"
   "                     \n\t"
   "  ret                \n\t"
   "1:                   \n\t"
   "  popl %%ebp         \n\t"
   "  popl %%ecx         \n\t"
   "  popl %%eax         \n\t"
  :
  :"a" (&from_stack), "c"  (&to_stack)
  :_ALL_REGISTERS, "memory"
 );
} while(false)
```

2.2 -- Fiber

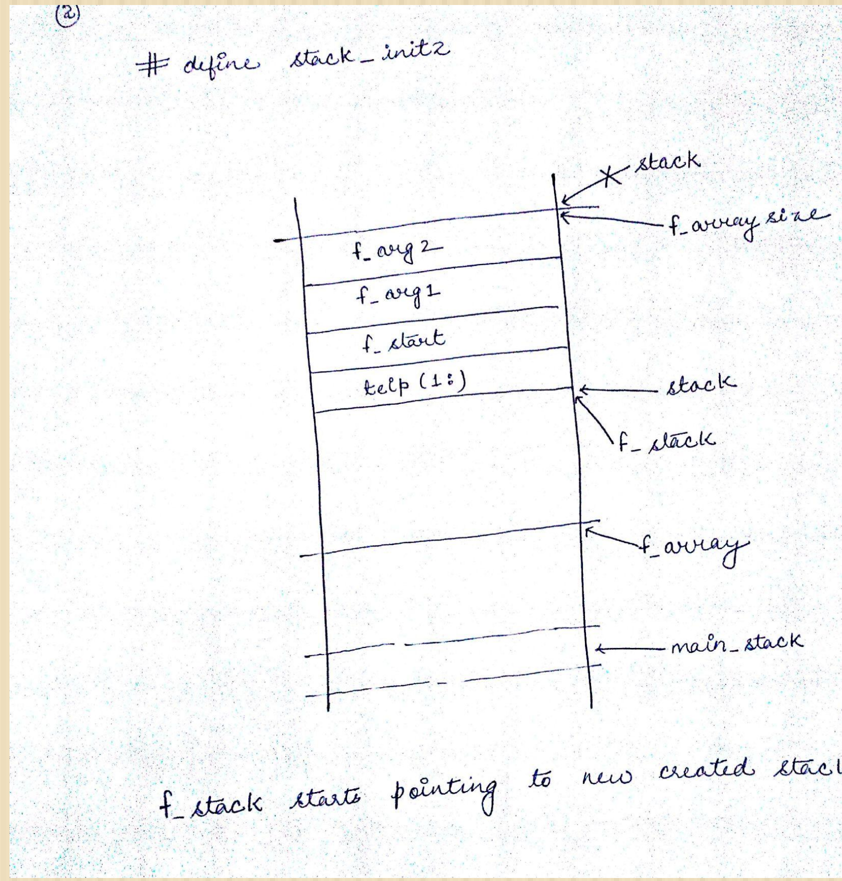
```
#define stack_inithelper(_teip)  do{
 asm volatile(
   "  movl $1f,%0      \n\t"
   "  jmp  2f          \n\t"
   "1:                 \n\t"
   "  movl $0, %%ebp   \n\t"
   "  jmp *(%%esp)    \n\t"
   "2:                 \n\t"
  :"=m" (_teip)
  :
 );
}while(false)
```

