Assignment-1

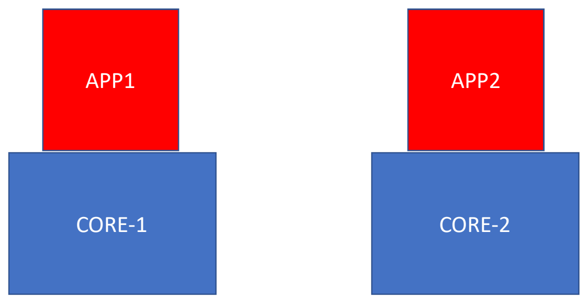
- git clone https://github.com/Systems-IIITD/SimpleMM.git
- "make" to build the SimpleMM library and the test case
  - generate libmemory.so and an executable random
  - execute ./random to run the test case
- "make run" to run the test case

OS

- OS allows multiple applications to execute at the same time

- OS enforces isolation among applications
  - will discuss later

Multiple applications



Multiple applications

- Can we share the stack among applications?
- Can we share registers among applications?
- Can we share the heap among applications?
- Can we share global variables among applications?

Can we share the stack among applications?

application:1

```
int foo(int a, int b) {
  a = a + b;
  return a;
}
```

application:2

```
int bar(int a) {
  a = a + a;
  return a;
}
```

We can’t share stack among applications, because applications may have different values of local variables, parameters, etc., which are at the same offset in the stack.

Can we share registers among applications?

application:1

```
int foo(int a, int b) {
  a = a + b;
  return a;
}
```

application:2

```
int bar(int a) {
  a = a + a;
  return a;
}
```

No, because the local variables can also live in the same registers in both the applications.

Can we share the heap among applications?

application:1

```
int *foo(int a, int b) {
  int *c = malloc(4);
  c[0] = a + b;
  return c;
}
```

application:2

```
int *bar(int a) {
   int *c = malloc(4);
   c[0] = a + a;
   return c;
}
```

Yes, because each invocation of malloc returns a unique address. If both the applications are calling malloc, then they will get different values.

Can we share global variables among applications?

application:1

```
int c;
int foo(int a, int b) {
  c = a + b;
  return c;
}
```

application:2

```
int c;
int bar(int a) {
   c = a + a;
   return c;
}
```

Yes, because the compiler allocates space for global variables in the program image that is loaded by the OS at different RAM addresses.

Thread

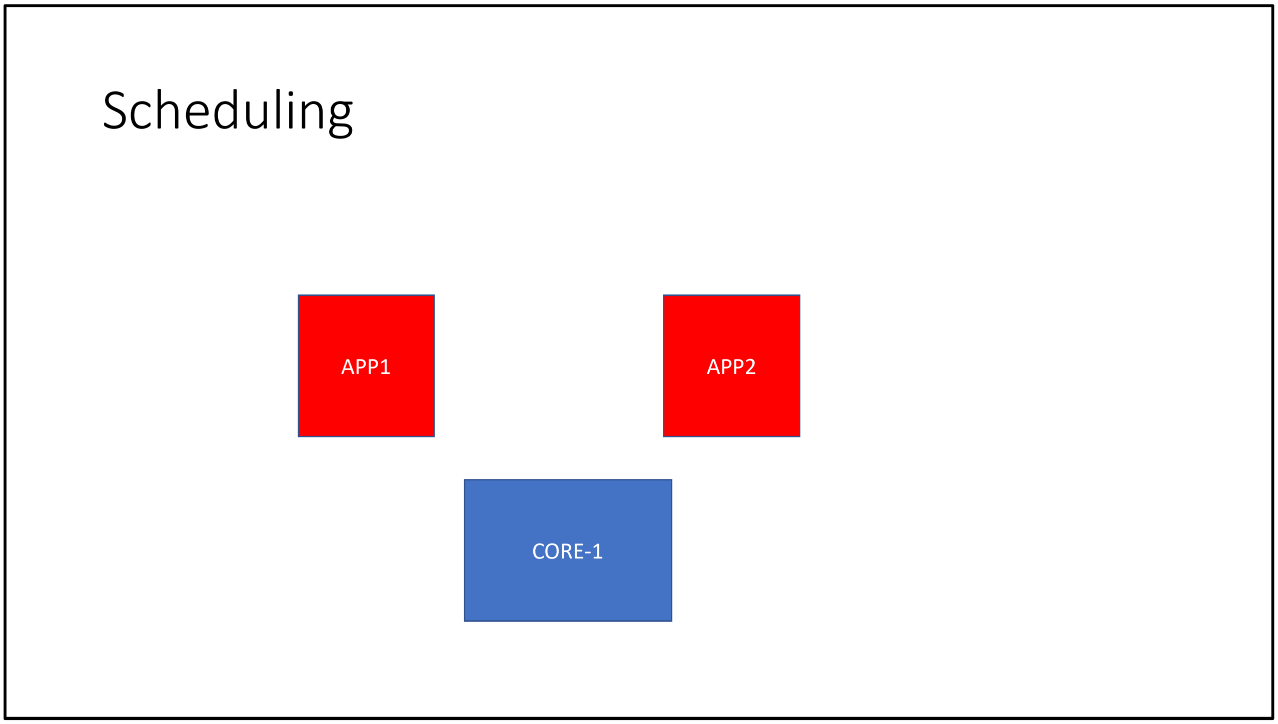
- An application is also called a thread
- Threads can share heap and global variables
- Threads have private stack and registers

