Scheduler

- The scheduler maintains a FIFO queue of all threads
  - read FCFS scheduling (Section- 5.3.1) from Silberschatz and Galvin

- The current running thread is preempted after some fixed interval (say 10 ms) and added to the end of FIFO queue
  - Also called round-robin scheduling
  - read RR scheduling (Section-5.3.4) from Silberschatz and Galvin

Recap

- An application is a group of processes
- A process is a group of threads
- An OS (interrupt_handler and yield so far) schedules the application threads in round-robin order

Recap

- How does OS take control from thread once after it is scheduled?
  - Using timer interrupt

Recap

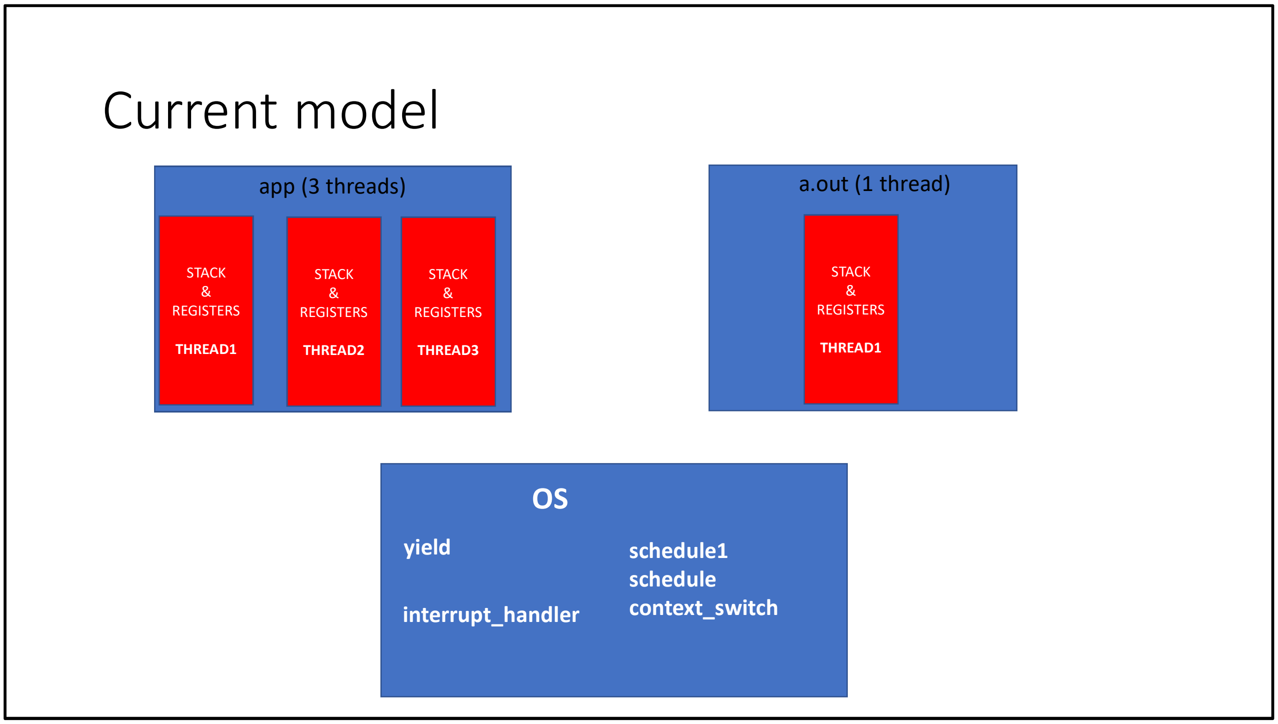
- What happens after a timer interrupt?
  - The corresponding interrupt handler is called that eventually calls context_switch

Recap

- Why does OS not schedule a process instead of a thread after a timer interrupt?
  - Because a process is a group of threads. A thread requires exclusive access to the CPU.

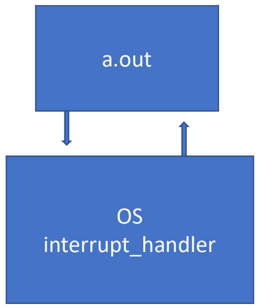
Recap

- What is non-preemptive scheduling?
  - Threads yield the CPU voluntarily. No preemption.