```
#define stack_init2(f_stack,f_array,f_array
 uintptr_t teip;
 stack_inithelper(teip);
 addr_t  stack=addr_t(f_array)+f_arraysize;
 stack=stack_push(stack,f_arg2);
 stack=stack_push(stack,f_arg1);
 stack=stack_push(stack,f_start);
 stack=stack_push(stack,teip);
 f_stack=stack;
}while(false)
```



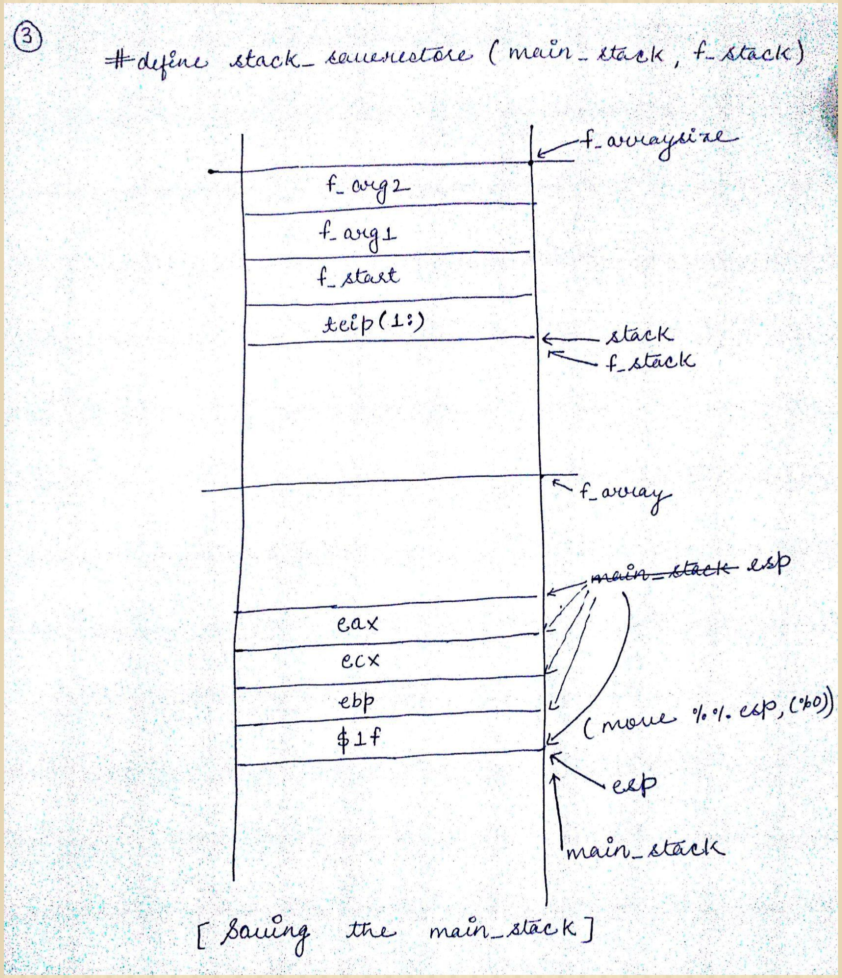
2.2 -- Fiber

```
#define stack_init2(f_stack,f_array,f_array
 uintptr_t teip;
 stack_inithelper(teip);
 addr_t  stack=addr_t(f_array)+f_arraysize;
 stack=stack_push(stack,f_arg2);
 stack=stack_push(stack,f_arg1);
 stack=stack_push(stack,f_start);
 stack=stack_push(stack,teip);
 f_stack=stack;
}while(false)
```



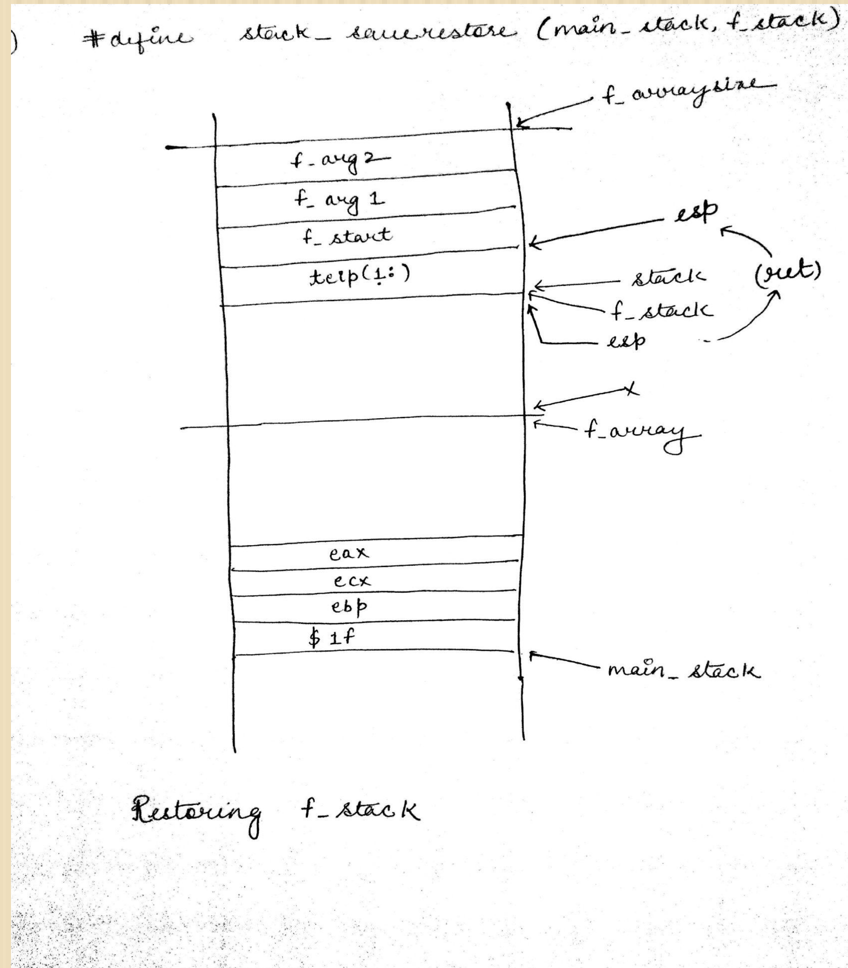
2.2 -- Fiber

