Operating System Design and Implementation

Lecture 7: Context switch

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

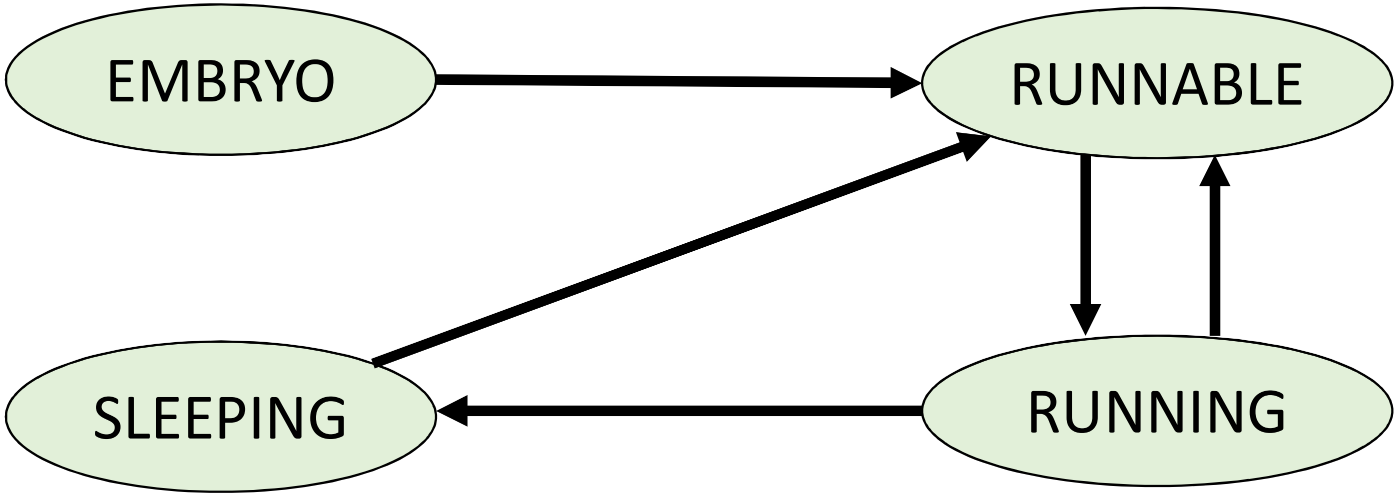
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Context switch
  - Timer interrupt
  - Process scheduler
  - overhead
- Process v.s threads
  - Sleeping and wake up

Process state

• **Process state**: specifies the state of the process



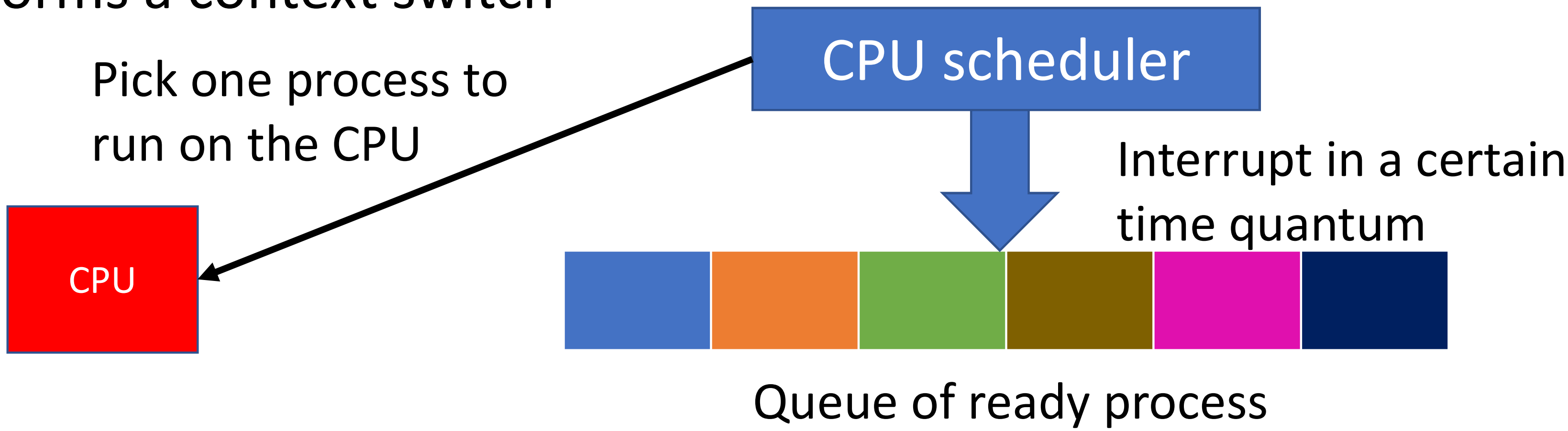
1. **EMBRYO**: The new process is currently being created
2. **RUNNABLE**: Ready to run
3. **RUNNING**: Currently executing
4. **SLEEPING**: Blocked for an I/O