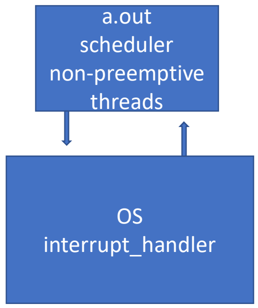
The OS sees a process as a group of threads and schedules one thread at a time in the round-robin order.

Current model



a.out is a single-threaded application. OS preempts the application on a timer interrupt. a.out is completely unaware of the fact that it is being preempted.

Assignment-2



a.out is still a single-threaded application from the OS perspective. OS preempts the application on a timer interrupt. a.out is completely unaware of the fact that it is being preempted. However, a.out can internally create multiple non-preemptive threads(hidden from the OS), and a scheduler in a.out schedules them.

thread_create in a.out

- thread_create in a.out adds the new thread to the scheduler list (scheduler is in a.out)

Scheduler in a.out

- Does the scheduler in a.out need to disable interrupt?

Scheduler in a.out

- Does the scheduler in a.out need to disable interrupt?
  - a.out can't control interrupt
  - interrupts are handled by the OS

Scheduler in a.out

- Does the scheduler in a.out need to disable interrupt?
  - a.out can't control interrupt
  - interrupts are handled by the OS

- How does a.out schedule a new thread if it can't see interrupts?

Scheduler in a.out

- Does the scheduler in a.out need to disable interrupt?
  - a.out can’t control interrupt
  - interrupts are handled by the OS

- How does a.out schedule a new thread if it can’t see interrupts?
  - non-preemptive scheduling doesn’t require interrupts

schedule1

```
schedule1() {
  ~~status = disable_interrupt();~~
  push_back(ready_list, cur_thread);
  schedule();
  ~~set_interrupt_status(status);~~
}
```

schedule

```
struct thread *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_front(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

thread_yield

- Threads invoke thread_yield to yield the CPU voluntarily
  - thread_yield calls schedule1

thread_exit

- Threads invoke thread_exit to terminate themselves
  - thread_exit calls schedule

wait_for_all

- The caller waits until no other thread is available to run
  - repeatedly call schedule1 until the ready_list is empty

thread_create(func, param)

- thread_create allocates struct thread for the new thread
- thread_create adds the current thread to ready_list
- thread_create allocates stack for the new thread
- The target thread function always calls thread_exit to terminate itself

schedule

```
struct thread *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_front(ready_list);
  cur_thread = next;
  context_switch(prev, next);
}
```

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

```
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
```

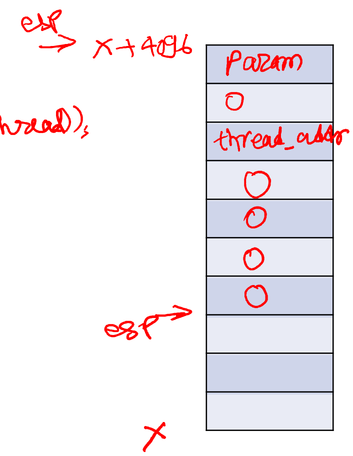
```
struct thread {
   void *esp;
   struct thread *next;
   struct thread *prev;
};
```

thread_create

unsigned *esp = malloc(4096);

esp += 1024;

struct thread *t = malloc(sizeof(struct thread));



thread_create