```
#define stack_saverestore(from_stack,to_stack)
 asm volatile(
   "  pushl %%eax       \n\t"
   "  pushl %%ecx       \n\t"
   "  pushl %%ebp       \n\t"
   "  pushl $1f         \n\t"
   "                    \n\t"
   "  movl  %%esp,(%0)  \n\t"
   "  movl  (%1),%%esp  \n\t"
   "                    \n\t"
   "  ret               \n\t"
   "1:                  \n\t"
   "  popl %%ebp        \n\t"
   "  popl %%ecx        \n\t"
   "  popl %%eax        \n\t"
  :
  :"a" (&from_stack), "c"  (&to_stack)
  :_ALL_REGISTERS, "memory"
 );
} while(false)
```

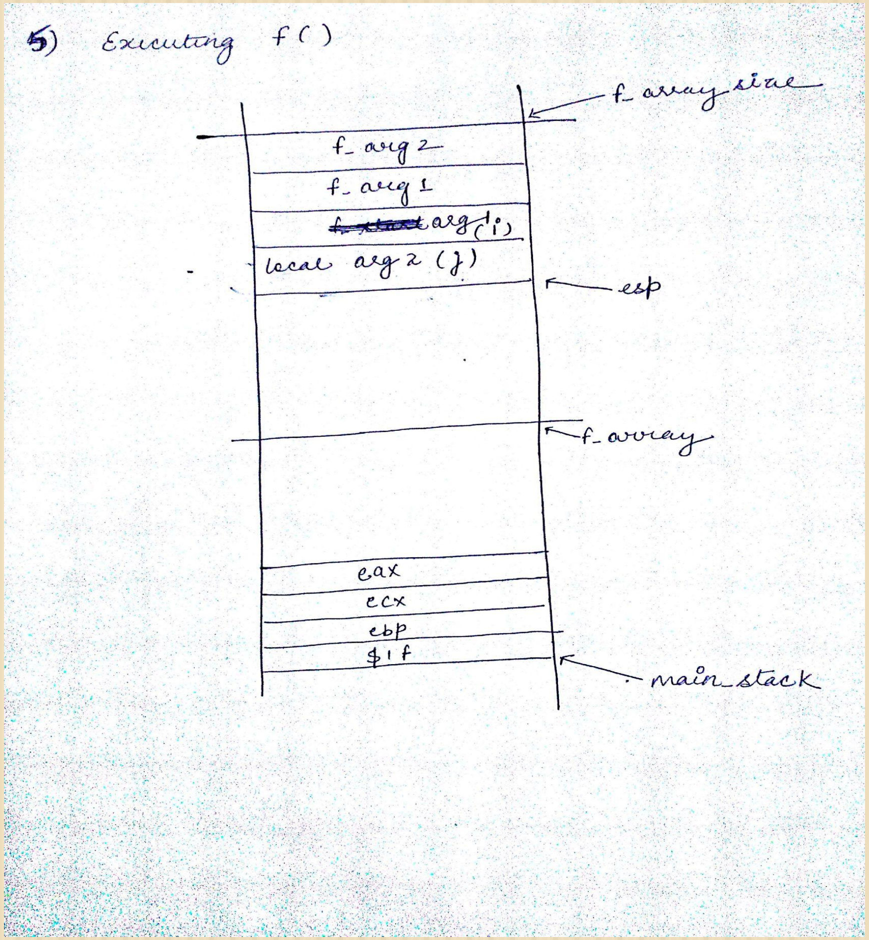


2.2 -- Fiber

