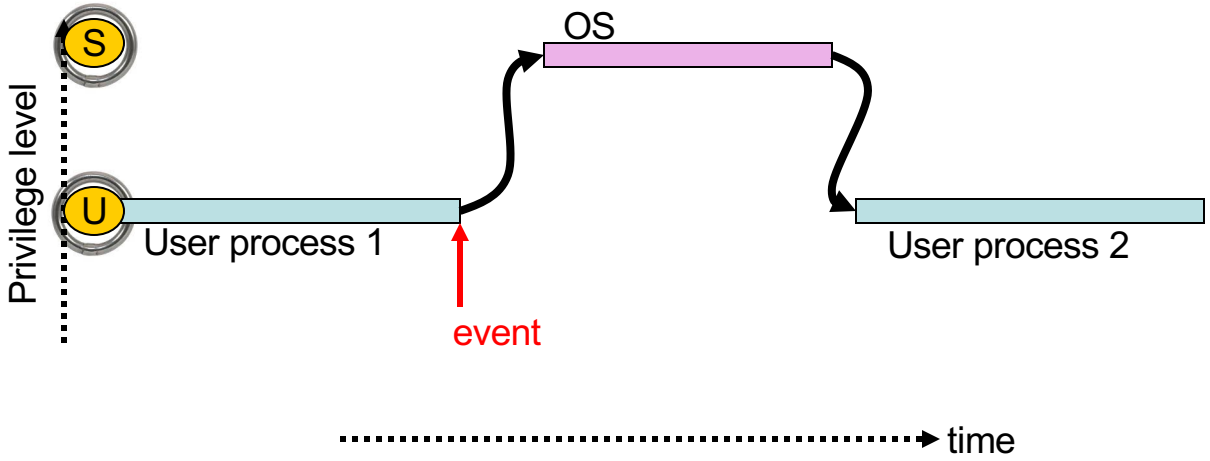
Interrupts, Exceptions, System Calls and Context Switching

Chester Rebeiro
IIT Madras

OS and Events

- OS is event driven
  - i.e. executes only when there is an interrupt, trap, or system call

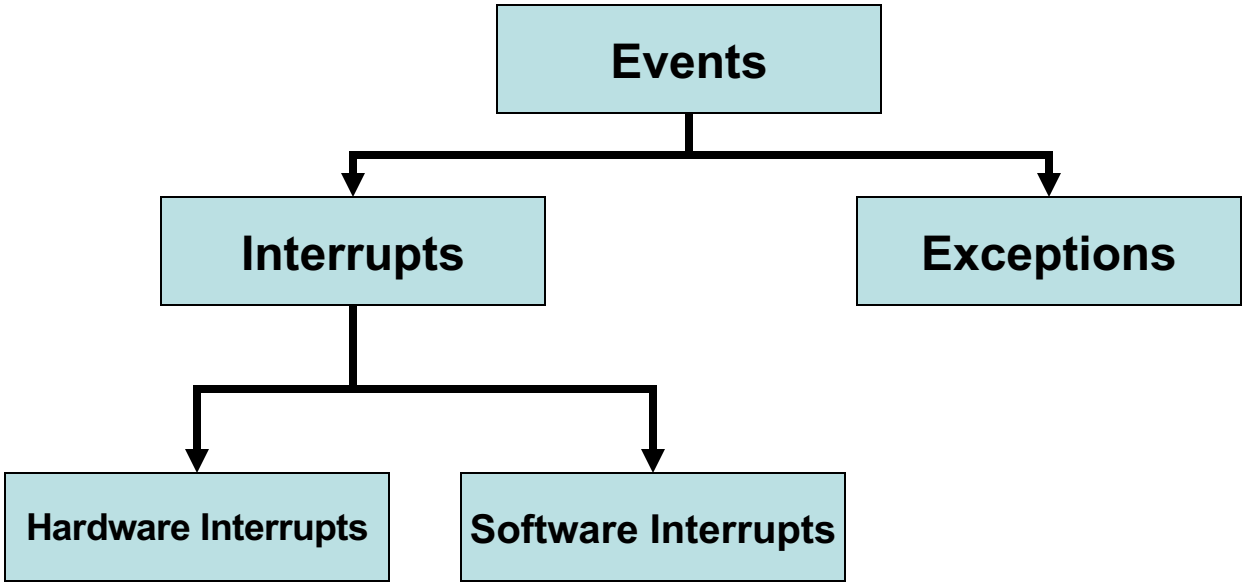


Why event driven design?

- OS cannot trust user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS

- OS needs to guarantee fairness to all user processes
  - One process cannot ‘hog’ CPU time
  - Timer interrupts

Event Types


Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Events

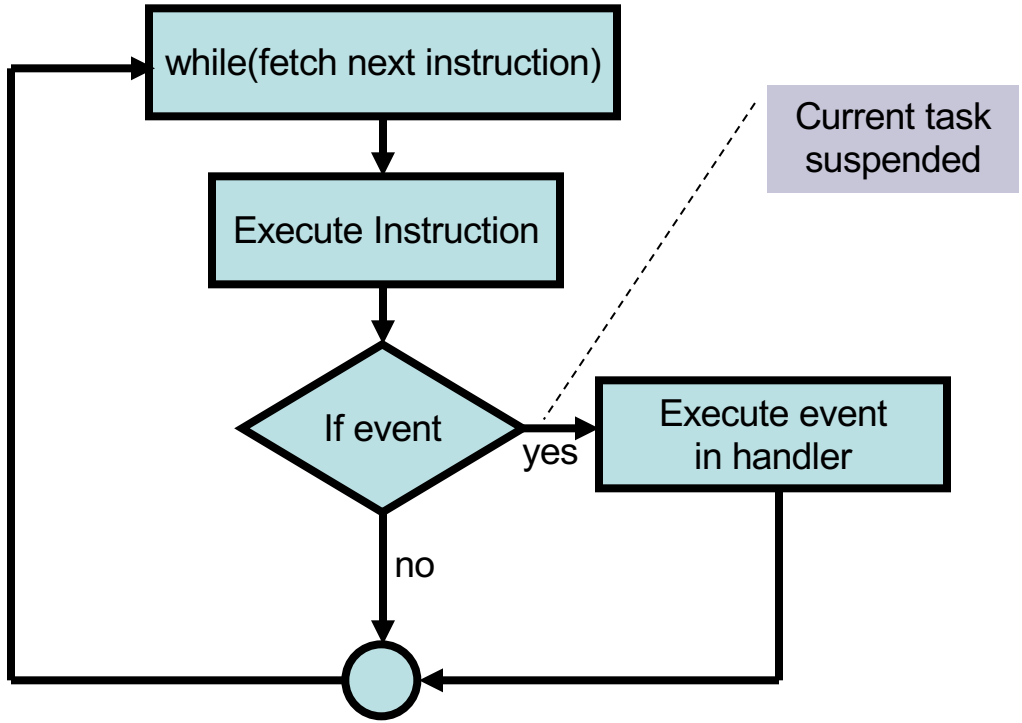
- Interrupts : raised by hardware or programs to get OS attention
  - Types
    - Hardware interrupts: raised by hardware devices
    - Software Interrupts: raised by user programs (such as ecall instruction)

- Exceptions : due to illegal operations

Why Hardware Interrupts?

- Several devices connected to the CPU
  - eg. Keyboards, mouse, network card, etc.
- These devices occasionally need to be serviced by the CPU
  - eg. Inform CPU that a key has been pressed
- These events are asynchronous i.e. we cannot predict when they will happen.