Context switches

- When a process switches from **RUNNING** to **SLEEPING**
  - Due to an I/O request
- When a process switches from **RUNNING** to **RUNNABLE**
  - When an interrupt occurs
- When a process switches from **SLEEPING** to **RUNNABLE**
  - Due to I/O completion
- When a process terminates

The full picture of context switch

- Scheduler is triggered to run
  - When timer interrupt occurs
  - When running process is blocked on I/O
- Scheduler picks another process from the ready queue
  - Performs a context switch



How to switch between process ?

- How can the operating system regain control of the CPU so that it can switch between processes ?
- A cooperative approach: wait for system calls
  - When the process transfer control back to the OS ?
  - **Using system calls**:
    - most processes use **system calls** to transfer control of the CPU to the OS (e.g. **yield** system call)
  - **When processes do something illegal**
    - If an application divides by zero
    - Generate a trap to the OS, the OS will have control of the CPU again
  - The OS regains control of the CPU by waiting for a system call or an illegal operation

How to switch between process ? (cont.)

- What happens if a process ends up in an infinite loop and never makes a system call ?
- A non-cooperative approach: The OS takes control
  - The OS must inform the hardware which code to run when the timer interrupt occurs
    - **A timer interrupt:** A timer device can be programmed to raise an interrupt every so many milliseconds
    - A **pre-configured interrupt handler** in the OS runs
    - During the boot sequence, the OS must start the timer
    - The OS can feel save in that control once the timer has begun

Saving and restoring context

- How does the return-from trap instruction resume the running program correctly ?
  - The scheduler decides whether to continue running the currently-running process or switch to a different one
- **Context switch**
  - Save a few register values for the currently-executing process onto its kernel stack
    - The general purpose registers, PC, and then kernel stack pointer
  - Restoring a few for the soon-to-be-executing process from its kernel stack

Timer interrupt execution protocol

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
		1. Process A
	1. Timer interrupt 2. Save regs(A) -> k-stack(A) 3. Move to kernel mode 4. Jump to trap handler	
1. Handle the trap 2. Call switch() routine 3. save regs(A) -> proc_t(A) 4. Restore regs(B) -> proc_t(A) 5. Switch to k-stack(B) 6. Return-from-trap(into B)		
	1. Restore regs(B) <- k-stack(B) 2. Move to user mode 3. Jump to B’s PC	
		1. Process B

Process contexts

- Process context
  - Contains all information, which would allow the process to resume after a context switch
- Contexts contain 5 registers
  - edi, esi, ebx, ebp, eip
- Contexts always stored at the bottom of the process's kernel stack

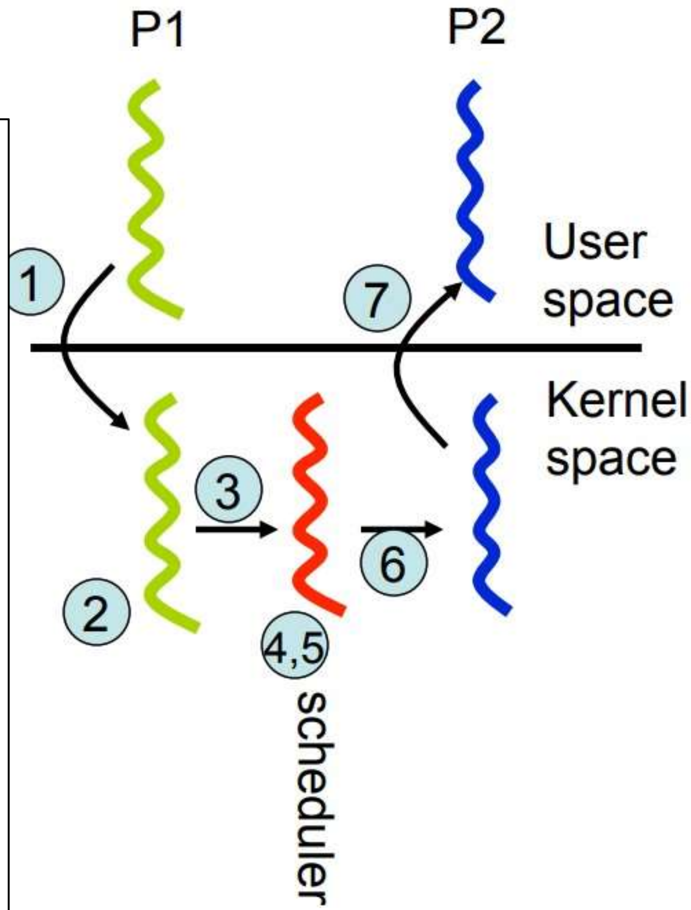
How to perform a context switch ?