Uniprocessor

- First, we will discuss how things work on a uniprocessor system
- Multiprocessor will be discussed later

Scheduling

- How does OS run multiple applications at the same time on a uniprocessor system



Only one application runs at a given time. If there are more applications than the number of CPUs, then OS multiplexes the CPU among the applications.

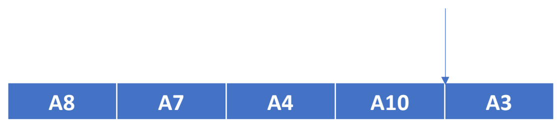
Scheduling

- How does OS run multiple applications at the same time on a uniprocessor system
  - At a given time only one thread executes
  - OS multiplexes the CPU between applications but the multiplexing is so fast (e.g., after every 10 ms) that user doesn't notice the pause

Scheduling

- The job of the scheduler to multiplex the CPU among threads
- The scheduler maintains a queue (ready queue) that contains all threads that need to be scheduled
- Whenever the scheduler is called, it inserts the current thread to the ready queue and removes a thread to the ready queue and starts its execution

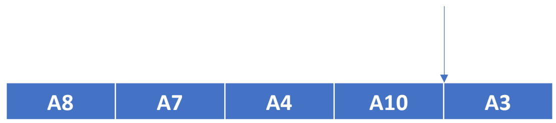
Scheduling



A1 is running

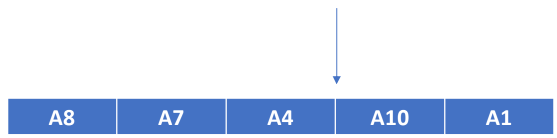
In this example, the OS scheduler is maintaining a FIFO queue (also called ready queue), that contains all applications that need CPU. When the scheduler is invoked, it puts the current application to queue and schedules an application that is next in the FIFO order. This is a very simple scheduling strategy. The actual scheduling implemented by the OS can be very complicated.

Scheduling



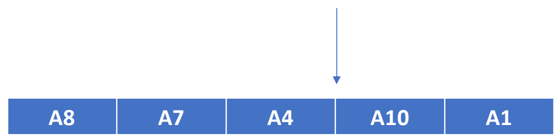
A1 is running
A1 calls scheduler
A3 was picked for execution

Scheduling



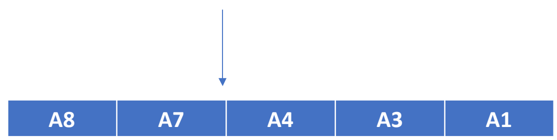
A1 is running
A1 calls scheduler
A3 was picked for execution
A3 is executing

Scheduling



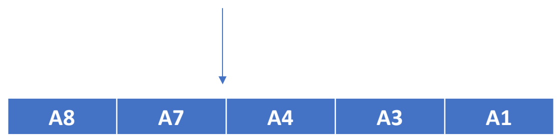
A1 is running
A1 calls scheduler
A3 was picked for execution
A3 is executing
A3 calls scheduler

Scheduling



A1 is running
A1 calls scheduler
A3 was picked for execution
A3 is executing
A3 calls scheduler
A10 was picked for scheduling

Scheduling



A1 is running
A1 calls scheduler
A3 was picked for execution
A3 is executing
A3 calls scheduler
A10 was picked for scheduling
A10 is executing

Who calls the scheduler?