Event view of CPU



Interrupts in RISC V

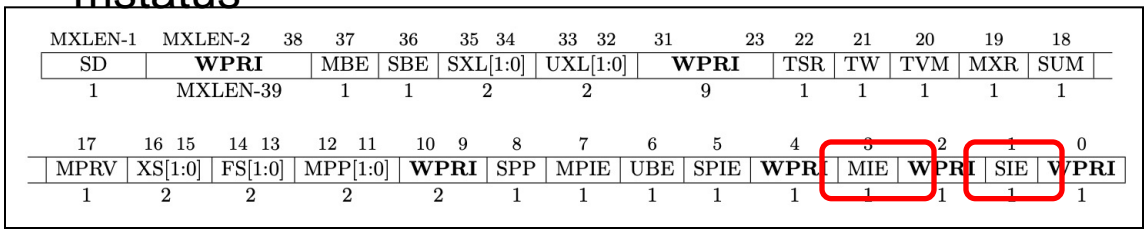
- Different interrupts for different modes
- Enabling and Disabling Interrupts **globally**
  - MIE bit in mstatus register for enabling machine mode interrupts
  - SIE bit in the mstatus register for enabling supervisor mode interrupts
- Previous Mode before the interrupt
  - MPP (2 bits) and SPP (1 bit)

User Mode

Supervisor Mode

Machine Mode

mstatus



MXLEN-1	MXLEN-2 38	37	36	35 34	33 32	31 23	22	21	20	19	18
SD	**WPRI**	MBE	SBE	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM
1	MXLEN-39	1	1	2	2	9	1	1	1	1	1

17	16 15	14 13	12 11	10 9	8	7	6	5	4	3	2 1	0
MPRV	XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	UBE	SPIE	**WPRI**	MIE	**WPRI**	SIE	**WPRI**
1	2	2	2	2	1	1	1	1	1	1	1	1	1

Interrupts in RISC V

- Interrupts Pending and Interrupt Enable

15	12	11	10	9	8	7	6	5	4	3	2	1	0
0		MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
4		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.14: Standard portion (bits 15:0) of `mip`.

15	12	11	10	9	8	7	6	5	4	3	2	1	0
0		MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
4		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.15: Standard portion (bits 15:0) of `mie`.

Machine mode visible

mip.ssip : supervisor software interrupt pending
mip.msip : machine mode software interrupt pending
mie.ssie : supervisor software interrupt enable
mip.msie : machine mode software interrupt enable

mip.stip : supervisor timer interrupt pending
mip.mtip : machine mode timer interrupt pending
mie.stie : supervisor timer interrupt enable
mip.mtie : machine mode timer interrupt enable

15	10	9	8	6	5	4	2	1	0
0		SEIP	0		STIP	0		SSIP	0
6		1	3		1	3		1	1

Figure 4.6: Standard portion (bits 15:0) of `sip`.

15	10	9	8	6	5	4	2	1	0
0		SEIE	0		STIE	0		SSIE	0
6		1	3		1	3		1	1

Supervisor mode visible

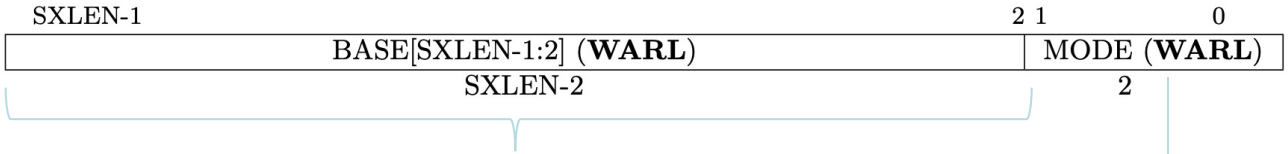
mip.seip : supervisor external interrupt pending
mip.meip : machine mode external interrupt pending
mie.seie : supervisor external interrupt enable
mip.meie : machine mode external interrupt enable

RISC V Exception & Interrupt Handling

Event occured **What to execute next?**

Set PC to interrupt vector address

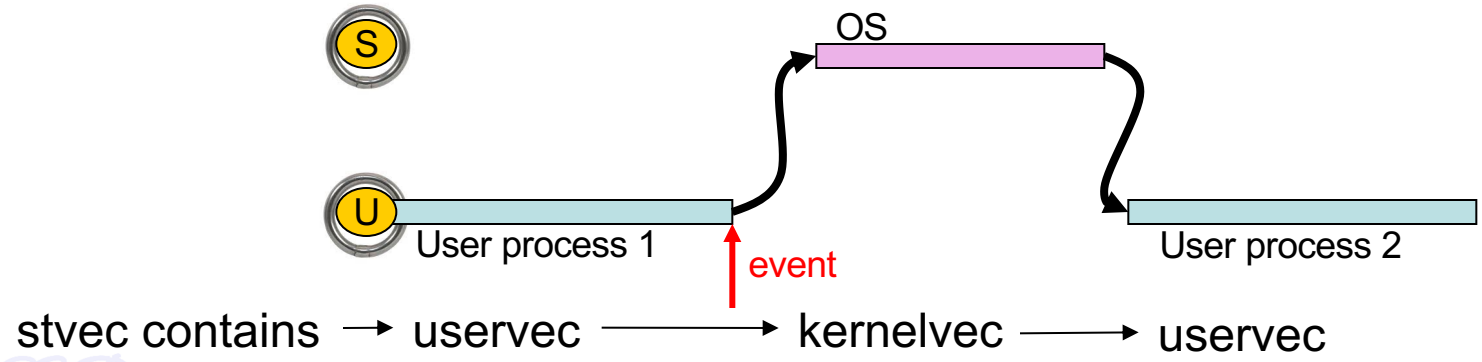
stvec register



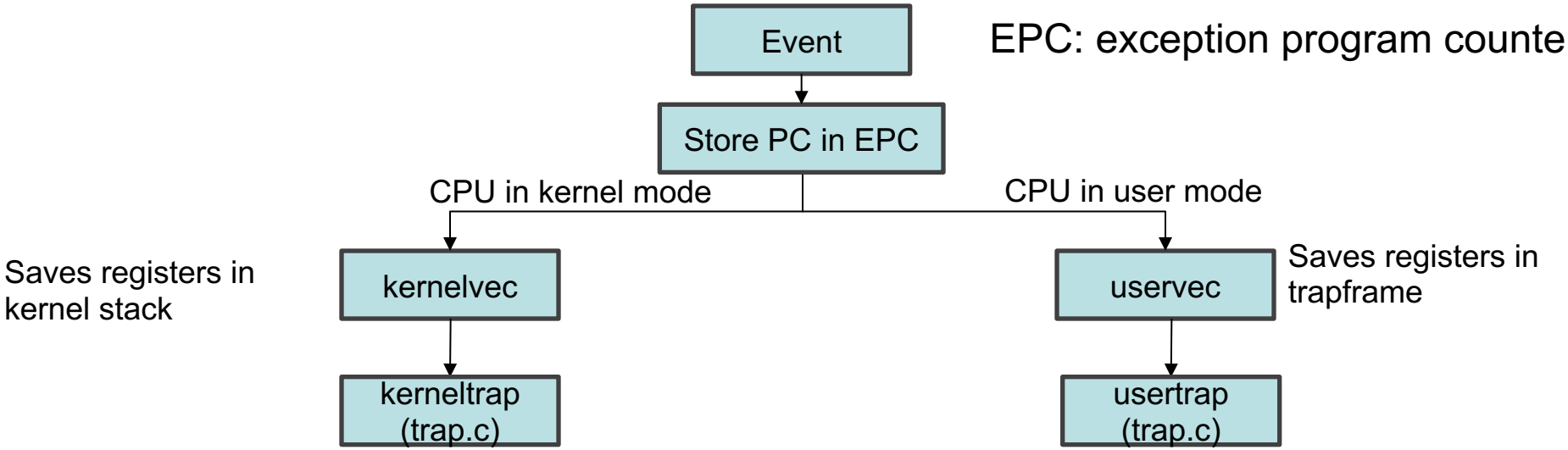
Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Interrupt Vector Handlers in xv6