- Need to save current process registers without changing them
  - Not easy !!
  - Saving state needs to execute code, which will modify registers
  - Solution: Use hardware + software … architecture dependent

1. Save current process state
2. Load state of the next process
3. Continue execution of the next process

Context switch in xv6

1. Gets triggered when any interrupt is invoked
   a. Save P1's user mode CPU context and switch from user to kernel mode
2. Handle system call or interrupt
3. Save P1's kernel CPU context and switch to scheduler CPU context
4. Select another process P2
5. Switch to P2's address space
6. Save scheduler CPU context and switch to P2's kernel CPU context
7. Switch from kernel to user mode and load P2's user-mode CPU context

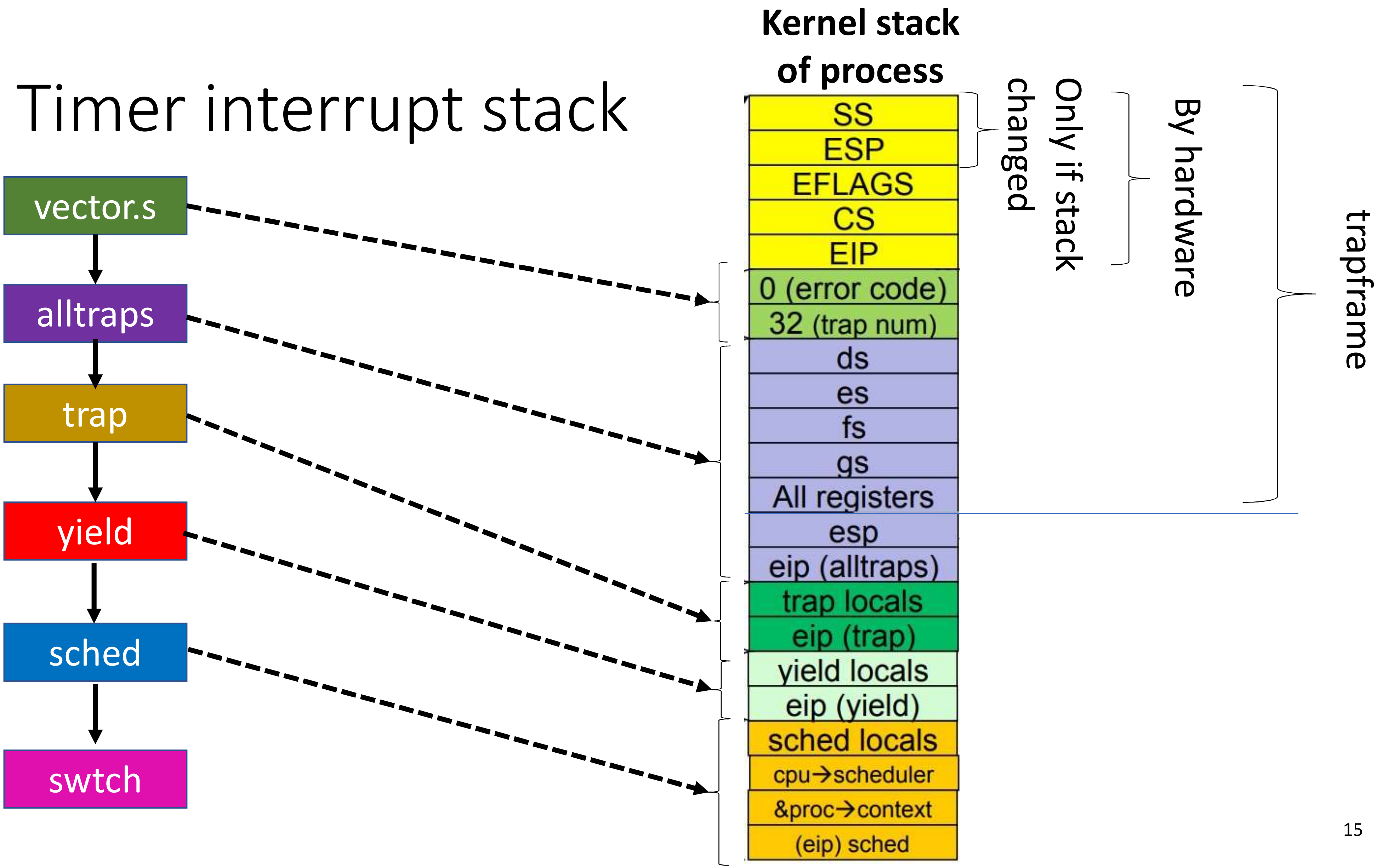


http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

The timer interrupts

- Single processor system
  - Periodic interrupt timer (PIT)
- Multi-processor systems
  - Programmable interrupt controller (LAPIC)
- Programmed to interrupt processor every 10 ms

Timer interrupt stack



trap, yield & sched

trap.c

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

proc.c

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

proc.c

```
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);
  mycpu()->intena = intena;
}
```

swtch(&proc->context, cpu->scheduler)

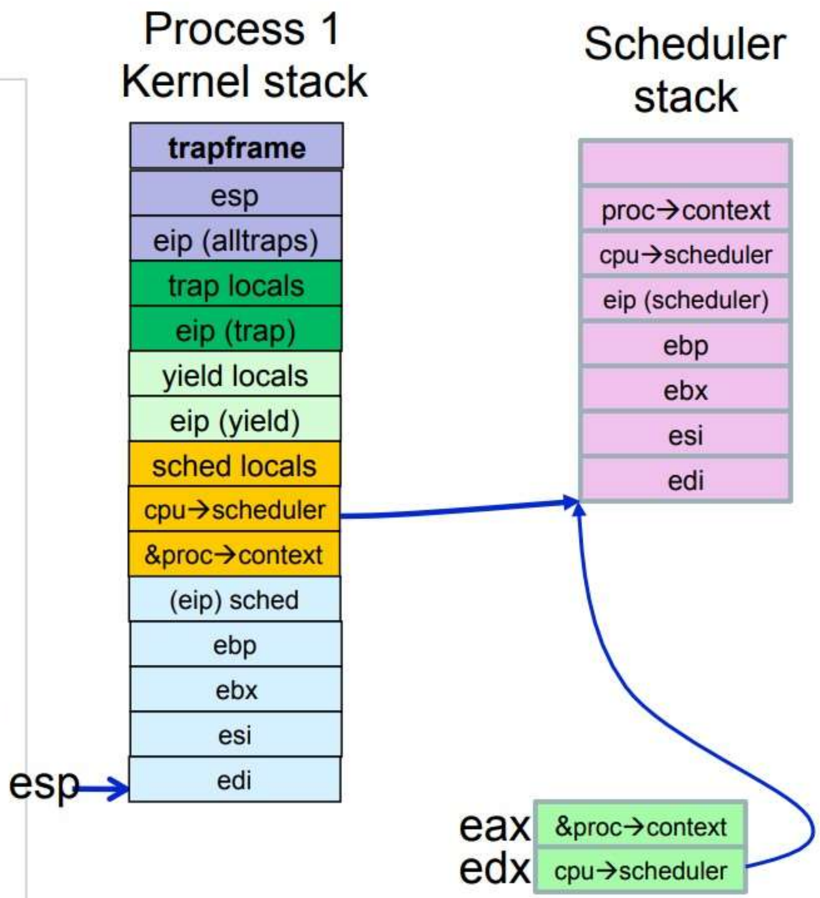
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

