How to Yield/Wait?

- Must switch the "CPU state" (the context) captured in its registers and PSW
- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING → READY
  - Next process: READY → RUNNING

1. Save kernel registers of Current on its kernel stack
2. Save kernel SP of Current in its PCB
3. Restore kernel SP of Next from its PCB
4. Restore kernel registers of Next from its kernel stack

How to Yield/Wait?

- Must switch the "CPU state" (the context) captured in its registers and PSW
- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING → READY
  - Next process: READY → RUNNING

content of general registers while running in kernel mode

1. Save kernel registers of Current on its kernel stack
2. Save kernel SP of Current in its PCB
3. Restore kernel SP of Next from its PCB
4. Restore kernel registers of Next from its kernel stack

Yielding in Slo-Mo

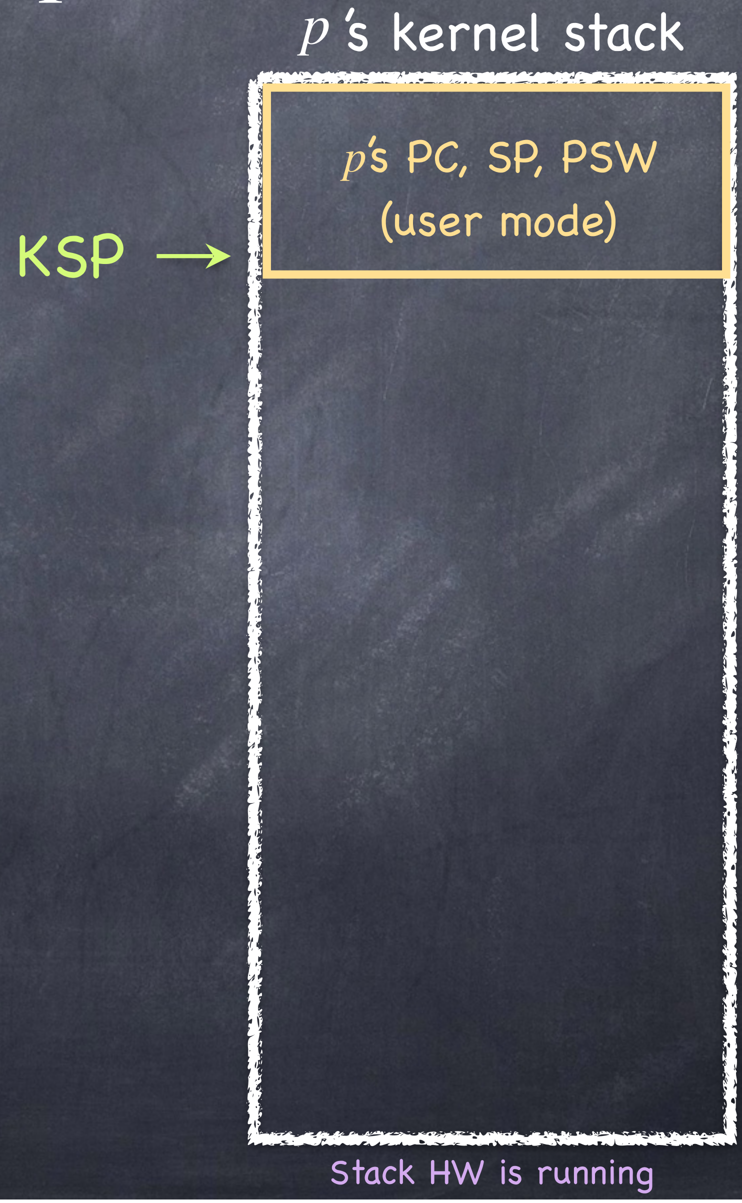
from p to q

- Process p receives a timer interrupt

Yielding in Slo-Mo

from p to q

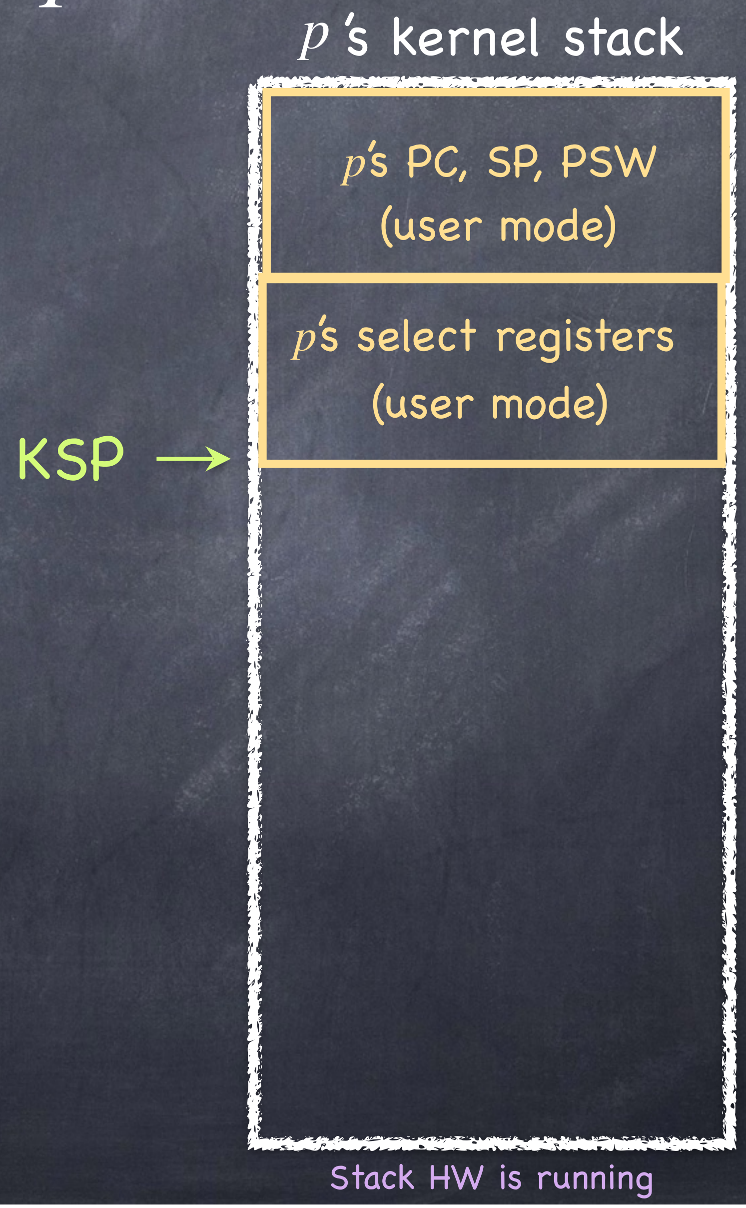
- Process p receives a timer interrupt
  - HW pushes PC, SP, PSW