```
unsigned *stack = malloc(4096);
stack += 1024;
struct thread *t = malloc(sizeof(struct thread));
push param  // parameter
push 0          // fake return address (never returns)
push target_fn
push 0  // ebx
push 0  // edi
push 0  // esi
push 0  // ebp
set esp in struct thread (t)
```

How can we restrict applications from calling schedule directly?

- The application can jump anywhere using the jmp instruction

- In C applications
  - a function pointer can be used to jump anywhere
  - a function can overwrite the return address on the stack to jump anywhere

How can we restrict applications from calling schedule directly?

- The goal of the OS to allow all kind of languages including assembly
- We need some sort of hardware support for this
- Let us assume for now that if an application directly jumps to OS routines, the OS kills the application

Entry to OS

- The entry points to OS are interrupt handlers
  - x86 supports 256 different kinds of interrupts (some of these are exceptions)
    - timer interrupt is one example
    - there could be other interrupts as well
    - each interrupt is associated with a unique number (vector number)
    - e.g., timer interrupt vector number is a specific value between 0 to 256
    - the software can invoke an interrupt using **int $vector_number** instruction

int $ 100

An application can invoke software interrupt (using int instruction) to call an OS routine. The int instruction takes the vector number of the target interrupt handler.

Interrupt descriptor table (IDT)

- IDT lives in memory
- IDT has 256 entries
- IDT contains the addresses of the interrupt handlers
- On timer interrupt 1, handler_1 is called
    - The CPU looks in the IDT for handler_1 and jumps to it

Interrupt descriptor table register (IDTR)

- x86 maintains a special register called IDTR that stores the base and limit of the interrupt descriptor table
  - lidt instruction takes a 6-byte value
  - lower 2-bytes are the limit (size of IDT in bytes)
  - higher 4-bytes are the base address of IDT

IDTR

- On startup, OS disables the interrupt
- It creates an IDT in memory
- Load the base and limit of IDT in IDTR using lidt instruction
- Enables the interrupt

IDT

- What prevents applications from creating a new IDT in their own quota of memory and load them in IDTR using lidt instruction?

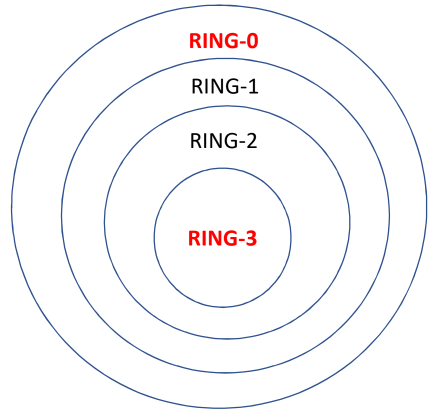
Protection rings

- x86 has four protection rings: 0, 1, 2, and 3
- 0 is most privileged and 3 is the least privileged
- OS executes in ring 0
- Applications execute in ring 3
- Most OSes including Windows and Linux do not use ring 1 and 2

Protection rings

- Some instructions are only allowed in RING-0
  - e.g., lidt, cli, sti
  - IDT can be loaded to IDTR only in RING-0

- Meaning of some instructions are different in RING-0 and RING-3
  - e.g., popf doesn't restore the interrupt flag in RING-3



Protection rings

- We'll come back to protection rings and IDT when we discuss memory isolation

How do applications call yield?

- Can we assign an interrupt vector (say y) to yield
  - Applications can invoke yield using int $y
    - Not good. OS can export at most 256 routines to applications

How do applications call OS routines?

yield => vector 0

read => vector 1

write => vector 3

--

open => vector 255

no more routines can be supported

If we reserve a unique interrupt vector for each OS routine that is exported to user programs, then at most 256 different routines can be exported.

How do applications call OS routines?

- Assign a unique id to each routine that OS exports to applications

routine0 => id 0
routine1 => id 1
routine2 => id 2
…
routine1000 => id 1000
…
routine9999 => id 9999

Instead, the OS assigns a function id (not interrupt vector number) to each of the routines that it wants to export.

How do applications call OS routines?

- OS implements a routine system_call
- assign a vector to routine system_call
  - system_call => 128
- To invoke the system call applications can use
  - int $128

One interrupt vector is assigned to system_call routine in OS. Applications can pass the function id as an argument to the system call handler (system_call).

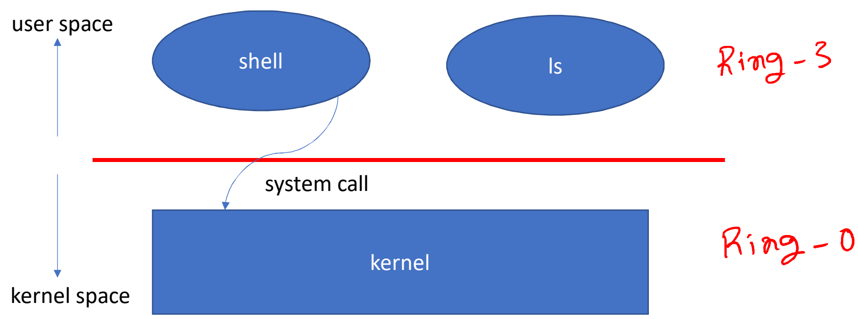
system_call