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

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

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

eip → pushl %edi



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

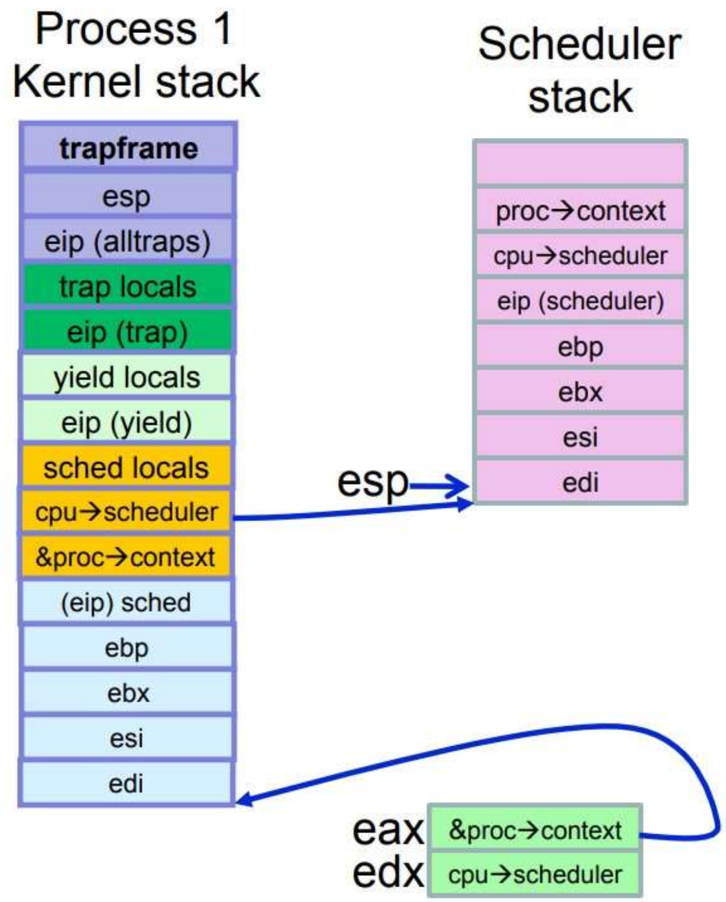
swtch(&proc->context, cpu->scheduler)