```
#define stack_saverestore(from_stack,to_stack)
 asm volatile(
   "  pushl %%eax       \n\t"
   "  pushl %%ecx       \n\t"
   "  pushl %%ebp       \n\t"
   "  pushl $1f         \n\t"
   "                    \n\t"
   "  movl  %%esp,(%0)  \n\t"
   "  movl  (%1),%%esp  \n\t"
   "                    \n\t"
   "  ret               \n\t"
   "1:                  \n\t"
   "  popl %%ebp        \n\t"
   "  popl %%ecx        \n\t"
   "  popl %%eax        \n\t"
  :
  :"a" (&from_stack), "c"  (&to_stack)
  :_ALL_REGISTERS, "memory"
 );
} while(false)
```



2.2 -- Fiber



5) Executing f( )
f_array_size
f_arg 2
f_arg 1
arg 1 (i)
local arg 2 (j)
esp
f_array
eax
ecx
ebp
$1f
main_stack

Example

- 3*3 Matrix Generation

  *[1*1, 1*2, 1*3*
  *2*1, 2*2, 2*3*
  *3*1, 3*2, 3*3*]

```
for(i=1;i<=3;i++){
  for(j=1;j<=3;j++){
     ret=i*j; done=false;  }   }
done = true;
```

Example

```
uint8_t f_array[F_STACKSIZE];
const size_t f_arraysize=F_STACKSIZE;

addr_t main_stack;
addr_t f_stack;

stack_reset4(f_stack, &f_array, f_arraysize, &f, &main_stack, &f_stack, &shell.f_ret, &shell.f_done);

stack_saverestore(main_stack,f_stack); //post cond: f_ret=1*1  f_done=false
stack_saverestore(main_stack,f_stack); //post cond: f_ret=1*2  f_done=false
stack_saverestore(main_stack,f_stack); //post cond: f_ret=1*3  f_done=false
stack_saverestore(main_stack,f_stack); //post cond: f_ret=2*1  f_done=false
stack_saverestore(main_stack,f_stack); //post cond: f_ret=2*2  f_done=false
...
stack_saverestore(main_stack,f_stack); //post cond: f_ret=0    f_done=true
```

Example

```
void f(addr_t* pmain_stack, addr_t* pf_stack, int* pret, bool* pdone){
  addr_t& main_stack = *pmain_stack; // boilerplate: to ease the transi
  addr_t& f_stack    = *pf_stack;
  int& ret           = *pret;
  bool& done         = *pdone;

  int i;
  int j;