```
system_call(int id) {
  switch (id) {
    case 0: routine0(); break;
    case 1: routine1(); break;
    …
    case 9999: routine9999(); break;
    default: // not a valid system call
  }
}
```

Unix operating systems

- xv6 and Linux are Unix-like operating systems
  - read chapter-0 from the xv6 book
- Applications execute in the user space (ring-3)
- OS (kernel) executes in the kernel space (ring-0)
- An application thread makes system calls to use kernel services
  - e.g., the yield system call to give up the CPU

xv6


user space
shell
ls
Ring - 3
system call
kernel
Ring - 0
kernel space

shell

- shell is the first user process created by the Unix OS
- shell can create more processes
- shell is an ordinary program that takes commands from the user and executes them
  - A command contains the paths (locations) of executable(s)
  - If the absolute path is not given, the path is relative to the current working directory
    - ls (no absolute path), /bin/ls (absolute path)

shell

$
(default shell)

$ ls
when we type ls and press enter, shell executes the ls program and returns to the default shell

exec

- exec system call takes the path of an executable and arguments, which are passed to executable
  - int exec(char *path, char **argv)

- The arguments are passed to the main of the target executable
  - main (int argc, char *argv[])
    - argv in main contains the list of arguments passed by the command line
    - the first argument (argv[0]) is the name of the executable
    - the argv in exec system call is directly passed to the main of the target executable

main

- main(int argc, char *argv[])
  - ./a.out
    - argv[0] : “./a.out”
  - ./a.out hello world
    - argv[0] : “./a.out”
    - argv[1] : “hello”
    - argv[2] : “world”

exec

- exec(char *path, char **argv)