Yielding in Slo-Mo

from p to q

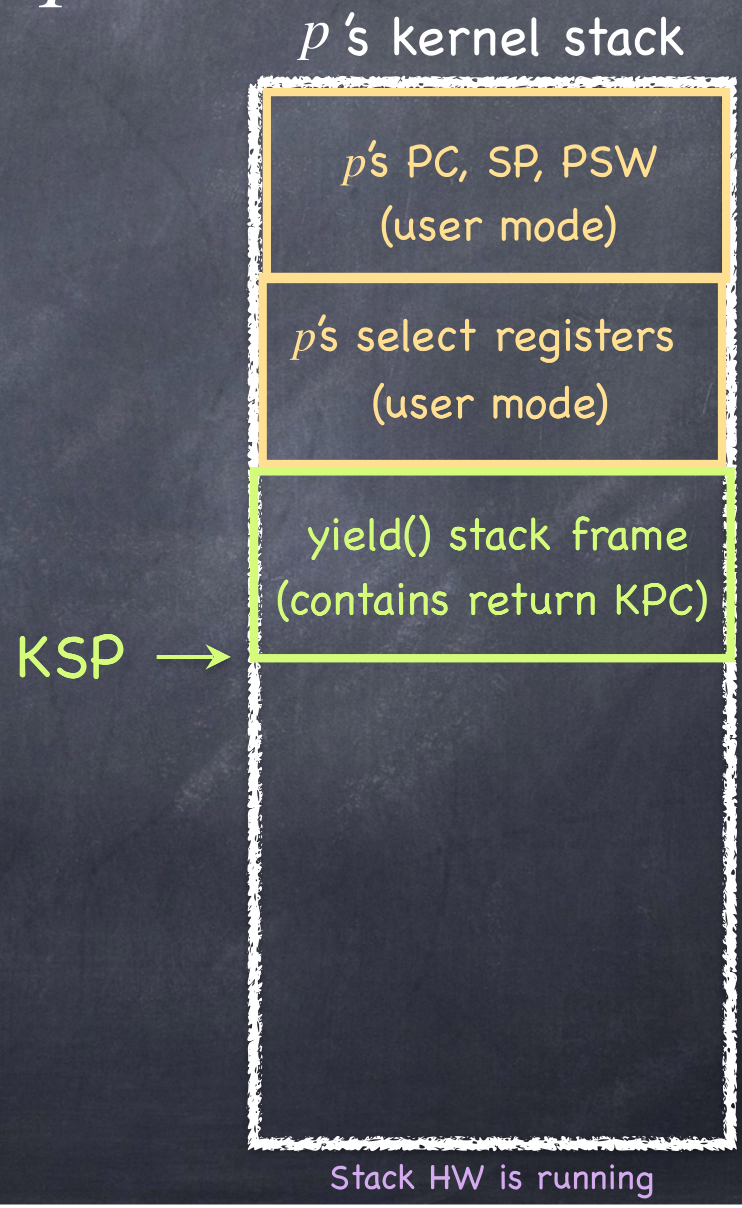
- Process p receives a timer interrupt
  - HW pushes PC, SP, PSW
  - SW (handler) pushes select general registers



Yielding in Slo-Mo

from p to q

- Process p receives a timer interrupt
  - HW pushes PC, SP, PSW
  - SW (handler) pushes select general registers
  - handler (dispatcher) calls yield()



Yield()

```
struct pcb *current, *next;

void yield(){
   assert(current->state == RUNNING);
   current->state = READY;
   readyQueue.add(current);
   next = scheduler();
   next->state = RUNNING;
   ctx_switch(&current->sp, next->sp)
   current = next;
}
```

Yield()

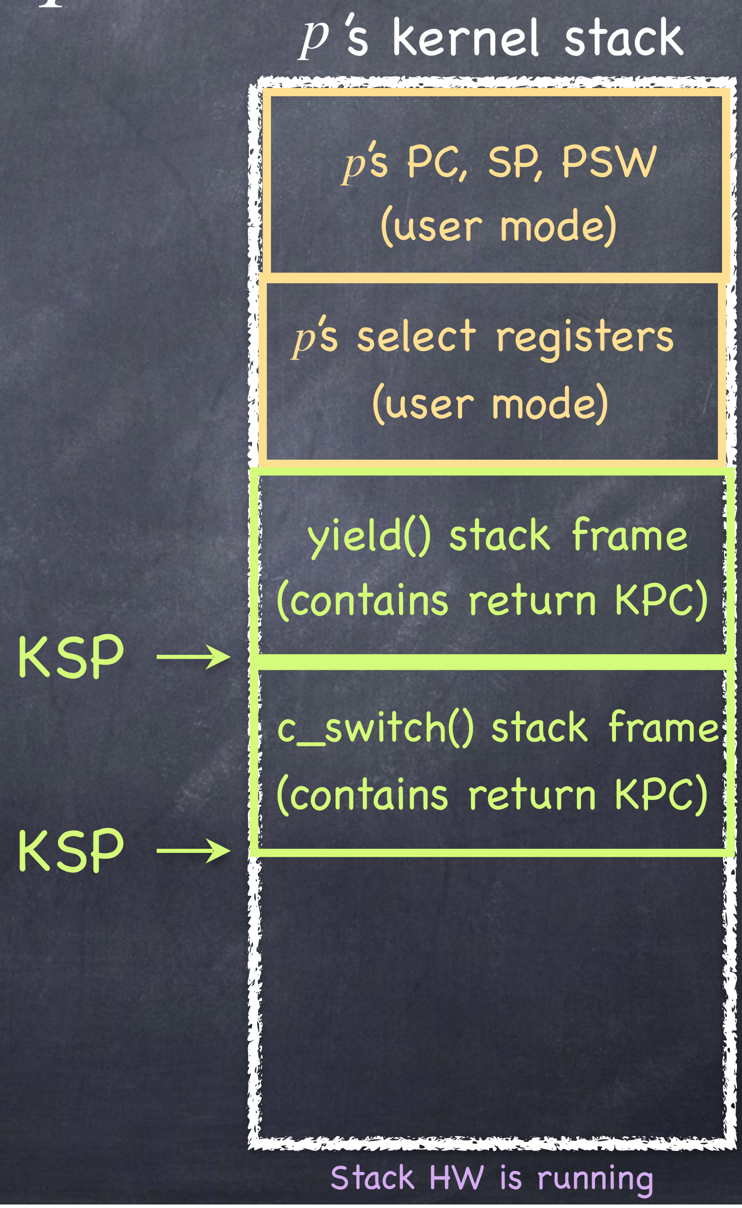
```
struct pcb *current, *next;

void yield(){
   assert(current->state == RUNNING);
   current->state = READY;
   readyQueue.add(current);
   next = scheduler();
   next->state = RUNNING;
   ctx_switch(&current->sp, next->sp)
   current = next;
}
```

