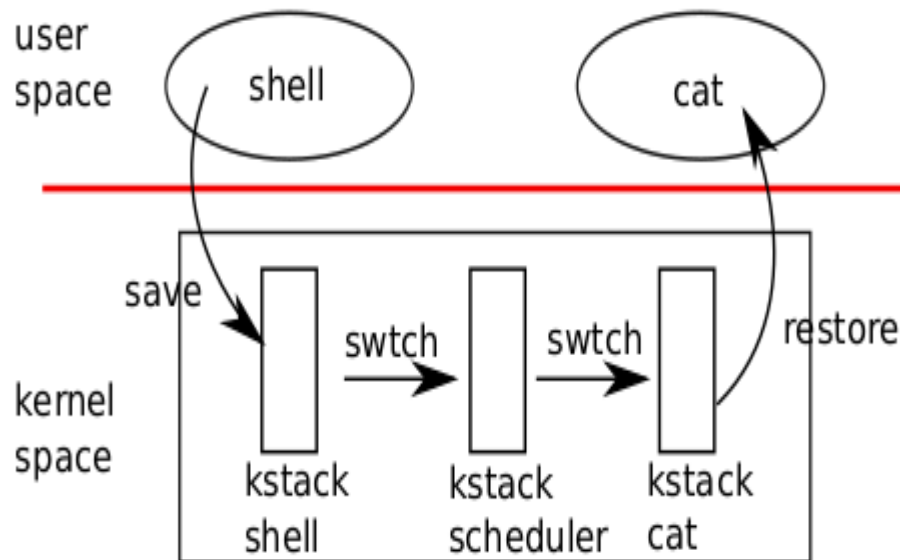# Scheduler

Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
  - **The process is chosen out of "READY" state processes**
  - **Saving of context of "earlier" process and loading of context of "next" process needs to happen**
- **Questions**
  - **What are the different scenarios in which a scheduler called ?**
  - **What are the intricacies of "passing control"**
  - **What is "context" ?**

Steps in scheduling scheduling

- **Suppose you want to switch from P1 to P2 on a timer interrupt**

- **P1 was doing**

  F() { i++; j++;}

- **P2 was doing**

  G() { x--; y++; }

- **P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2**

4 stacks need to change!

- **User stack of process -> kernel stack of process**
  - **Switch to kernel stack**
  - **The normal sequence on any interrupt !**
- **Kernel stack of process -> kernel stack of scheduler**
  - **Why?**
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- **Disable interrupts**

- **Find a RUNNABLE process. Simple round-robin!**

- **c->proc = p**

- **switchuvm(p) : Save TSS of scheduler's stack and make CR3 to point to new process pagedir**

- **p->state = RUNNING**

- **swtch(&(c->scheduler), p->context)**

swtch

```
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

scheduler()

- **swtch(&(c->scheduler), p->context)**

- **Note that when scheduler() was called, when P1 was running**

- **After call to swtch() shown above**

  - The call does NOT return!

  - The new process P2 given by 'p' starts running !

  - Let's review swtch() again

swtch(old, new)

- **The magic function in swtch.S**

- **Saves callee-save registers of old context**

- **Switches esp to new-context's stack**

- **Pop callee-save registers from new context**

  ret

  - where? in the case of first process – returns to forkret() because stack was setup like that !

  - in case of other processes, return where?
    - Return address given on kernel stack. But what's that?
    - The EIP in p->context
    - When was EIP set in p->context ?

scheduler()

- **Called from?**
  - **mpmain()**
  - **No where else!**
- **sched() is another scheduler function !**
  - **Who calls sched() ?**
    - exit() - a process exiting calls sched ()
    - yield() - a process interrupted by timer calls yield()
    - sleep() - a process going to wait calls sleep()

sched()

```
void
sched(void)
{
  int intena;

  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");

  if(mycpu()->ncli != 1)
    panic("sched locks");

  if(p->state == RUNNING)
    panic("sched running");

  if(readeflags()&FL_IF)
    panic("sched interruptible");

  intena = mycpu()->intena;

  swtch(&p->context, mycpu()->scheduler);

/*A*/  mycpu()->intena = intena;
}
```

- **get current process**
- **Error checking code (ignore as of now)**
- **get interrupt enabled status on current CPU (ignore as of now)**
- **call to swtch**
  - **Note the arguments' order**
  - **p->context first, mycpu()->scheduler second**
- **swtch() is a function call**
  - **pushes address of** */*A*/* **on stack of current process p**
  - **switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.**
  - **when was mycpu()->scheduler set? Ans: during scheduler()!**

sched() and schduler()

```
sched() {
...
  swtch(&p->context, mycpu()->scheduler); /* X */
}
```

```
scheduler(void) {
...
    swtch(&(c->scheduler), p->context);  / * Y */
}
```

- **scheduler() saves context in c->scheduler, sched() saves context in p->context**
- **after swtch() call in sched(), the control jumps to Y in scheduler**
  - Switch from process stack to scheduler's stack
- **after swtch() call in scheduler(), the control jumps to X in sched()**
  - Switch from scheduler's stack to new process's stack
- **Set of co-operating functions**

sched() and scheduler() as co-routines

- **In sched()**

  swtch(&p->context, mycpu()->scheduler);

- **In scheduler()**

  swtch(&(c->scheduler), p->context);

- **These two keep switching between processes**

- **These two functions work together to achieve scheduling**

- **Using asynchronous jumps**

- **Hence they are co-routines**

To summarize

- **On a timer interrupt during P1**
  - **trap() is called. Stack has changed from P1's user stack to P1's kernel stack**
  - **trap()->yield()**
  - **yield()->sched()**
  - **sched() -> swtch(&p->context, c->scheduler()**
  - **Stack changes to scheduler's kernel stack.**
  - **Switches to location "Y" in scheduler().**

- **Now the loop in scheduler()**
  - **calls switchkvm()**
  - **Then continues to find next process (P2) to run**
  - **Then calls switchuvm(p): changing the page table to the P2's page tables**
  - **then calls swtch(&c->scheduler, p2's->context)**
  - **Stack changes to P2's kernel stack.**
  - **P2 runs the last instruction it was was in ! Where was it?**
    - **mycpu()->intena = intena; in sched()**
    - **Then returns to the one who called sched() i.e. exit/sleep, etc**
    - **Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code**