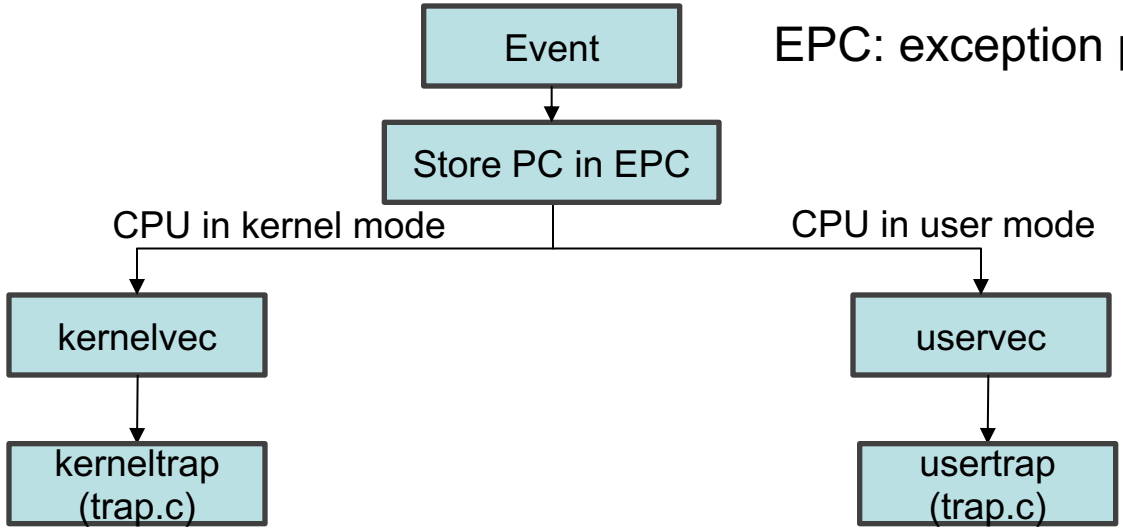
- kernelvec (kernelvec.S)
  - Used to handler interrupts when in kernel address space
- uservec (trampoline.S)
  - Used to handle interrupts when in user address space
  - (recollect) uservec is mapped to the highest page in each user process



Interrupt Handlers in xv6



Interrupt Handlers in xv6



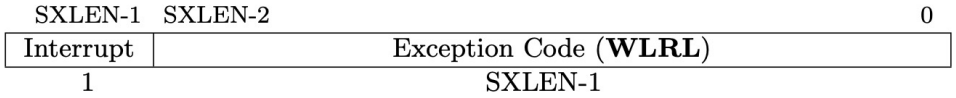
EPC: exception program counter

What caused the interrupt?

Hardware
Timer
Exception
→ panic

Hardware
Timer
Software (system call)
Exception
→ Kill process

scause register: what caused the interrupt?



scause register

Figure 4.11: Supervisor Cause register `scause`.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Service system calls

usertrap (trap.c)

```
53   if(r_scause() == 8){
54     // system call
55
56     if(p->killed)
57       exit(-1);
58
59     // sepc points to the ecall instruction,
60     // but we want to return to the next instruction.
61     p->trapframe->epc += 4;
62
63     // an interrupt will change sstatus &c registers,
64     // so don't enable until done with those registers.
65     intr_on();
66
67     syscall();
```

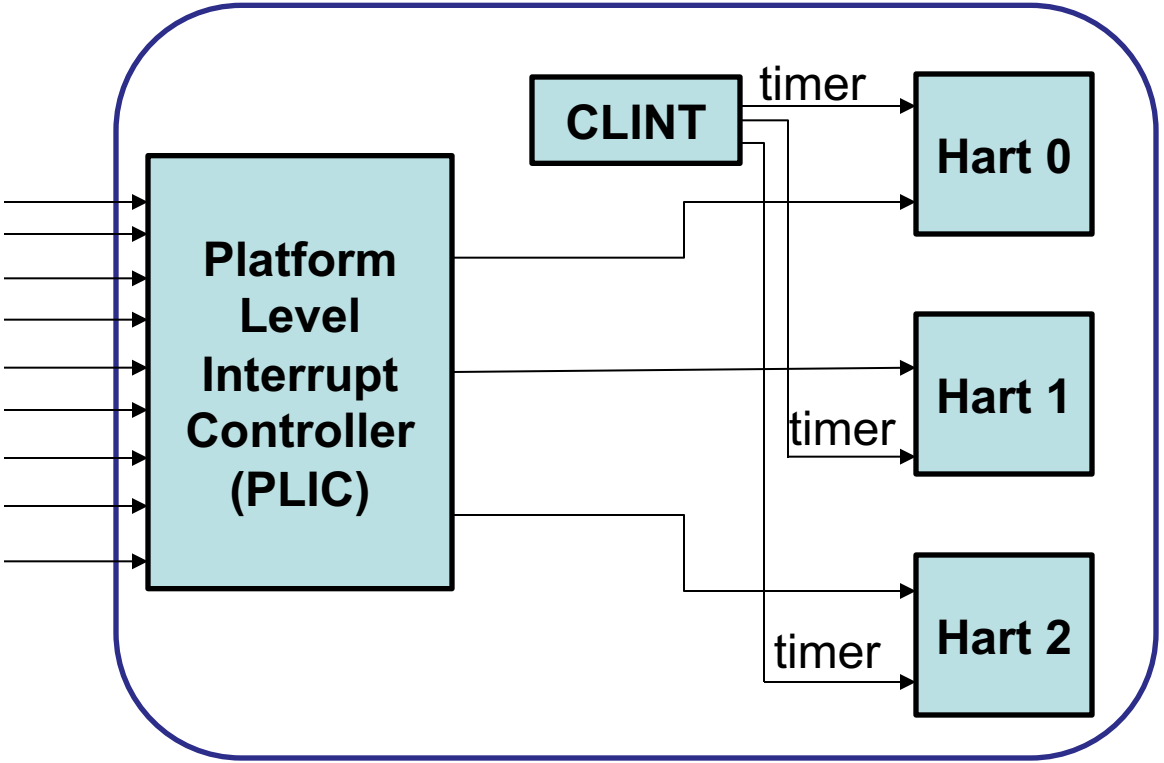
syscall.c

```
132 void
133 syscall(void)
134 {
135   int num;
136   struct proc *p = myproc();
137
138   num = p->trapframe->a7;
139   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140     p->trapframe->a0 = syscalls[num]();
141   } else {
142     printf("%d %s: unknown sys call %d\n",
143             p->pid, p->name, num);
144     p->trapframe->a0 = -1;
145   }
146 }
```

Invoke system call according to syscall number (in a7)

(a0 passed back to user program as system call return)

External and Timer Interrupts



Identifying External Interrupts

trap.c

```
176 int
177 devintr()
178 {
179   uint64 scause = r_scause();
180
181   if((scause & 0x8000000000000000L) &&
182      (scause & 0xff) == 9){
183     // this is a supervisor external interrupt, via PLIC.
184
185     // irq indicates which device interrupted.
186     int irq = plic_claim();
187
188     if(irq == UART0_IRQ){
189       uartintr();
190     } else if(irq == VIRTIO0_IRQ){
191       virtio_disk_intr();
192     } else if(irq){
193       printf("unexpected interrupt irq=%d\n", irq);
194     }
195
196     // the PLIC allows each device to raise at most one
197     // interrupt at a time; tell the PLIC the device is
198     // now allowed to interrupt again.
199     if(irq)
200       plic_complete(irq);
201
202     return 1;
203   } else if(scause == 0x8000000000000001L){
204     // software interrupt from a machine-mode timer interrupt,
205     // forwarded by timervec in kernelvec.S.
206
207     if(cpuid() == 0){
208       clockintr();
209     }
210
211     // acknowledge the software interrupt by clearing
212     // the SSIP bit in sip.
213     w_sip(r_sip() & ~2);
214
215     return 2;
216   } else {
217     return 0;
218   }
219 }
```

Check if its an interrupt

Check if its an external interrupt

Read more information from the PLIC and identify which device caused the interrupt. Call the corresponding interrupt handler