Yielding in Slo-Mo

from p to q

- Process p receives a timer interrupt
  - HW pushes PC, SP, PSW
  - SW (handler) pushes select general registers
  - handler (dispatcher) calls yield()
  - dispatcher calls context_switch()



ctx_switch(&old_sp, new_sp)

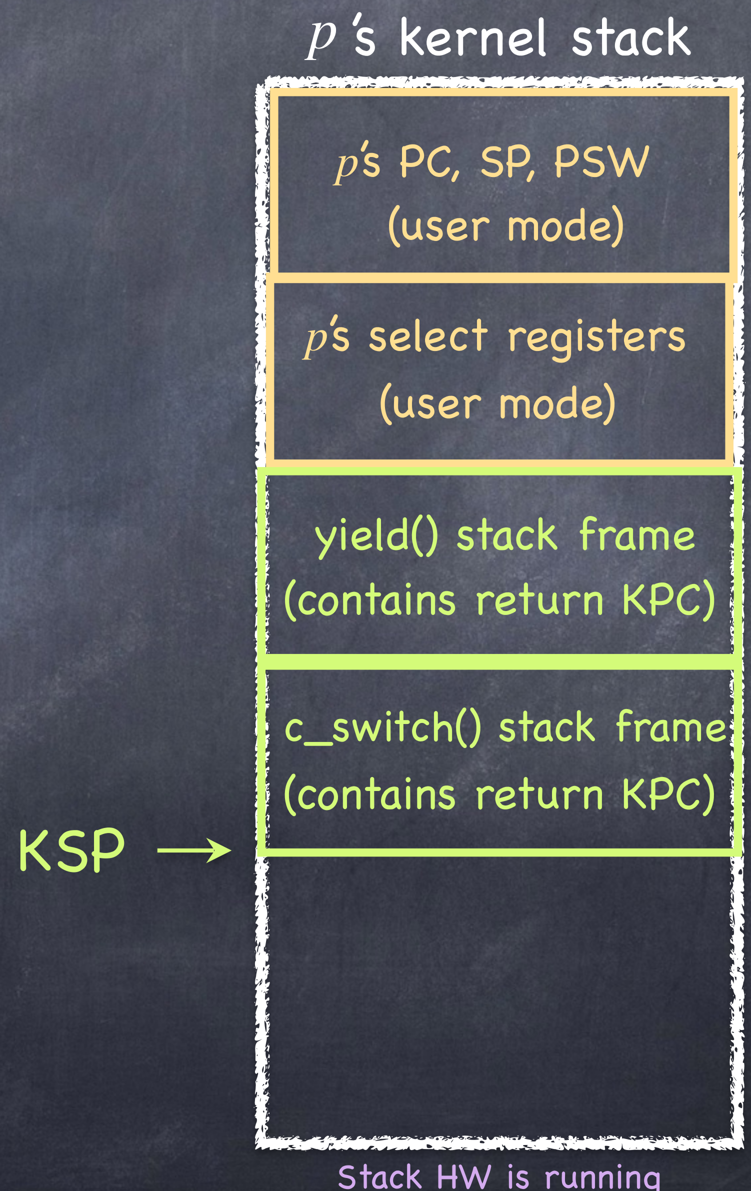
```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
```

```
struct pcb *current, *next;

void yield(){
    assert(current->state == RUNNING);
    current->state = READY;
    readyQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp)
    current = next;
}
```

ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
```

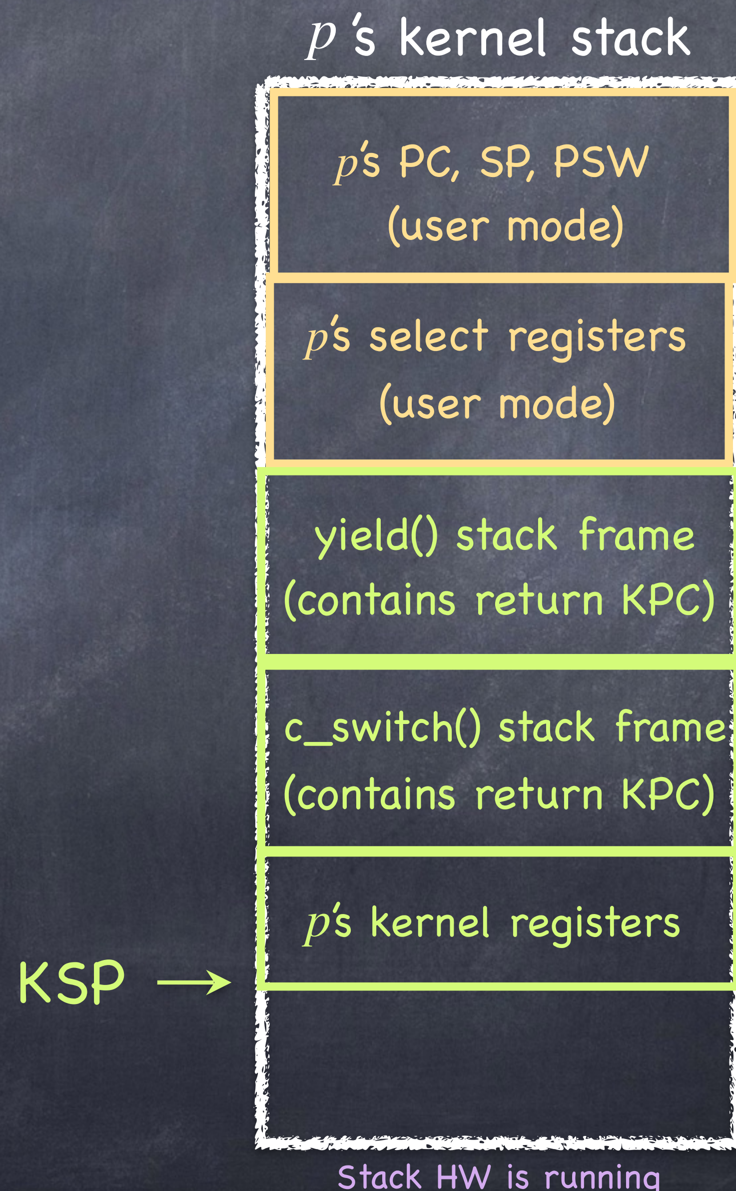


ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register



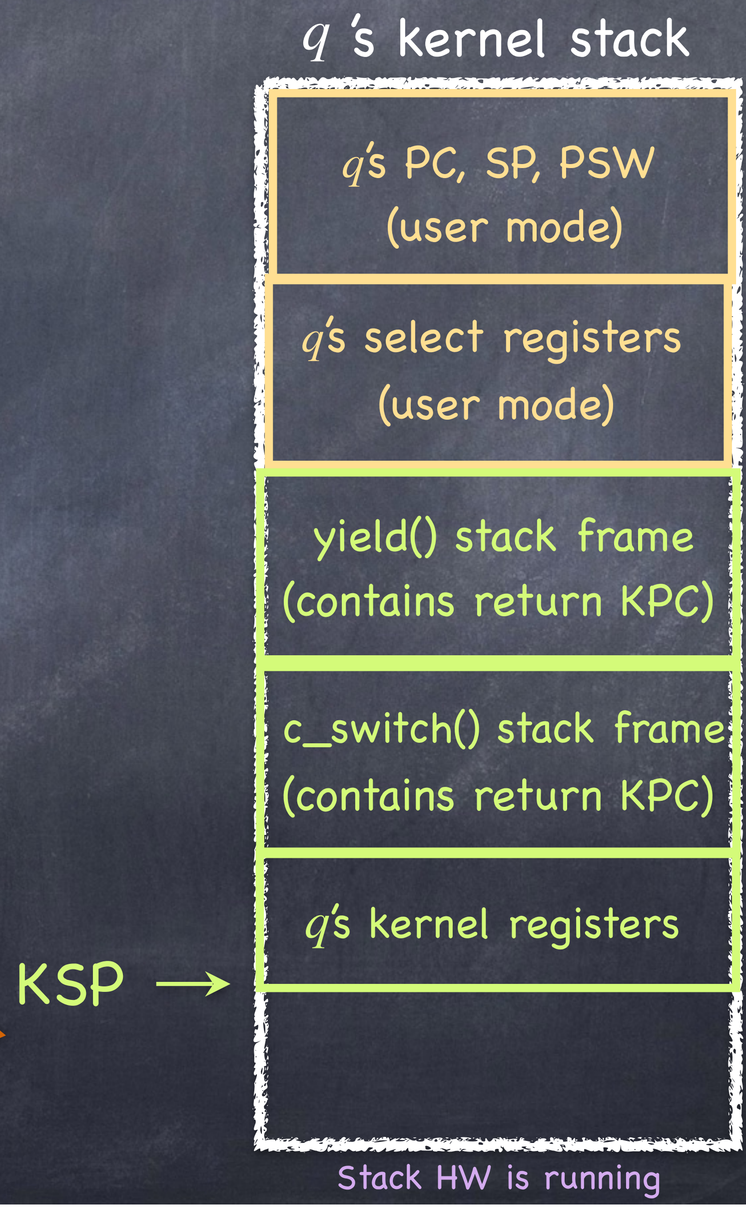
ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register

p is running, but using q's stack!

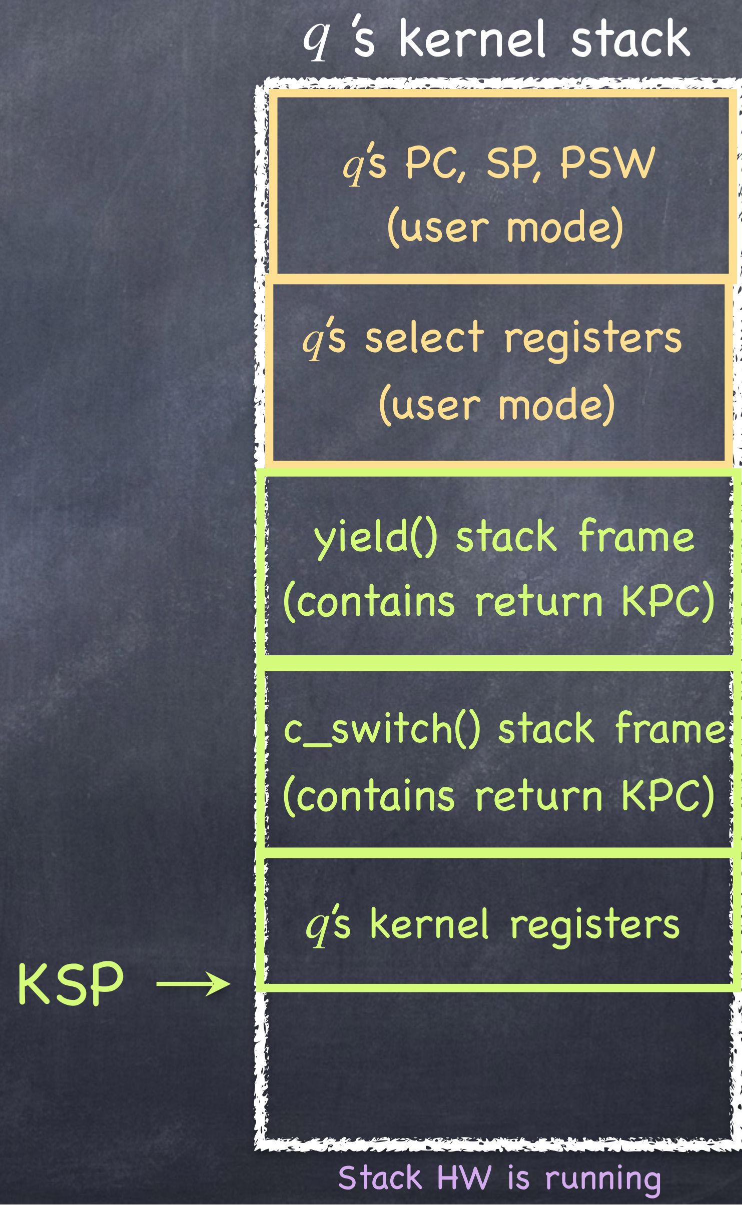


ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %r8
    popq    %r9
    popq    %r10
    popq    %r11
    popq    %r12
    popq    %r13
    popq    %r14
    popq    %r15
    popq    %rbx
    popq    %rbp
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register



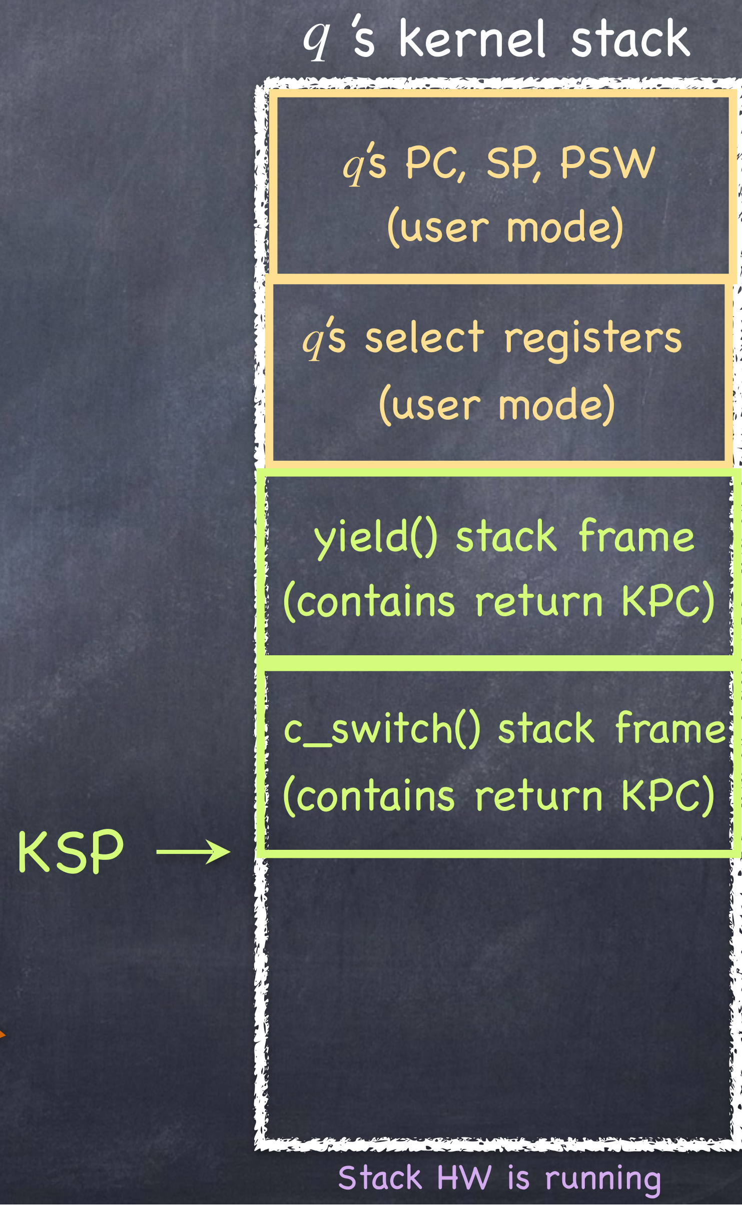
ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %r8
    popq    %r9
    popq    %r10
    popq    %r11
    popq    %r12
    popq    %r13
    popq    %r14
    popq    %r15
    popq    %rbx
    popq    %rbp
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register

p is popping q's kernel registers on the CPU!

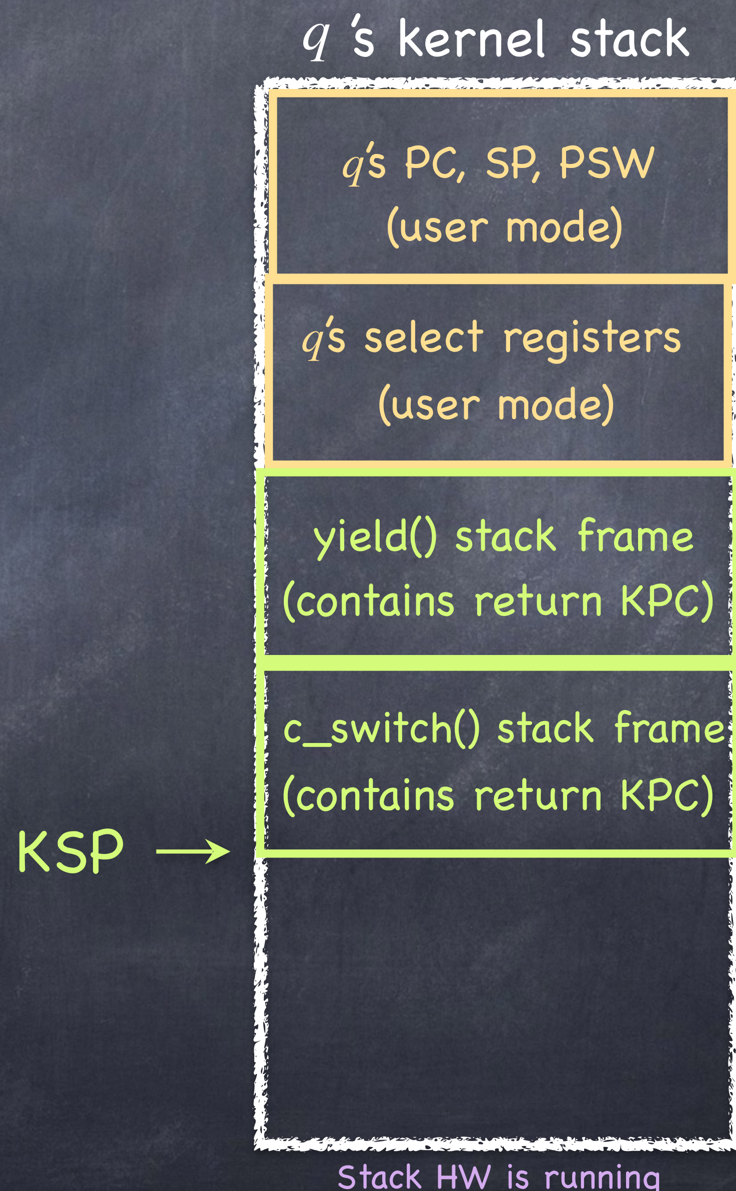


ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %r8
    popq    %r9
    popq    %r10
    popq    %r11
    popq    %r12
    popq    %r13
    popq    %r14
    popq    %r15
    popq    %rbx
    popq    %rbp
    retq
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register



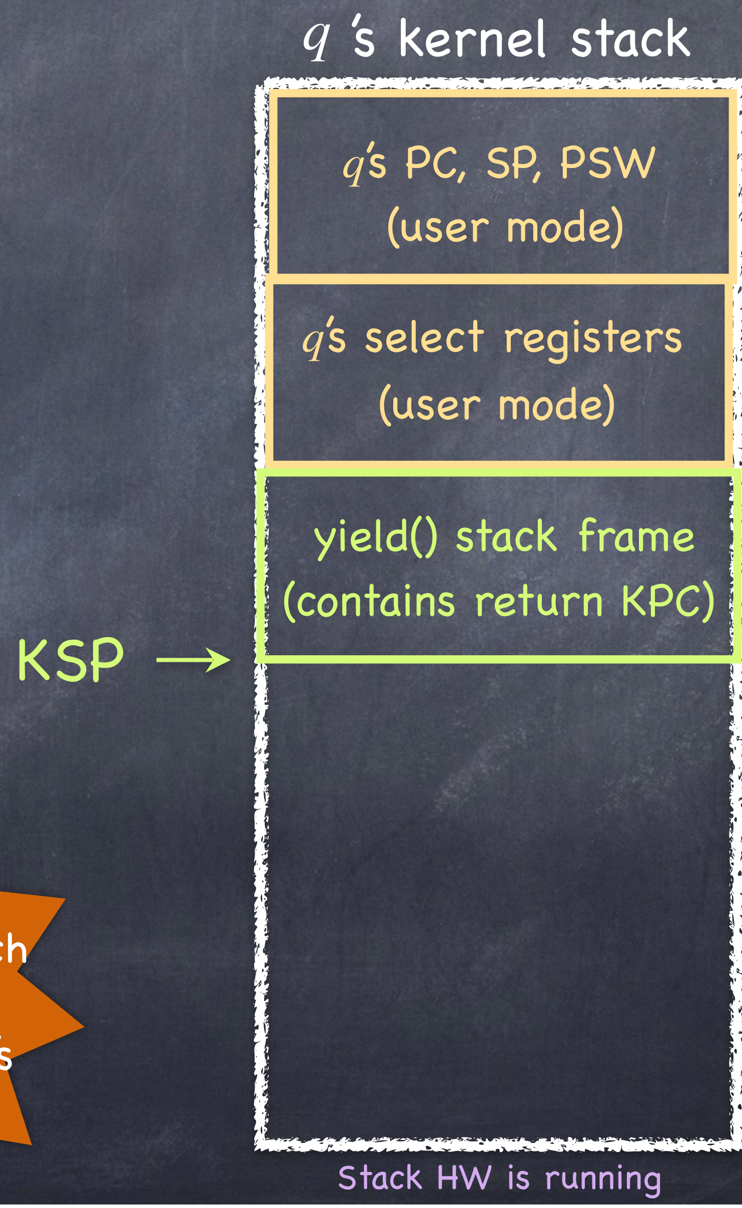
ctx_switch(&old_sp, new_sp)