- Applications can use thread_yield API to call the scheduler

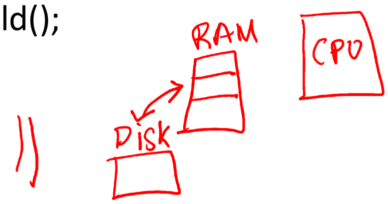
thread_yield

application:1

```
while (!work_to_do()) {
  thread_yield();
}
```

application:2

```
write_to_disk();
while (pending_io()) {
   thread_yield();
}
```



Sometimes, applications know when to call thread_yield. For example, suppose an application wants to write a buffer to the disk. Here the writing to disk is done by the disk device that is a different device. The writing to disk doesn't require CPU. The writing takes significant time (order of ms) because the disk is a very slow device. Instead of wasting CPU cycles while waiting for writing to finish, a well-behaved application may yield CPU to other threads.

thread_yield

Thread: 1

```
for (i = 0; i < n; i++) {
  sum += square(i);
}
```

Thread: 2

```
do_sort(arr);
```

Most of the time, applications don't know when to yield. For example, a compute-intensive application needs CPU all the time. For these applications, the OS has to forcefully take the CPU after some time interval to schedule a new thread.

Who calls the scheduler?

- Threads often don’t know when to yield

- Even if threads know when to yield, an OS can’t trust threads
  - e.g., a malicious thread may never yield and keep the CPU forever

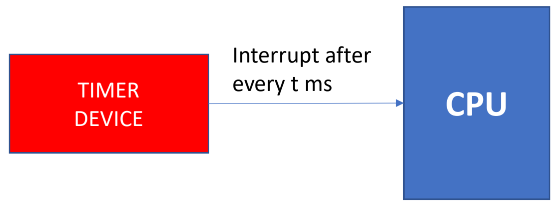
Who calls the scheduler?

- Interactive applications require scheduler to be called very often
  - e.g., after every 10ms

Examples of interactive applications are applications that are getting user attention, e.g., web browser, PowerPoint, word, etc.

Who calls the scheduler?

- A timer device is used to invoke the scheduler repeatedly, after a fixed time interval
- The timer device periodically sends interrupt to CPU



A timer device can send an interrupt to the CPU after every t seconds. Here, t can be configured to different values.

Interrupt

- On receiving an interrupt, the CPU sets the EIP to the address of the interrupt handler
- Interrupt handler is a piece of software
  - e.g., the schedule routine can be an interrupt handler
- Let us assume for now that there is some way to tell the address of the interrupt handler to the CPU

Interrupt

```
foo:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
add 16(%ebp), %eax
pop %ebp
ret
```

CPU received interrupt

```
interrupt_handler:
push %eax
push %ebx
push %esi
...
ret
```

Interrupt

foo:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
add 16(%ebp), %eax
pop %ebp
ret

CPU received interrupt

interrupt_handler:
push %eax
push %ebx
push %esi
…
ret

Interrupt

```
foo:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
add 16(%ebp), %eax
pop %ebp
ret
```

CPU received interrupt

```
interrupt_handler:
push %eax
push %ebx
push %esi
...
ret
```

On receiving an interrupt, the CPU jumps to interrupt handler (a different piece of code). An interrupt should not change the behavior of the application in any way. In other words, the program should behave in the same manner, regardless of whether it receives an interrupt or not. To understand this, let us first try to look at an interrupt handler that resumes the current application from where it was interrupted.

Interrupt

- How does interrupt handler know where to return?

Interrupt

- How does interrupt handler know where to return?
  - CPU automatically pushes the return address on the stack on interrupt

Interrupt

- Let us assume that the schedule routine is a C program that is compiled using gcc

- Can we directly use the schedule routine as the interrupt handler?
  - i.e., the hardware directly jumps to the schedule routine on interrupt

```
int a = 0;        mov $0, %eax
                          → inta
return a;         ret

                              schedule()
                              {

                              }
```

Interrupt

- Let us assume that the schedule routine is a C program that is compiled using gcc
- Can we directly use the schedule routine as the interrupt handler?
  - i.e., the hardware directly jumps to the schedule routine on interrupt
  - no, because schedule may trash CPU registers
- Which registers to save/restore in the interrupt handler?
  - Will saving/restoring caller-saved registers is enough?

An interrupt handler should not trash the value of any register because it may change the behavior of the application. If the CPU directly jumps to schedule routine on interrupt, it may trash the values of caller-saved registers.

Interrupt

- Let us assume that the schedule routine is a C program that is compiled using gcc
- Can we directly use the schedule routine as the interrupt handler?
  - i.e., the hardware directly jumps to the schedule routine on interrupt
  - no, because schedule may trash CPU registers
- Which registers to save/restore in the interrupt handler?
  - Will saving/restoring caller-saved registers is enough?
    - Yes, if interrupt handler doesn't modify callee-saved registers and just calls schedule

Interrupt

- Do we also need to save the flags?