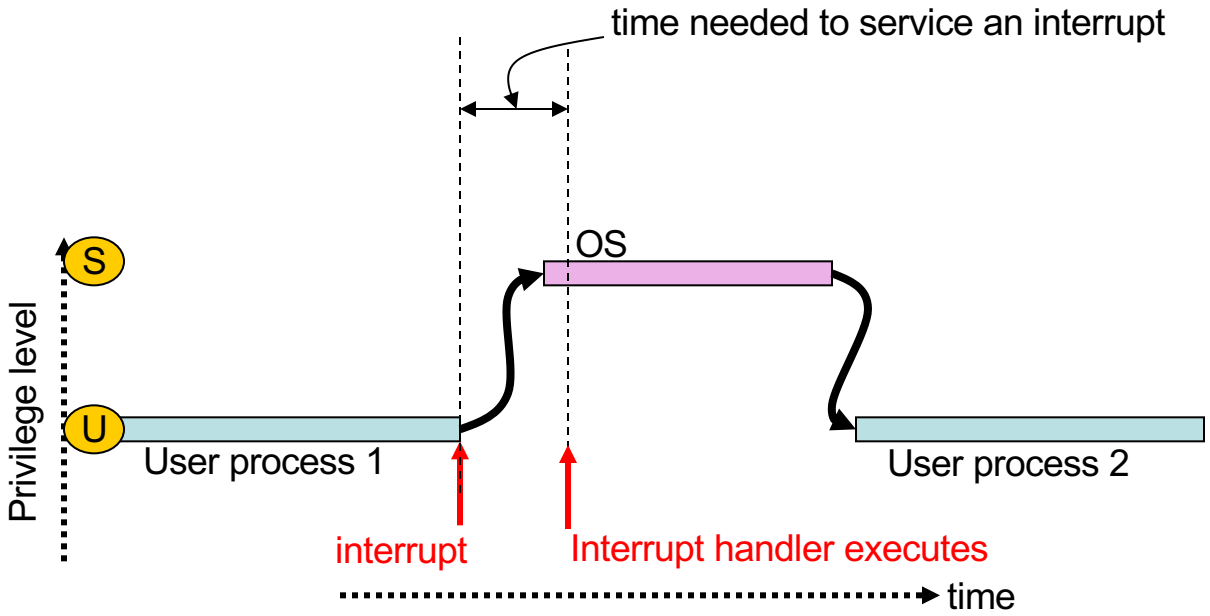
Is it a software interrupt from the machine mode

Invoked by usertrap and kerneltrap functions to identify what caused the interrupt

Interrupt Handlers

- Typical Interrupt Handler
  - Save additional CPU context (written in assembly)
    (done by alltraps in xv6)
  - Process interrupt (communicate with I/O devices)
  - Invoke kernel scheduler
  - Restore CPU context and return (written in assembly)

Interrupt Latency

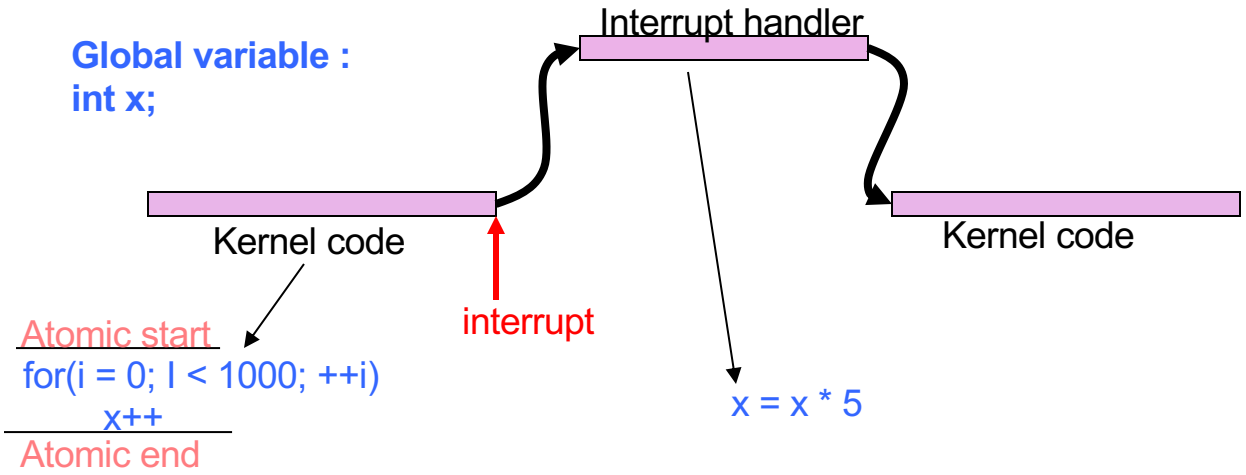


Interrupt latency can be significant

Importance of Interrupt Latency

- Real time systems
  - OS should ‘guarantee’ interrupt latency is less than a specified value
- Minimum Interrupt Latency
  - Mostly due to the interrupt controller
- Maximum Interrupt Latency
  - Due to the OS
  - Occurs when interrupt handler cannot be serviced immediately
    - Eg. when OS executing atomic operations, interrupt handler would need to wait till completion of atomic operations.

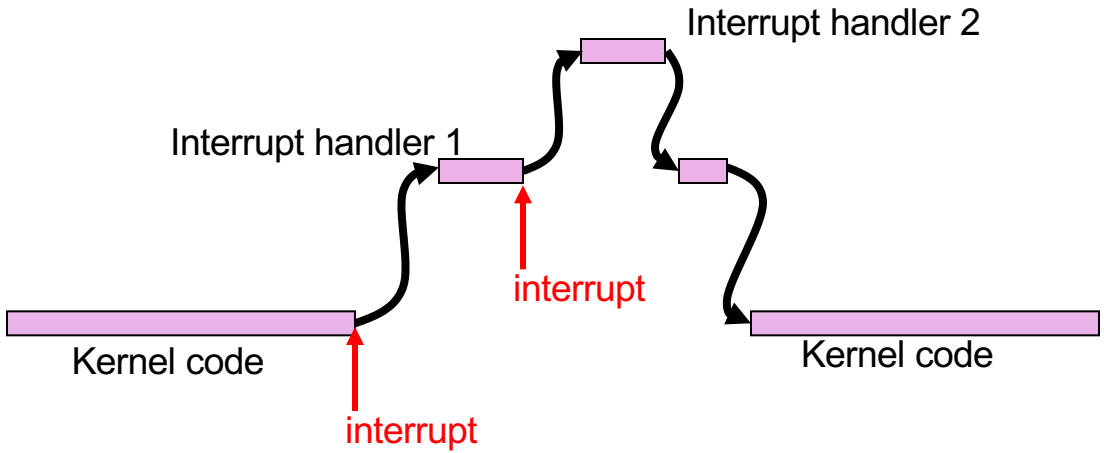
Atomic Operations



Value of x depends on whether an interrupt occurred or not!

Solution : make the part of code atomic (i.e. disable interrupts while executing this code)

Nested Interrupts



- Typically interrupts disabled until handler executes
  - This reduces system responsiveness
- To improve responsiveness, enable Interrupts within handlers
  - This often causes nested interrupts
  - Makes system more responsive but difficult to develop and validate
- **Linux Interrupt handler approach:** design interrupt handlers to be small so that nested interrupts are less likely
- RISC V supports 2 levels of interrupts

Small Interrupt Handlers

- Do as little as possible in the interrupt handler
  - Often just queue a work item or set a flag
- Defer non-critical actions till later

Top and Bottom Half Technique (Linux)

- Top half : do minimum work and return from interrupt handler
  - Saving registers
  - Unmasking other interrupts
  - Restore registers and return to previous context
- Bottom half : deferred processing
  - eg. Workqueue
  - Can be interrupted

Timers in RISC V

- **mtime**
  - 64-bit register the increments at a constant frequency
- **mtimecmp**
  - 64-bit register that is used to compare with mtime.
  - Interrupt occurs if mtime > mtimecmp

Note: there is no stime and stimecmp, which means supervisor mode cannot configure timers

Workaround:

Configure timers to interrupt in machine mode.