```
ctx_switch: //PC already saved in frame!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %r8
    popq    %r9
    popq    %r10
    popq    %r11
    popq    %r12
    popq    %r13
    popq    %r14
    popq    %r15
    popq    %rbx
    popq    %rbp
    retq
```

copies p's stack pointer into p's PCB (pointed to by address in rdi)

copies q's KSP (stored in rsi) into CPU's stack pointer register

Upon return from ctx_switch PC register is loaded with q's saved PC!



Back to Yield()

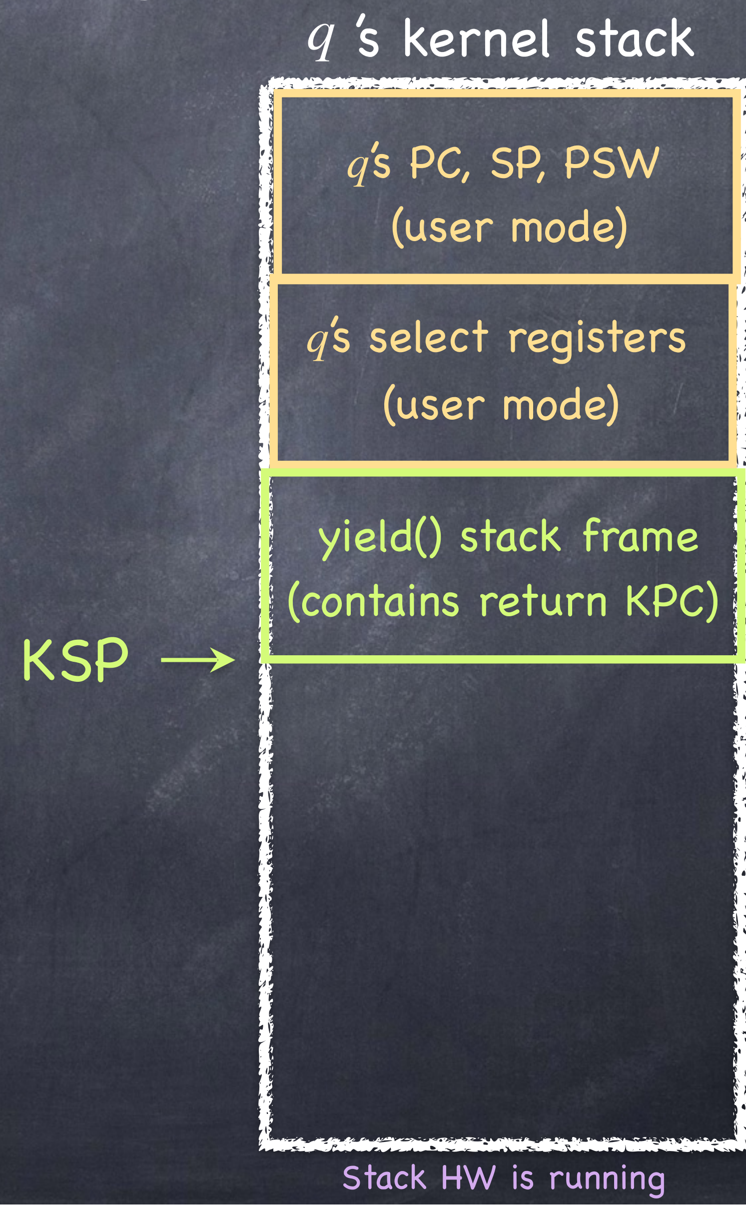
(but with q running!)