Yes, flags are also needed to be saved. Because a routine may get interrupted between the setting and use of the EFLAG register.

Interrupt

foo:

…

cmp %ecx, %edx

ja 1f

..

1:

..

Interrupted

If an interrupt is triggered after the compare instruction, then the program may not behave correctly if the interrupt handler modifies flags.

Interrupt

- In addition to EIP, the CPU also pushes the EFLAGS on the stack
  - In fact, CPU saves more than just the EIP and EFLAGS
    - will discuss later
- iret instruction pops and restores all the values saved by the CPU on the stack

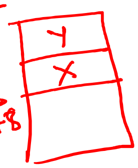
Is this a valid interrupt handler?

iret ✓

mov $0, %eax
→ int0

x = ret
y = eflags
%esp = Z

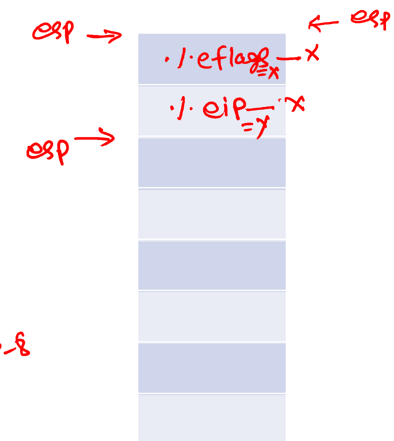
%esp = Z



%esp = Z+8

iret

iret
%esp = %esp-8
eip = x
eflags = y



Yes, this is a valid interrupt handler, because the iret instruction restores the flags and EIP pushed by the hardware and doesn't modify any other CPU state.

Is this a valid interrupt handler?

mov $0, %eax

iret

No, because the interrupted routine might be using %eax right now. Setting %eax to zero may change the behavior of the application.

Is this a valid interrupt handler?

call schedule
iret

No, because schedule routine may trash caller-saved registers.

Is this a valid interrupt handler?

push %eax
push %edx
push %ecx
call schedule
pop %ecx
pop %edx
pop %eax
iret

%eax, %edx, %ecx are caller-saved registers.
%esi, %edi, %ebx, %ebp are callee-saved registers

Yes, because even though schedule routine may trash the caller-saved registers, the interrupt handler is saving/restoring them before/after calling schedule.

Is this a valid interrupt handler?

```
push %eax
push %edx
push %ecx
call schedule
mov $0, %eax
pop %ecx
pop %edx
pop %eax
iret
```

%eax, %edx, %ecx are caller-saved registers.
%esi, %edi, %ebx, %ebp are callee-saved registers

Yes, because the interrupt handler is restoring %eax after modifying it.

Is this a valid interrupt handler?

```
push %eax
push %edx
push %ecx
call schedule
mov $0, %esi
pop %ecx
pop %edx
pop %eax
iret
```

%eax, %edx, %ecx are caller-saved registers.
%esi, %edi, %ebx, %ebp are callee-saved registers

No, because the interrupt handler is trashing the %esi register that may be used by the application after returning from the interrupt.

Interrupt handler

```
interrupt_handler:
push %eax
push %edx
push %ecx
call schedule
pop %ecx
pop %edx
pop %eax
iret
```

Let us consider this is the interrupt handler, which calls the scheduler.

scheduler

```
struct list_node *ready_list;
struct thread *cur_thread;
void schedule() {
  if (empty(ready_list))
     return;
  list_insert(ready_list, cur_thread);
  struct thread *prev = cur_thread;
  struct thread *next =  get_next_thread(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

The ready queue node corresponding to a thread is of type "struct thread"

The scheduler maintains some metadata corresponding to each thread (struct thread). In this example, the scheduler is maintaining a FIFO queue, as discussed before. The schedule routine finds the thread metadata corresponding to current and next thread and calls the context_switch routine that does the actual context switching.

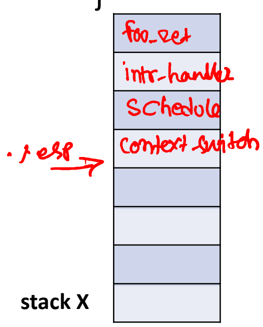
scheduler

Thread:1

foo() {

  …

}

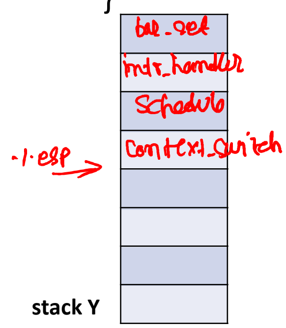


Thread:2

bar() {

  …

}



The actual context switch (switching from the previous thread to the next thread) happens in the context switch routine. Each thread that was scheduled ever will go through this context_switch routine. On entry to the context switch routine, the top of the stack contains the return address of context_switch. The context switch routine saves the stack pointer (of the previous thread) in the thread metadata (struct thread) corresponding to the previous thread. The thread metadata corresponding to the next thread contains the stack pointer of the next thread, which also contains the return address of context_switch on the top (because it was saved when last time context_switch API was called by the next thread). context_switch sets the stack pointer to the stack pointer of the next thread (from next thread's thread metadata). After returning from context_switch, the CPU executes the routines which are in the call stack of the next thread.

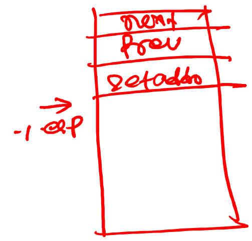
scheduler

struct thread{
void *esp;
}

context_switch(struct thread *prev, struct thread *next);

context_switch:

```
mov 4(%esp), %eax      => %eax = prev
mov %esp, (%eax)       => prev->esp = %esp
mov 8(%esp), %eax      => %eax = next
mov (%eax), %esp       => esp = next->esp
ret
```



scheduler

```
context_switch(struct thread *prev, struct thread *next);
context_switch:
mov 4(%esp), %eax
mov %esp, (%eax)
mov 8(%esp), %eax
mov (%eax), %esp
ret
```

However, there is a problem with this approach.

scheduler

Thread1:

```
int foo() {
  int a = 100;        //%esi
  return a;
}
```

Thread2:

```
int bar() {
  int a = 101;      //%esi
  return a;
}
```

scheduler

```
Thread1:
int foo() {
  int a = 100;      //%esi
  return a;
}

Thread2:
int bar() {
  int a = 101;      //%esi
  return a;
}
```

interrupt:2

interrupt:1

interrupt:3

In this example, both foo and bar are using %esi register for local variable a.

scheduler

foo:

```
push %esi
mov $100, %esi      → intr:2
mov %esi, %eax
pop %esi
ret
```

bar:

```
push %esi           → intr:1
mov $101, %esi      → intr:3
mov %esi, %eax
pop %esi
ret
```

foo
interrupt handler
schedule
context_switch
%esi = 100

bar
%esi = 101
interrupt_handler
schedule
context_switch

foo

bar is running
intr1
foo is running
intr2
bar is running
intr3
foo is running

Let us consider that bar was moved to the ready queue after receiving interrupt (intr:1), and foo was scheduled. After intr:2, interrupt_handler is called; interrupt_handler called schedule; schedule called context_switch. Assuming, none of these routines touch the %esi register, %esi was never saved/restored along this path. After the context switch, bar resumed execution, set %esi to 101, and get interrupted again (intr:3). The context switch logic resumed foo. Because %esi was not touched anywhere in interrupt_handler, schedule, and context_switch routines, when foo resumes, it finds 101 in %esi that is wrong. This problem is because of the context_switch routine. On entry to the context_switch routine: %esi, %edi, %esi, and %ebp are live and contain the values corresponding to the previous thread, whereas on returning, they contain values corresponding to the next thread. If we don't save the values of previous thread's live registers and restore the next thread's live registers in context_switch routine, then we are implicitly incorrectly copying their values. To avoid this problem, along with the %esp, we also have to save/restore the values of live registers.

scheduler

context_switch(struct thread *prev, struct thread *next);

context_switch:

mov ~~4~~ 20(%esp), %eax

push %esi
push %edi
push %ebp
push %ebx

mov %esp, (%eax)

mov ~~8~~ 24(%esp), %eax

mov (%eax), %esp

pop %ebx
pop %ebp
pop %edi
pop %esi

ret

We can save the live registers of the previous thread on the previous thread's stack before saving the stack pointer in its metadata (struct thread). Similarly, we can restore the values of live registers after setting the %esp to next thread's esp (from next thread's metadata).

scheduler

```
context_switch:
push %ebp
push %esi
push %edi
push %ebx
mov 20(%esp), %eax
mov %esp, (%eax)            // prev->esp = %esp
mov 24(%esp), %eax
mov (%eax), %esp            // %esp = next->esp
pop %ebx
pop %edi
pop %esi
pop %ebp
ret
```

previous thread

next thread

scheduler

```
struct list_node *ready_list;
struct thread *cur_thread;
void schedule() {
  if (empty(ready_list))
     return;
  list_insert(ready_list, cur_thread);
  struct thread *prev = cur_thread;
  struct thread *next =  get_next_thread(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

What if the scheduler receives an interrupt in the schedule function?

will discuss in next class