In machine mode trigger software interrupt to supervisor mode

Timers in xv6

memlayout.h

```
// local interrupt controller, which contains the timer.
#define CLINT 0x2000000L
#define CLINT_MTIMECMP(hartid) (CLINT + 0x4000 + 8*(hartid))
#define CLINT_MTIME (CLINT + 0xBFF8) // cycles since boot.
```

Note: 1 mtime for all cores
but each core has a different mtimecmp

Timers in xv6

start.c (invoked from start from machine mode before switching to supervisor mode)

```
// set up to receive timer interrupts in machine mode,
// which arrive at timervec in kernelvec.S,
// which turns them into software interrupts for
// devintr() in trap.c.
void
timerinit()
{
  // each CPU has a separate source of timer interrupts.
  int id = r_mhartid();

  // ask the CLINT for a timer interrupt.
  int interval = 1000000; // cycles; about 1/10th second in qemu.
  *(uint64*)CLINT_MTIMECMP(id) = *(uint64*)CLINT_MTIME + interval;

  // prepare information in scratch[] for timervec.
  // scratch[0..3] : space for timervec to save registers.
  // scratch[4] : address of CLINT MTIMECMP register.
  // scratch[5] : desired interval (in cycles) between timer interrupts.
  uint64 *scratch = &mscratch0[32 * id];
  scratch[4] = CLINT_MTIMECMP(id);
  scratch[5] = interval;
  w_mscratch((uint64)scratch);

  // set the machine-mode trap handler.
  w_mtvec((uint64)timervec);

  // enable machine-mode interrupts.
  w_mstatus(r_mstatus() | MSTATUS_MIE);

  // enable machine-mode timer interrupts.
  w_mie(r_mie() | MIE_MTIE);
}
```

Each hart configures its mtimecmp to interrupt after 100ms.

Set up timervec as the interrupt handler

Enable machine mode interrupts

Enable timer interrupts

Timers in xv6

timervec (in kernelvec.S)

```
 93 timervec:
 94         # start.c has set up the memory that mscratch points to:
 95         # scratch[0,8,16] : register save area.
 96         # scratch[32] : address of CLINT's MTIMECMP register.
 97         # scratch[40] : desired interval between interrupts.
 98
 99         csrrw a0, mscratch, a0
100         sd a1, 0(a0)
101         sd a2, 8(a0)
102         sd a3, 16(a0)
103
104         # schedule the next timer interrupt
105         # by adding interval to mtimecmp.
106         ld a1, 32(a0) # CLINT_MTIMECMP(hart)
107         ld a2, 40(a0) # interval
108         ld a3, 0(a1)
109         add a3, a3, a2
110         sd a3, 0(a1)
111
112         # raise a supervisor software interrupt.
113         li a1, 2
114         csrw sip, a1
115
116         ld a3, 16(a0)
117         ld a2, 8(a0)
118         ld a1, 0(a0)
119         csrrw a0, mscratch, a0
120
121         mret
```

1

2

Does two important things:

1. Set the timecmp for the next interrupt

2. Raise the supervisor software interrupt

Timer Interrupts

trap.c

```
176 int
177 devintr()
178 {
179   uint64 scause = r_scause();
180
181   if((scause & 0x8000000000000000L) &&
182      (scause & 0xff) == 9){
183     // this is a supervisor external interrupt, via PLIC.
184
185     // irq indicates which device interrupted.
186     int irq = plic_claim();
187
188     if(irq == UART0_IRQ){
189       uartintr();
190     } else if(irq == VIRTIO0_IRQ){
191       virtio_disk_intr();
192     } else if(irq){
193       printf("unexpected interrupt irq=%d\n", irq);
194     }
195
196     // the PLIC allows each device to raise at most one
197     // interrupt at a time; tell the PLIC the device is
198     // now allowed to interrupt again.
199     if(irq)
200       plic_complete(irq);
201
202     return 1;
203   } else if(scause == 0x8000000000000001L){
204     // software interrupt from a machine-mode timer interrupt,
205     // forwarded by timervec in kernelvec.S.
206
207     if(cpuid() == 0){
208       clockintr();
209     }
210
211     // acknowledge the software interrupt by clearing
212     // the SSIP bit in sip.
213     w_sip(r_sip() & ~2);
214
215     return 2;
216   } else {
217     return 0;
218   }
219 }
```

Check if its an interrupt

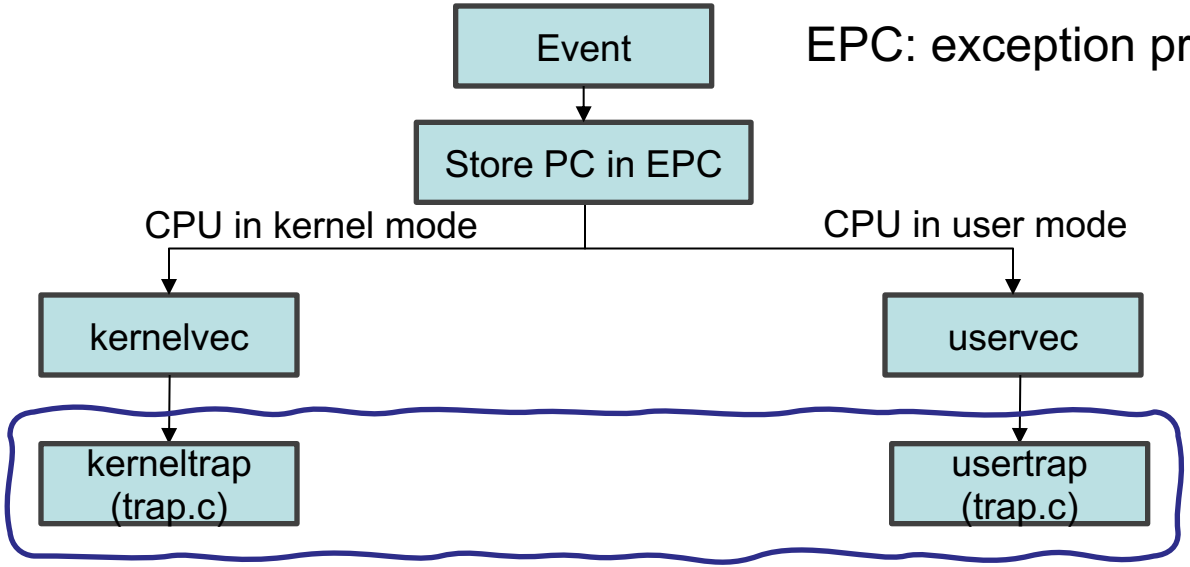
Check if its an external interrupt

Read more information from the PLIC and identify which device caused the interrupt. Call the corresponding interrupt handler

Is it a software interrupt from the machine mode

Invoked by usertrap and kerneltrap functions to identify what caused the interrupt