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

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

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

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



Execution in scheduler

- Switch to kvm pagetables
- Select new runnable process
- Switch to user process page tables
- swthch(&cpu->scheduler, proc->contxt)

```c
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

eip → (points to `switchkvm();`)

swtch(&proc->context, cpu->scheduler)

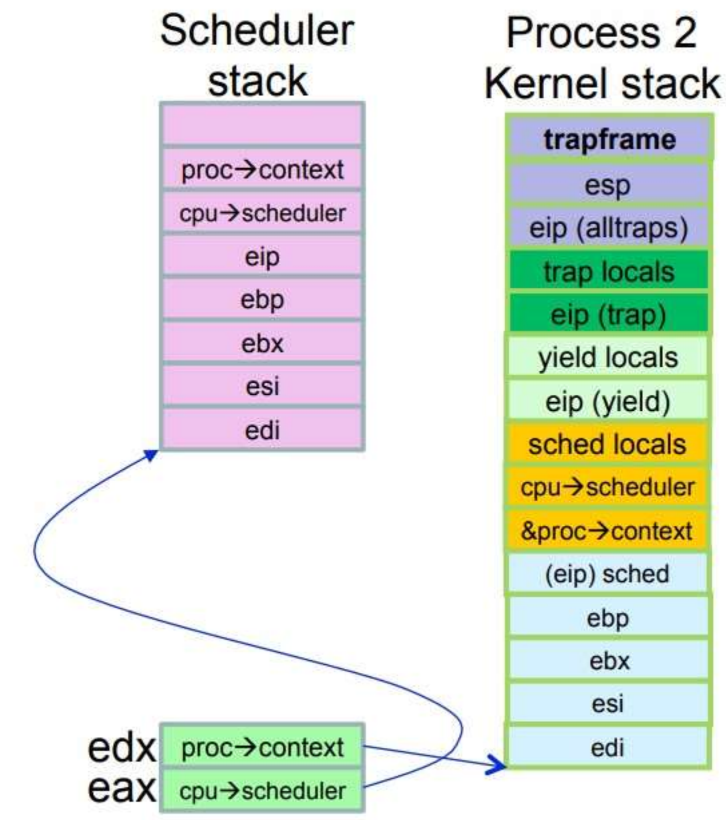
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

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

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

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

eip → movl %edx, %esp



Swtch returns to sched

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Sched in process 2’s context

- Sched returns to yield
- Yield returns to trap
- Trap returns to alltraps
- Alltraps restores user space registers of process 2 and invokes IRET

eip

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Context switch overheads

- **Direct factors**
  - Timer interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute
- **Indirect factors**
  - TLB needs to be reloaded
  - Loss of cache locality (more cache misses)
  - Processor pipeline flush

Context switch quantum

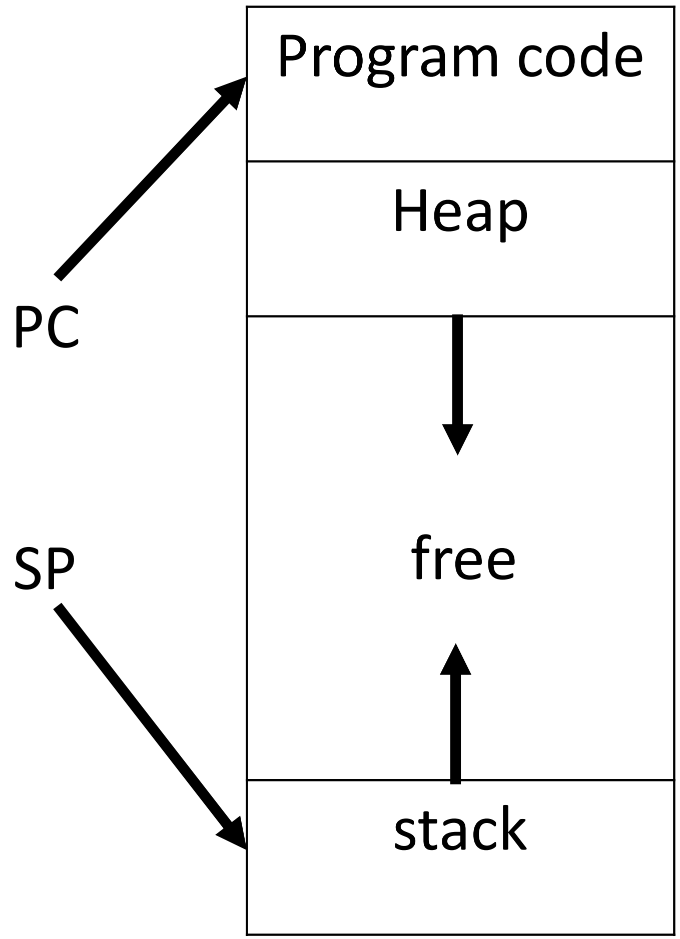
- **A short quantum**
  - **Good**, because processes need not wait long before they are scheduled in
  - **Bad**, because, context switch overhead increase
- **A long quantum**
  - **Bad**, because processes no longer appear to execute concurrently
  - May degrade system performance
- Typically kept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

How long context switches take ?