  for(i=1;i<=3;i++){
    for(j=1;j<=3;j++){
      ret=i*j;done=false; stack_saverestore(f_stack,main_stack);
    }
  }
  for(;;){
    ret=0;done=true; stack_saverestore(f_stack,main_stack);
  }
}
```

Preemptive vs Non-preemptive Scheduling

- In preemptive scheduling,
  - the running task is interrupted by scheduler for some time.
  - the control is transferred to some other task.
  - the previously running task may be resumed at some later point in time depending upon the scheduling algorithm.

- In non-preemptive scheduling,
  - a running task is executed till completion. It cannot be interrupted by the scheduler.
  - control can be transferred to other tasks by the scheduler only when the currently running task voluntarily releases(yeilds) the control to the shell.

2.3 -- Non-preemptive scheduling

- So far, we have the capability to run only one fiber
- enhancing our shell to support multiple pending long computation task.
- You shall support atleast two additional long computation tasks as fibers (Retain previous menu items).

- For these additional long computation tasks:
  - Same command/menu item may be entered multiple times, but at max 3 times.
  - Total number of fibers in progress shall be limited to maximum of 5

Non-preemptive scheduling

- G:: GArg -> GResult   H.:: HArg -> HResult
- We also want to support multiple invocations of these fibers. (atmax 3).
- total number of instances for G and H should be <= 5.
- have to store 3*(GArg,GResult) and 3*(HArg,HResult) in shellstate_t..
- What should be a good data structure for storing these?
  - 3*(GArg,GResult) and 3*(HArg,HResult)
  - 5* Union of (GArg,GResult) and (HArg,HResult)

Non-preemptive scheduling

- G:: GArg -> GResult H.:: HArg -> HResult
- We also want to support multiple invocations of these fibers. (atmax 3)
- total number of instances for G and H should be <= 5.
- have to store 3*(GArg,GResult) and 3*(HArg,HResult) in shellstate_t..

Non-preemptive scheduling

- How to do scheduling?
- Let's say, we have a circular buffer/linked list of pending tasks
- When someone wanted to start a task, just check the resource limitations.
- If available, change state and add into the queue.
- When current running fiber yeilds, invoke fiber_scheduler
- In each invocation of fiber_scheduler, just pick one fiber, and execute.
- In next invocation - pick the next fiber and execute it.. so on.

Think of our own scheduling
(e.g. round robin)

2.4 -- Preemptive scheduling

- To achieve responsiveness, we added yield points explicitly in 2.2.
- To achieve better responsiveness -- pre-emptive scheduling
- Pre-emption requires support for timer interrupts, which means we need to write interrupt handlers and program Interrupt Descriptor Tables(IDTs).

Preemptive scheduling

- Timer : devices/lapic.h
  - one-shot LAPIC timer to raise an interrupt after a specified time
- LAPIC
  - **Local Advanced Programmable Interrupt Controller**
  - It is hardwired to each CPU core
  - Software sets a "initial count“
  - The local APIC decrements the count until it reaches zero, then generates a timer IRQ

Preemptive scheduling

- LAPIC Timer Modes
  - **Periodic Mode**
    - resets the current count to the initial count when the current count reaches zero
    - begins decrementing the current count again
  - **One-Shot Mode**
    - it doesn't reset the current count to the initial count when the current count reaches zero.
    - Software has to set a new count each time if it wants more timer IRQs.
- Dynamic timers -- If there's no fibers running, there shouldn't be any timers firing

Preemptive scheduling

- labs/fiber.cc and labs/fiber_scheduler.cc -- set the timer.
  - **dev_lapic_t** object
  - reset_timer_count(int count)

- Decide the timer interval wisely.

Preemptive scheduling

- Interrupt handler : labs/preempt.h
  - ring0_preempt
  - To be written in assembly
  - should switch ‘funct_stack' to ‘main_stack’
- Interrupt Descriptor Table(IDT) : x86/except.cc
- Reuse shell_step_fiber_scheduler(2.3) to do the scheduling

Preemptive scheduling

- Similar to stack_saverestore
- It shall save and restore FPU/SIMD registers (context) as well during the context switch

```
#define _ring0_preempt(_name,_f)

_name:
      call C function: _f

      // begin
      if thread is already inside yield,
        jmp iret_toring0

      save the CPU state to core_t.preempt.foo
      switch stack
      restore CPU state from core_t.main_stack
      // end

      jmp iret_toring0
```

Preemptive scheduling

- Out of two additional fibers implemented during fiber_scheduler:
  - One of the fiber should be running normally with non-preemptive yields (stack_saverestore) (This is to trigger race condition between yield and ring0_preempt)
  - another fiber shall be modified to execute without yields in between the computation (This is to check preemption is working or not)