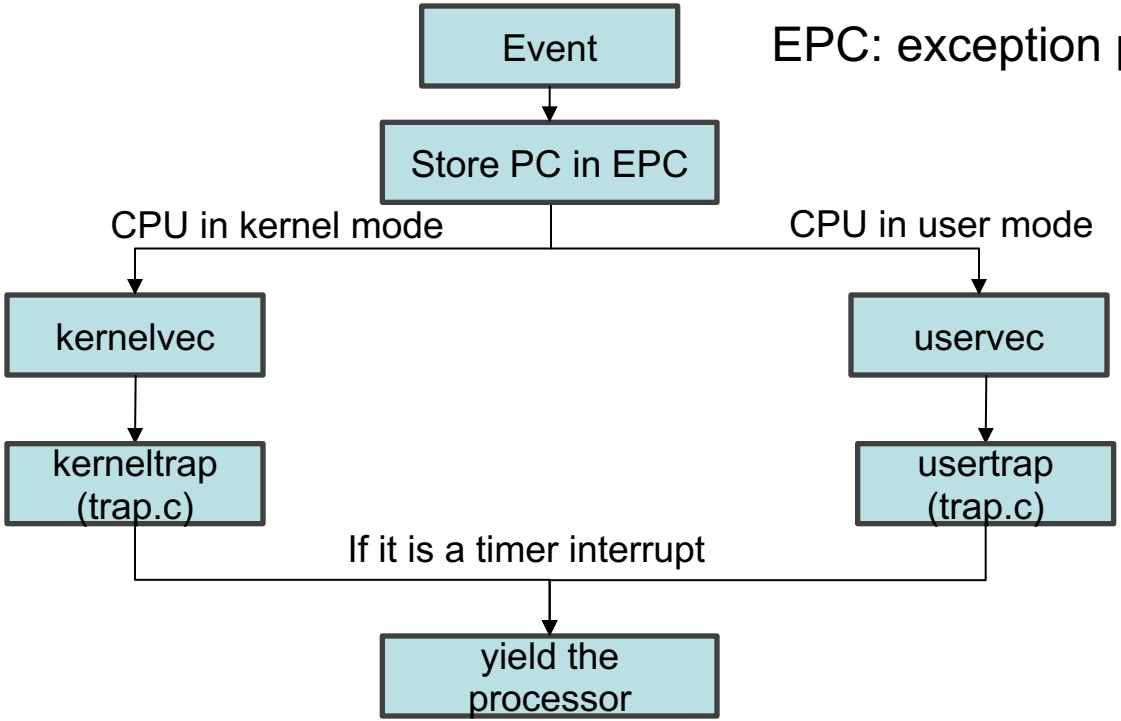
Timer Interrupts



EPC: exception program counter

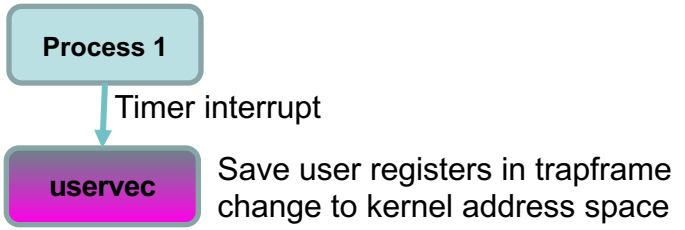
Use function
devintr to identify cause of interrupt

Timer Interrupts



EPC: exception program counter

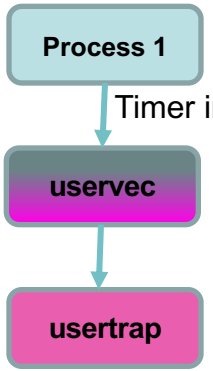
Context Switching



trampoline.S

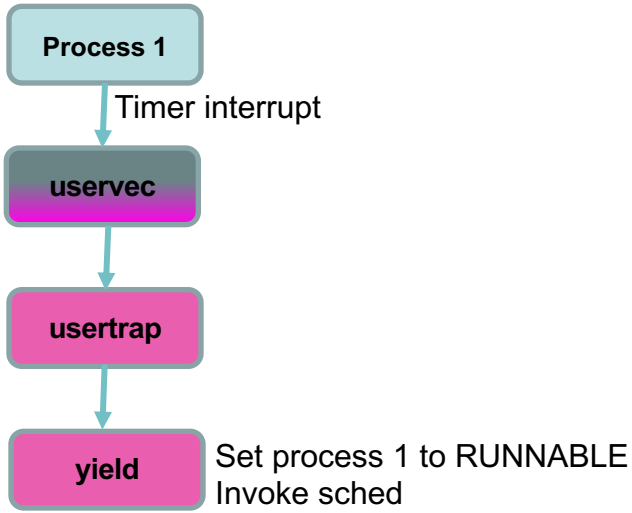
```
16 uservec:
17         #
18         # trap.c sets stvec to point here, so
19         # traps from user space start here,
20         # in supervisor mode, but with a
21         # user page table.
22         #
23         # sscratch points to where the process's p->tf is
24         # mapped into user space, at TRAPFRAME.
25         #
26
27         # swap a0 and sscratch
28         # so that a0 is TRAPFRAME
29         csrrw a0, sscratch, a0
30
31         # save the user registers in TRAPFRAME
32         sd ra, 40(a0)
33         sd sp, 48(a0)
34         sd gp, 56(a0)
35         sd tp, 64(a0)
36         sd t0, 72(a0)
37         sd t1, 80(a0)
38         sd t2, 88(a0)
39         sd s0, 96(a0)
40         sd s1, 104(a0)
41         sd a1, 120(a0)
42         sd a2, 128(a0)
43         sd a3, 136(a0)
44         sd a4, 144(a0)
45         sd a5, 152(a0)
46         sd a6, 160(a0)
47         sd a7, 168(a0)
48         sd s2, 176(a0)
49         sd s3, 184(a0)
50         sd s4, 192(a0)
51         sd s5, 200(a0)
52         sd s6, 208(a0)
53         sd s7, 216(a0)
54         sd s8, 224(a0)
55         sd s9, 232(a0)
```

Context Switching



Identify cause of the interrupt. If the interrupt happens to be a timer interrupt invoke yield

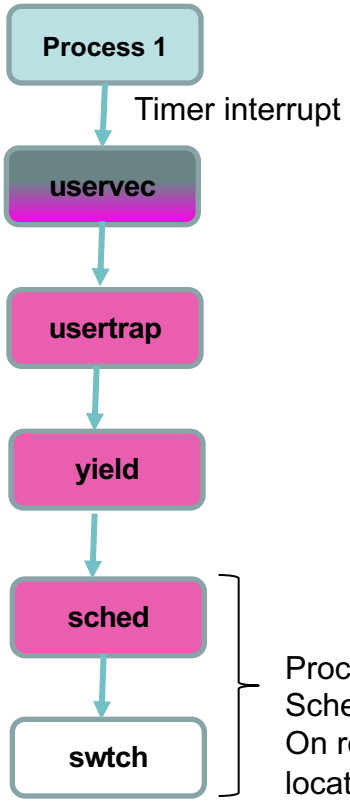
Context Switching



proc.c

```
499 void
500 yield(void)
501 {
502   struct proc *p = myproc();
503   acquire(&p->lock);
504   p->state = RUNNABLE;
505   sched();
506   release(&p->lock);
507 }
```

Context Switching



Process context saved in proc struct;
Scheduler context loaded from mycpu()->scheduler
On return from swtch, CPU continues executing from the location of the saved scheduler's context

proc.c

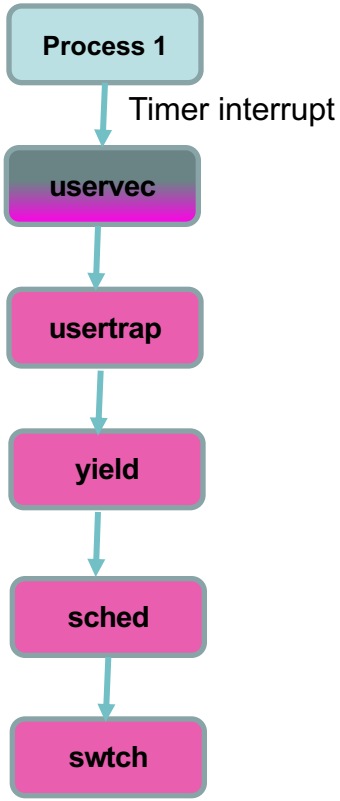
```
478 void
479 sched(void)
480 {
481   int intena;
482   struct proc *p = myproc();
483
484   if(!holding(&p->lock))
485     panic("sched p->lock");
486   if(mycpu()->noff != 1)
487     panic("sched locks");
488   if(p->state == RUNNING)
489     panic("sched running");
490   if(intr_get())
491     panic("sched interruptible");
492
493   intena = mycpu()->intena;
494   swtch(&p->context, &mycpu()->scheduler);
495   mycpu()->intena = intena;
496 }
```

a0: &p->context
a1: &mycpu()->scheduler

swtch.c

```
 8 .globl swtch
 9 swtch:
10         sd ra, 0(a0)
11         sd sp, 8(a0)
12         sd s0, 16(a0)
13         sd s1, 24(a0)
14         sd s2, 32(a0)
15         sd s3, 40(a0)
16         sd s4, 48(a0)
17         sd s5, 56(a0)
18         sd s6, 64(a0)
19         sd s7, 72(a0)
20         sd s8, 80(a0)
21         sd s9, 88(a0)
22         sd s10, 96(a0)
23         sd s11, 104(a0)
24
25         ld ra, 0(a1)
26         ld sp, 8(a1)
27         ld s0, 16(a1)
28         ld s1, 24(a1)
29         ld s2, 32(a1)
30         ld s3, 40(a1)
31         ld s4, 48(a1)
32         ld s5, 56(a1)
33         ld s6, 64(a1)
34         ld s7, 72(a1)
35         ld s8, 80(a1)
36         ld s9, 88(a1)
37         ld s10, 96(a1)
38         ld s11, 104(a1)
39
40         ret
```

Context



Context saves registers and permits to continue executing at a later time by reloading the registers.