- How long does something like a context switch take? Or even a system call ?
  - Running Linux 1.3.37 on a 200-MHz P6 CPU in 1996
    - System calls took roughly 4 microseconds
    - A context switch roughly 6 microseconds
    - Will faster processors help for the reduction of system call and context switch latency ?
  - Not all operating system actions track CPU performance
    - Many OS operations are memory intensive
    - Depending on workloads, the latest and greatest processor may not speed up your OS as much as you might hope

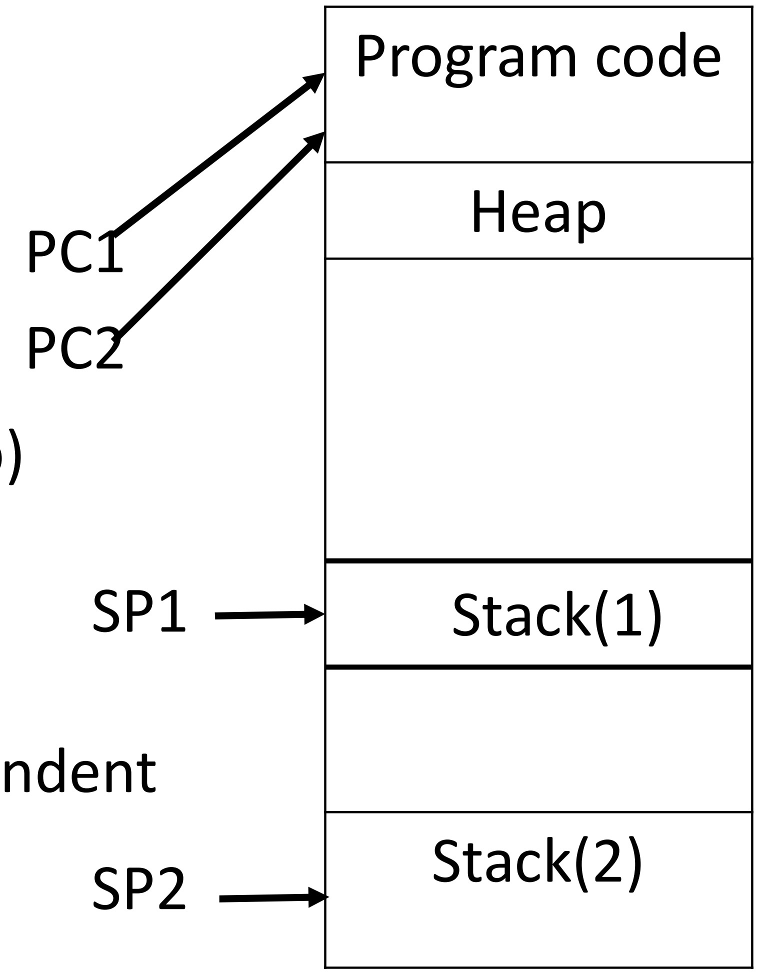
Recap: process

- So far, we have studied single threaded programs
- A process in execution
  - **Program counter (PC)**
    - Points to current instruction being run
  - **Stack pointer (SP)**
    - Points to stack frame of current function call
- However, a program can have multiple threads in execution



Multi-thread process

- A process can have multiple threads
  - Each of them executes independently
- **Threads**
  - Share the same address space(code, heap)
  - Each thread has separate PC
  - Each thread can run over different part of the program
  - Each thread has separate stack for independent function calls



Processes v.s threads

- In UNIX, a process is created using **fork()** and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - A single thread, which is the only entity known by the scheduler
- Additional threads can be created inside an existing processing, using pthread_create()
  - They run in the same address space as the initial thread of the process
  - They start executing a function passed as argument to pthread_create()

Processes v.s threads

- **Parent (P) and Child (C) process**
  - P and C do not share any memory
  - Communicate through inter-process communication (IPC)
  - Extra copies of code, data in memory