```
char *argv[4];
argv[0] = "./a.out";
argv[1] = "hello";
argv[2] = "world";
argv[3] = NULL;
exec("./a.out", argv);
printf("exec error\n");
```

exec

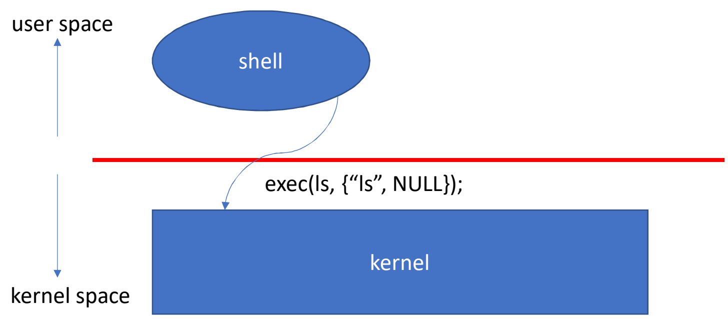
- exec system call loads the target executable in the memory and jumps to main of the target executable

- if the loading is successful, exec system call never returns
  - loading may be unsuccessful if the executable doesn't exist

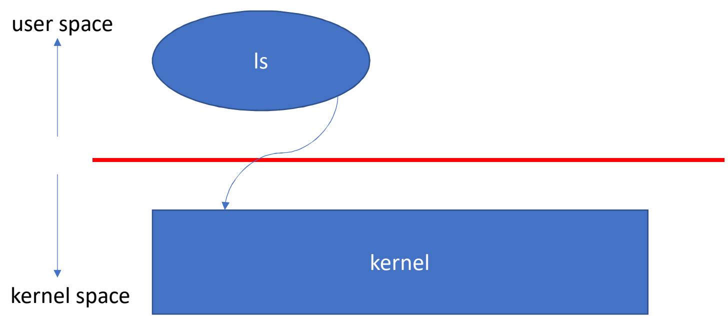
- exec system call passes the arguments to main of the target executable

exec system call takes the name of a target executable and arguments to the main routine of executable. exec system call handler loads the target executable into memory, jumps to the main routine, and pass parameters to the main routine. If loading is successful the exec system call never returns. In other words, the exec system call overwrites the current process code and data with the target executable code and data.

exec



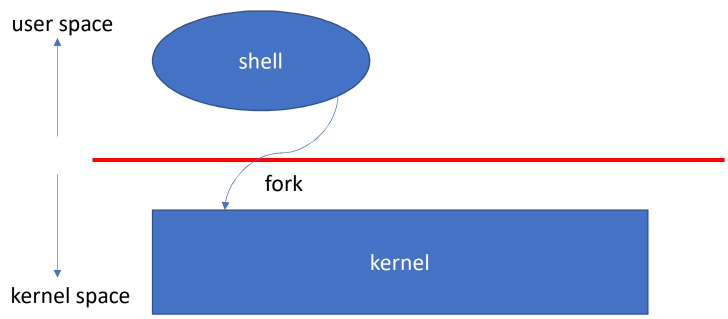
after exec

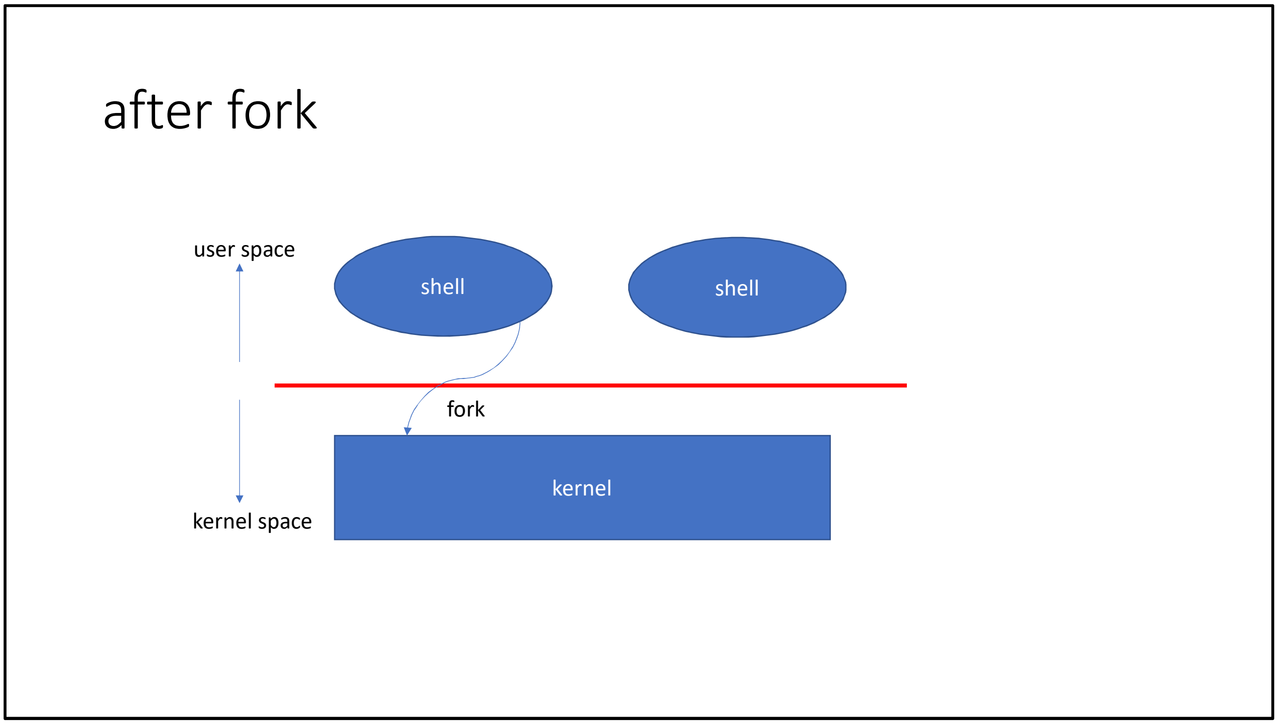


fork

- fork system call creates a new process that is identical to the caller
  - The new process is called the child process
  - The kernel associates a process identifier, or pid, with each process
  - fork returns in both the parent and the child
  - In the parent, fork returns the child's pid
  - In the child, fork returns 0

fork





After the fork, a replica of the same process is created. The new process is called the child process. The child process starts execution just after the fork system call.

fork

```
int pid = fork();
if (pid > 0) {
  printf("In parent: child's pid : %d\n", pid);
} else if (pid == 0) {
  printf("In child\n");
} else {
  printf("fork error\n");
}
```

If the fork is successful, this program will print both the statements (one in the parent, other in the child). The print statements can be in any order, depending on when the scheduler was invoked.

exit

- exit system call terminates the current process
  - exit system call causes the calling process to stop executing and release all the resources (e.g., memory)

wait

- The parent can invoke the wait system call to wait for one of its children to exit

wait

```
int pid = fork();
if (pid > 0) {
  printf("parent: child=%d\n", pid);
  pid = wait();
  printf("parent: child %d is done\n", pid);
} else if (pid == 0) {
  printf("child: exiting\n");
  exit();
} else
  printf("fork error\n");
```

Sample output1:
parent: child=1234
child: exiting
parent: child 1234 is done

(parent called the printf first)

Sample output2:
child: exiting
parent: child=1234
parent: child 1234 is done

(child called the printf first)

In this case, printf after the wait system call in the parent will execute after the child has exited.