```
struct pcb *current, *next;

void yield(){
   assert(current->state == RUNNING);
   current->state = READY;
   readyQueue.add(current);
   next = scheduler();
   next->state = RUNNING;
   ctx_switch(&current->sp, next->sp)
   current = next;
}
```

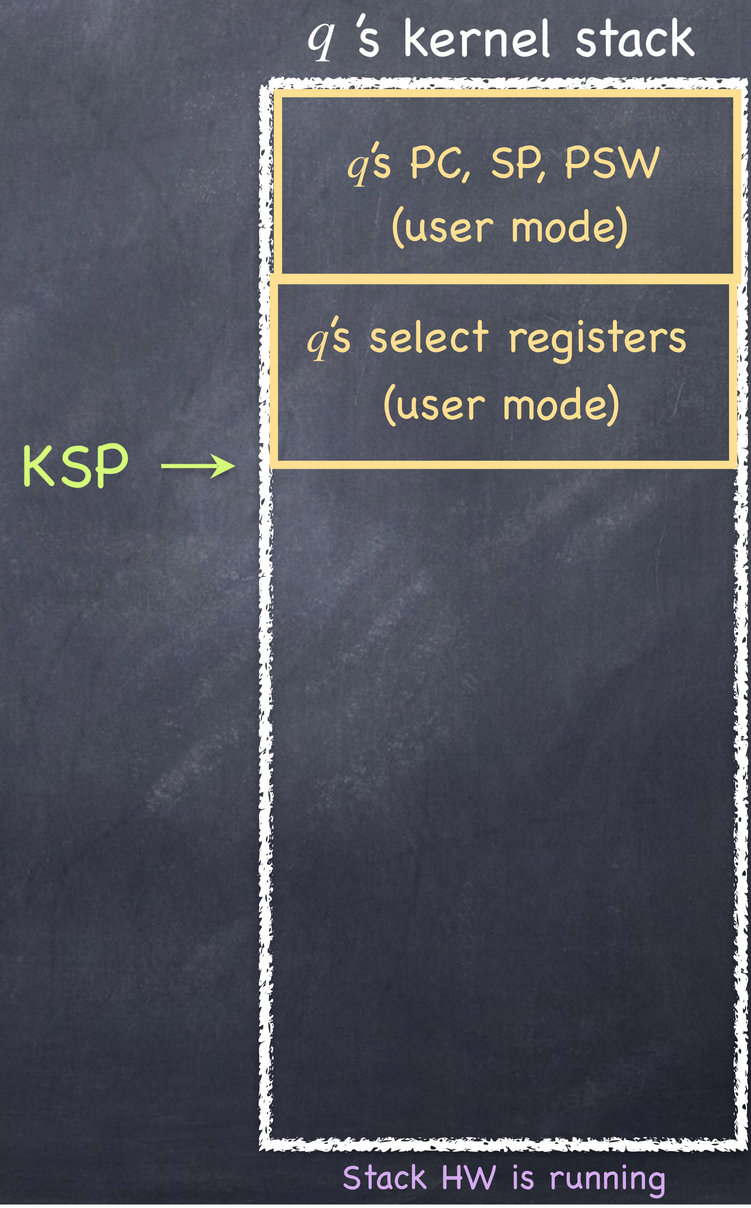
KPC → current = next;

q returns to timer interrupt handler that invoked yield()



Back in the Handler

- We are out the woods — we know what happens now!
  - before the handler terminates, it pops form the kernel stack the user mode registers it saved on the stack when it was invoked

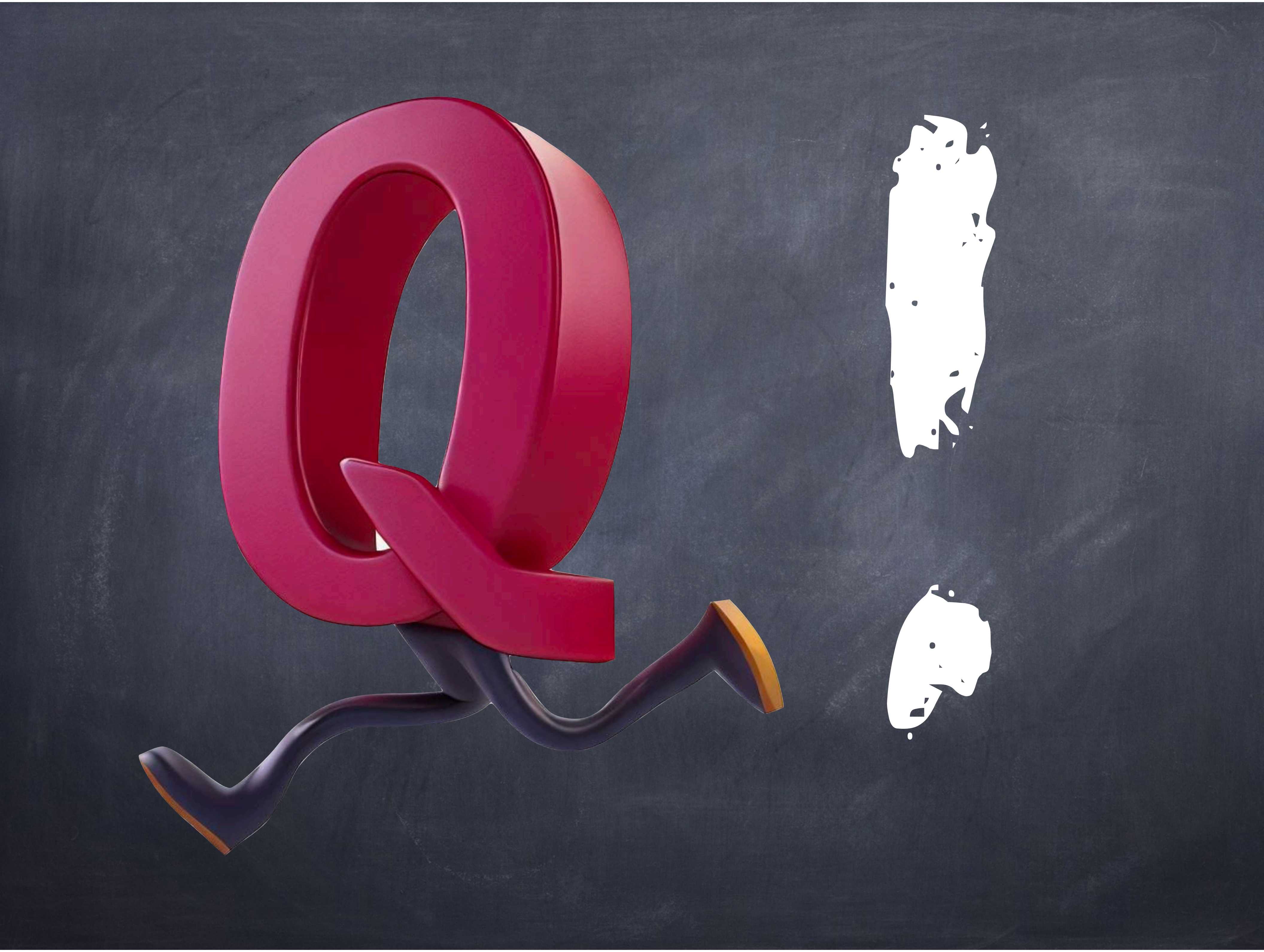


Back in the Handler

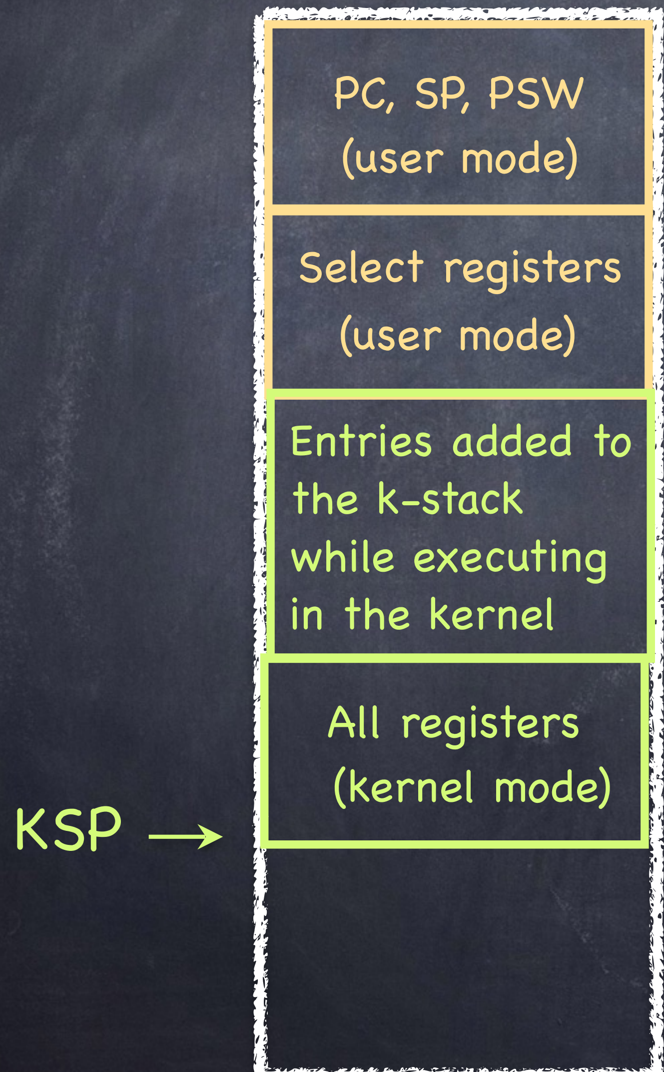
- We are out the woods — we know what happens now!
  - before the handler terminates, it pops form the kernel stack the user mode registers it saved on the stack when it was invoked
  - RETURN_FROM_INTERRUPT!
    - HW pops the save d values of PC, SP, and PSW to the appropriate registers

Back in the Handler

- We are out the woods — we know what happens now!
  - before the handler terminates, it pops form the kernel stack the user mode registers it saved on the stack when it was invoked
  - RETURN_FROM_INTERRUPT!
    - HW pops the save d values of PC, SP, and PSW to the appropriate registers



The “Hybernated” Process p



- PCB contains address of the base of p's kernel stack
- Kernel stack stores
  - p's context when it was running in user mode
  - the content of all p's registers when it was running in kernel mode, before being suspended, including the value of the PC it will get back to when resumed

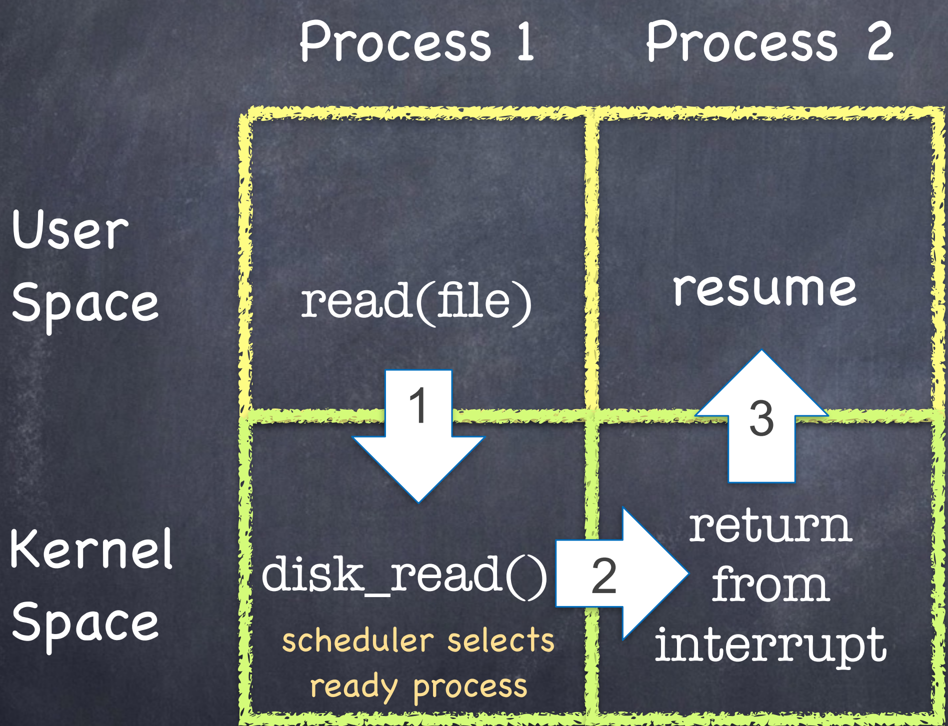
Anybody there?

- What if no process is READY?
  - scheduler() would return NULL — aargh!
- No panic on the Titanic:
  - OS always runs a low priority process, in an infinite loop executing the HLT instruction
    - halts CPU until next interrupt
  - Interrupt handler executes yield() if some other process is put on the Ready queue

Three Flavors of Context Switching

- Interrupt: from user to kernel space
  - on system call, exception, or interrupt
  - Stack switch: P_x user stack $\rightarrow P_x$ kernel stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Stack switch: P_x kernel stack $\rightarrow P_y$ kernel stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Stack switch: P_x kernel stack $\rightarrow P_x$ user stack

Switching between Processes



1. Save Process 1 user registers
2. Save Process 1 kernel registers and restore Process 2 kernel registers
3. Restore Process 2 user registers

System Calls to Create a New Process

- Must, implicitly or explicitly, specify the initial state of every OS resource belonging to the new process.
    - Windows
        - CreateProcess(...);
    - Unix (Linux)
        - fork() + exec(...)

CreateProcess (Simplified)

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

fork (actual form)

process identifier