Process 1 context

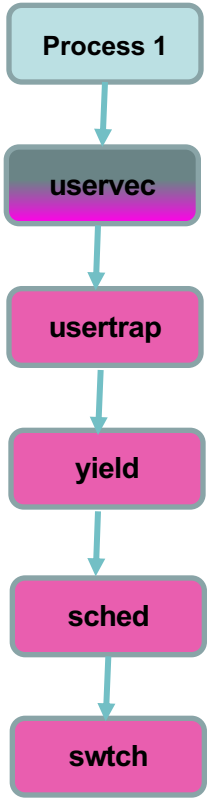
Process 2 context

Scheduler context

```
// Saved registers
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

Context Switching



proc.c

```
478 void
479 sched(void)
480 {
481   int intena;
482   struct proc *p = myproc();
483
484   if(!holding(&p->lock))
485     panic("sched p->lock");
486   if(mycpu()->noff != 1)
487     panic("sched locks");
488   if(p->state == RUNNING)
489     panic("sched running");
490   if(intr_get())
491     panic("sched interruptible");
492
493   intena = mycpu()->intena;
494   swtch(&p->context, &mycpu()->scheduler);
495   mycpu()->intena = intena;
496 }
```

Process 1 context — proc.c:494

Scheduler context — proc.c:460

Assume scheduler context is this for now

Process context saved in proc.context (proc.c:494)
Scheduler context loaded from mycpu()->scheduler
The current register has context proc.c:494

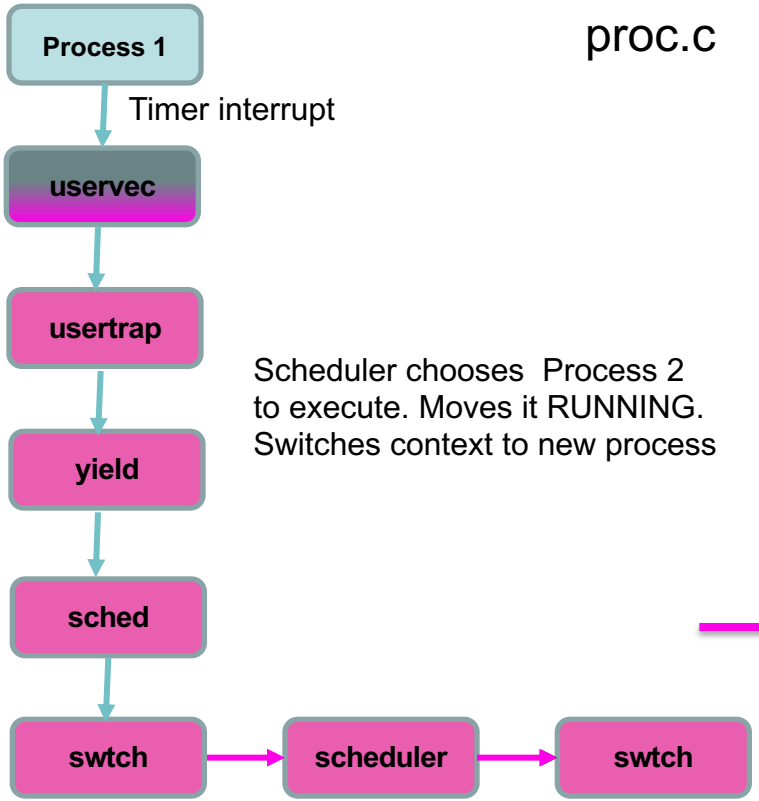
Context Switching

Process 1 context — proc.c:494

Process 2 context — proc.c:494

Scheduler context — proc.c:460

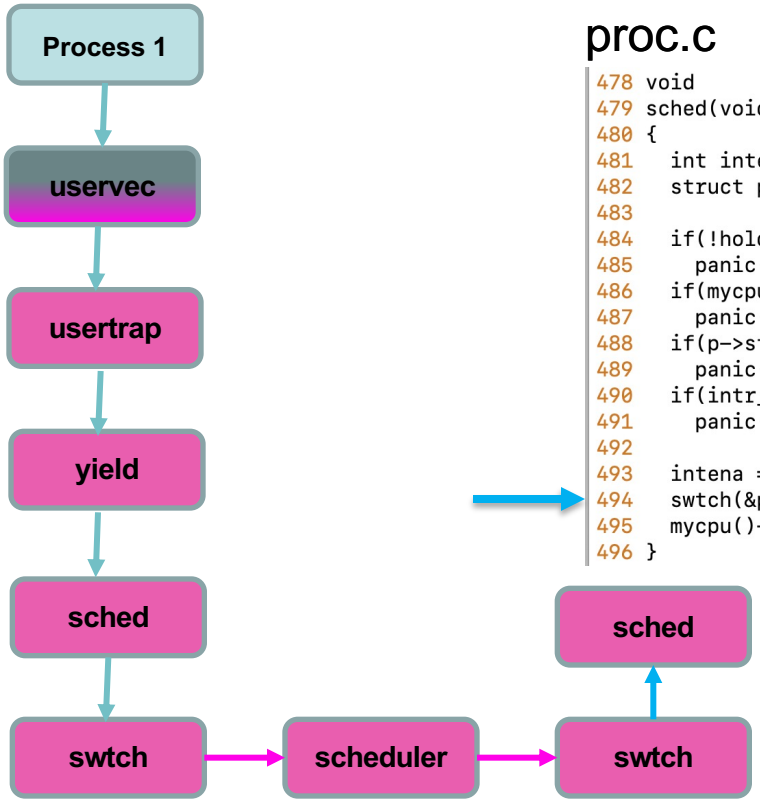
proc.c



Scheduler chooses Process 2 to execute. Moves it RUNNING. Switches context to new process

```
441 void
442 scheduler(void)
443 {
444   struct proc *p;
445   struct cpu *c = mycpu();
446
447   c->proc = 0;
448   for(;;){
449     // Avoid deadlock by ensuring that devices can interrupt.
450     intr_on();
451
452     for(p = proc; p < &proc[NPROC]; p++) {
453       acquire(&p->lock);
454       if(p->state == RUNNABLE) {
455         // Switch to chosen process.  It is the process's job
456         // to release its lock and then reacquire it
457         // before jumping back to us.
458         p->state = RUNNING;
459         c->proc = p;
460         swtch(&c->scheduler, &p->context);
461
462         // Process is done running for now.
463         // It should have changed its p->state before coming back.
464         c->proc = 0;
465       }
466       release(&p->lock);
467     }
468   }
469 }
```

Context Switching



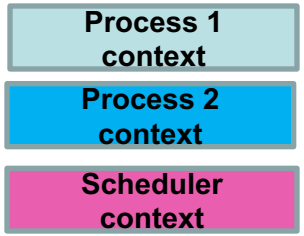
proc.c

```
478 void
479 sched(void)
480 {
481   int intena;
482   struct proc *p = myproc();
483
484   if(!holding(&p->lock))
485     panic("sched p->lock");
486   if(mycpu()->noff != 1)
487     panic("sched locks");
488   if(p->state == RUNNING)
489     panic("sched running");
490   if(intr_get())
491     panic("sched interruptible");
492
493   intena = mycpu()->intena;
494   swtch(&p->context, &mycpu()->scheduler);
495   mycpu()->intena = intena;
496 }
```

Process 1 context	proc.c:494
Process 2 context	proc.c:494
Scheduler context	proc.c:460

Process 2 continues to execute from its last known context that was at proc.c:494

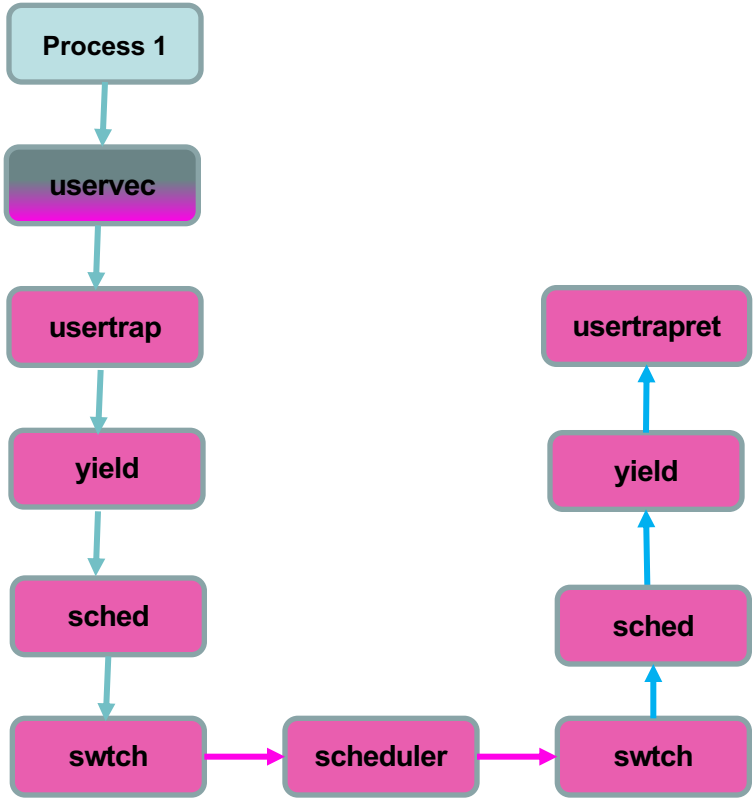
Context Switching



proc.c:494

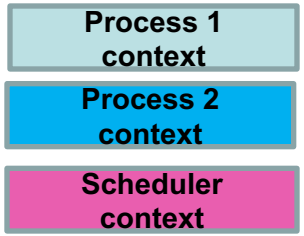
proc.c:494

proc.c:460



Setup trapframe with pointers to kernel satp, kernel stack for process2 and change from kernel address space to process address space

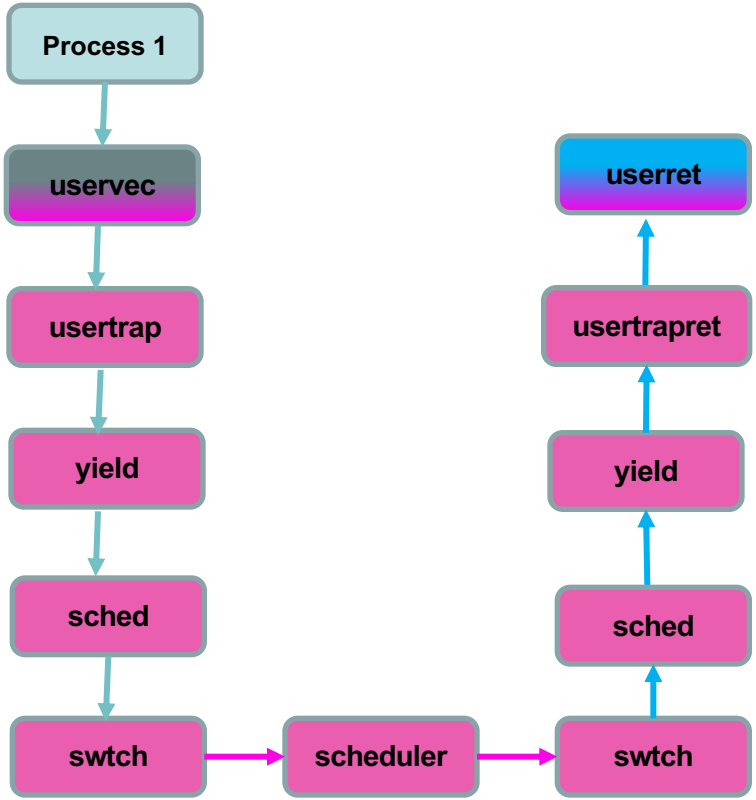
Context Switching



proc.c:494

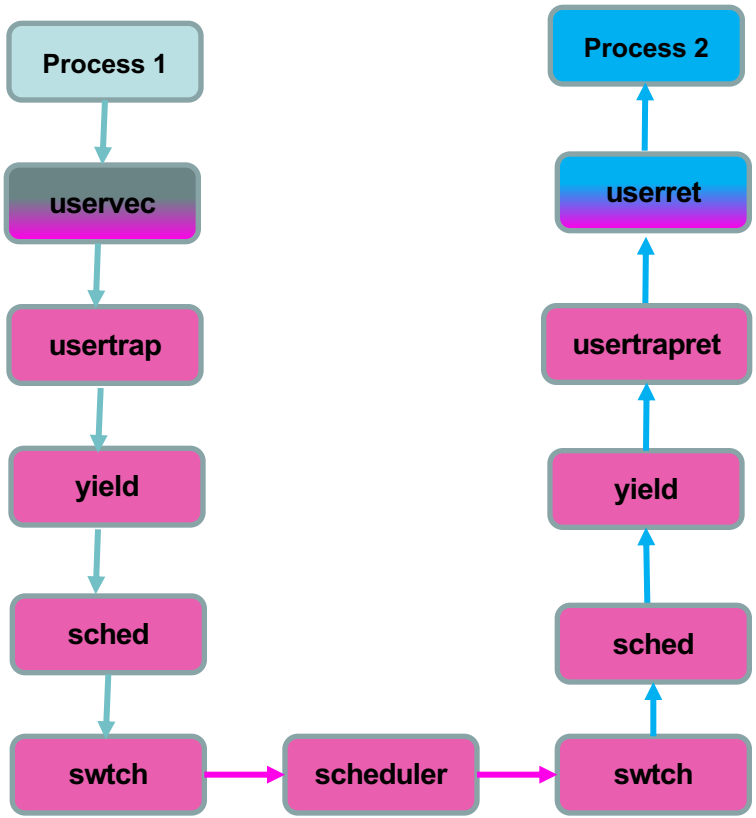
proc.c:494

proc.c:460



Executes in trampoline in the process 2's address space.
Restores the process 2's registers from the trapframe
and return from interrupt using mret

Context Switching



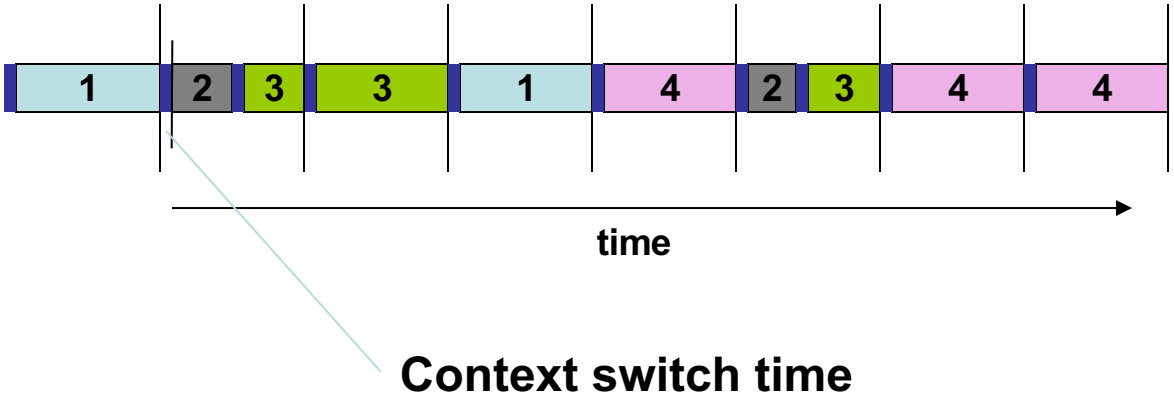
Process 1 context proc.c:494

Process 2 context proc.c:494

Scheduler context proc.c:460

Process 2 finally executes from where had previously stopped ☺☺☺

Multitasking and Context Switch



Context Switching Overheads

- Direct Factors affecting context switching time
  - Timer Interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute

- Indirect factors
  - TLB needs to be reloaded
  - Loss of cache locality (therefore more cache misses)
  - Processor pipeline flush

Context Switch Quantum

- A short quantum
  - Good because, processes need not wait long before they are scheduled in.
  - Bad because, context switch overhead increase
- A long quantum
  - Bad because processes no longer appear to execute concurrently
  - May degrade system performance
- Typically kept between 10ms to 100ms
  - xv6 programs timers to interrupt every 10ms.