```
int pid = fork();
```

..but needs exec(...)

[Unix]

Kernel Actions to Create a Process

- fork()
  - allocate ProcessID
  - initialize PCB
  - create and initialize new address space
    - identical to the one of the caller
    - returns twice, once to the parent and once to the child, but with different values (child's pid and 0, respectively)
  - inform scheduler new process is READY
- exec(program, arguments)
  - load program into address space
  - copy arguments into address space's memory
  - initialize h/w context to start execution at ``start"

Creating and managing processes

Syscall	Description
fork()	Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child
exec (prog, args)	Run application prog in the current process with the specified args (replacing any code and data that was present in process)
wait (&status)	Pause until a child process has exited
exit (status)	Current process is complete and should be garbage collected.
kill (pid, type)	Send an signal (≈ interrupt) of a specified type to a process (a bit of an overdramatic misnomer...)

[Unix]

Fork in action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Fork in action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 14
Program A/B

PC

pid
0

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

```
main() {
   ...
   return;
}
```

Fork in action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Status
0

Process 14
Program B

PC

```
main() {
   ...
  return;
}
```

calls exit(0)

Fork in action

```
#include <stdio.h>
#include <unistd.h>

int main() {

   int child_pid = fork();

   if (child_pid == 0) {          // child process
        printf("I am process %d… I mean, process %d\n", childpid, getpid());
        return 0;
   } else {                       // parent process
        printf("I am %d the parent of process %d\n", getpid(), child_pid);
        return 0;
   }
}
```

Possible outputs?