- **Threads (T1 and T2) within a process**
  - T1 and T2 share parts of the address space
  - Global variables can be used for communication
  - Small memory footprint
  - The context of a thread (PC, registers) is saved into/restored from **thread control block (TCB)**

Process and thread

- Each process has a thread of execution
  - The state of a thread (local variables, function call return address) is stored on the thread's stacks
  - Each process has two stacks: a user stack and a kernel stack

Process	Thread
Process is any in-execution program	Thread is the segment of a process
Process is isolated	Thread share memory
Process has its own process control block (PCB) and address space	Thread has parent's PCB, its own TCB, stack, and address space
Process takes more time for creation	Thread takes less time for creation

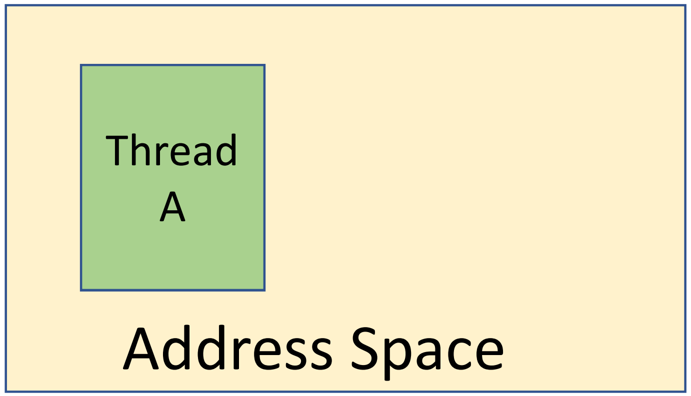
https://www.geeksforgeeks.org/difference-between-process-and-thread/

Why threads ?

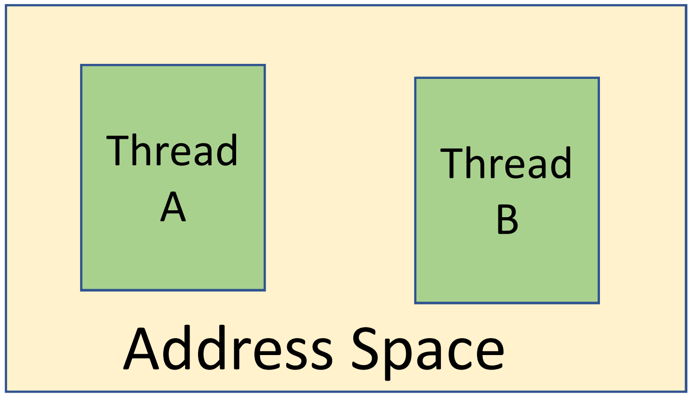
- **Parallelism**
  - Make a single process to effectively utilize multiple CPU cores
  - **Concurrency**
    - Running multiple threads/process, even on a single CPU core by interleaving their executions
    - Concurrency ensures effective use of the CPU even if no parallelism (e.g. overlapping I/O with other activities within a single program)
  - **Parallelism**
    - Running multiple threads/process in parallel over different CPU cores

Process, thread: kernel point of view

- **In kernel space**
  - Each running thread is represented by a structure of type **“struct task_struct**”
  - No difference between the initial thread of a process and all additional threads created dynamically using **pthread_create()**

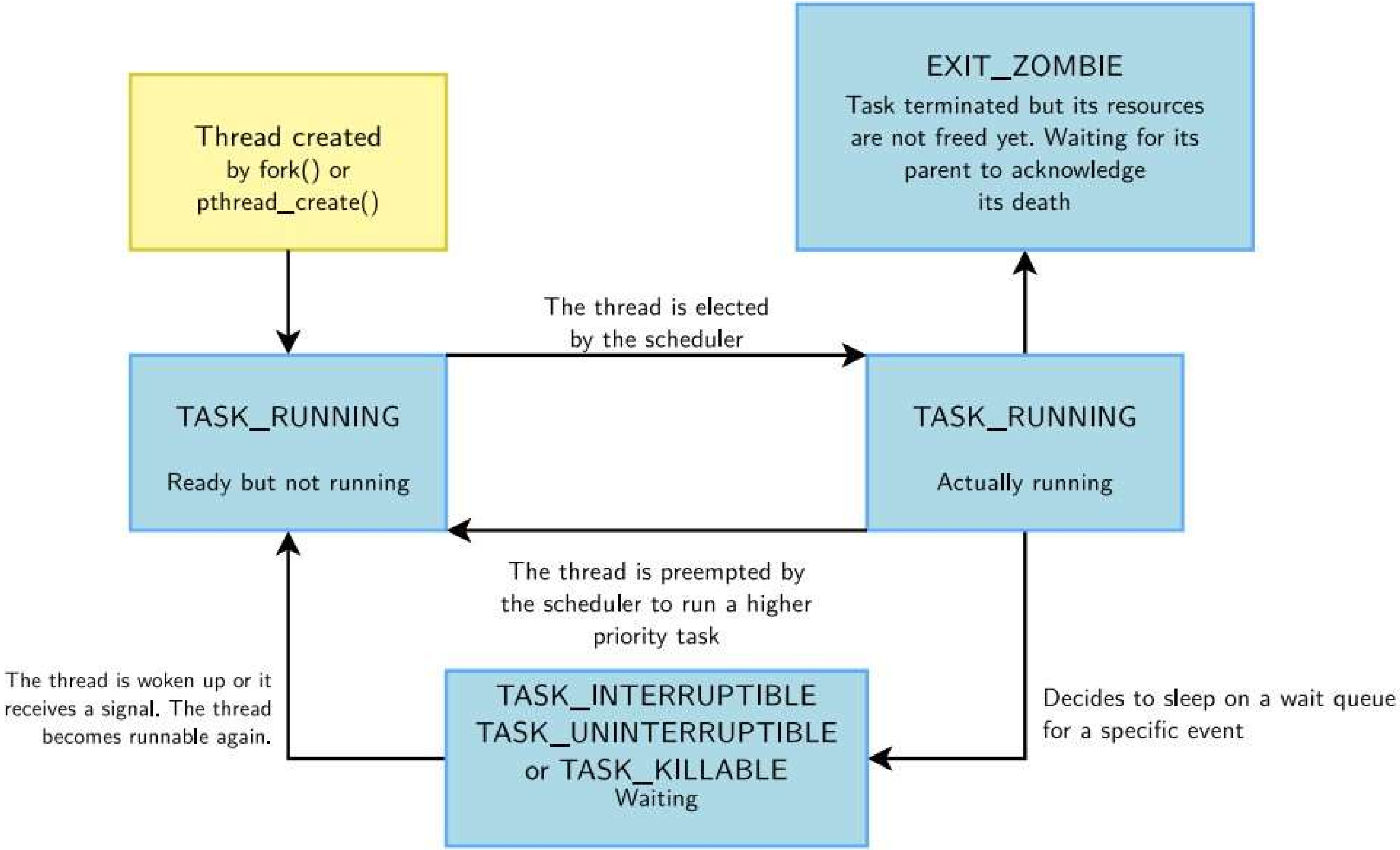


Process after fork()



Same process after pthread_create()

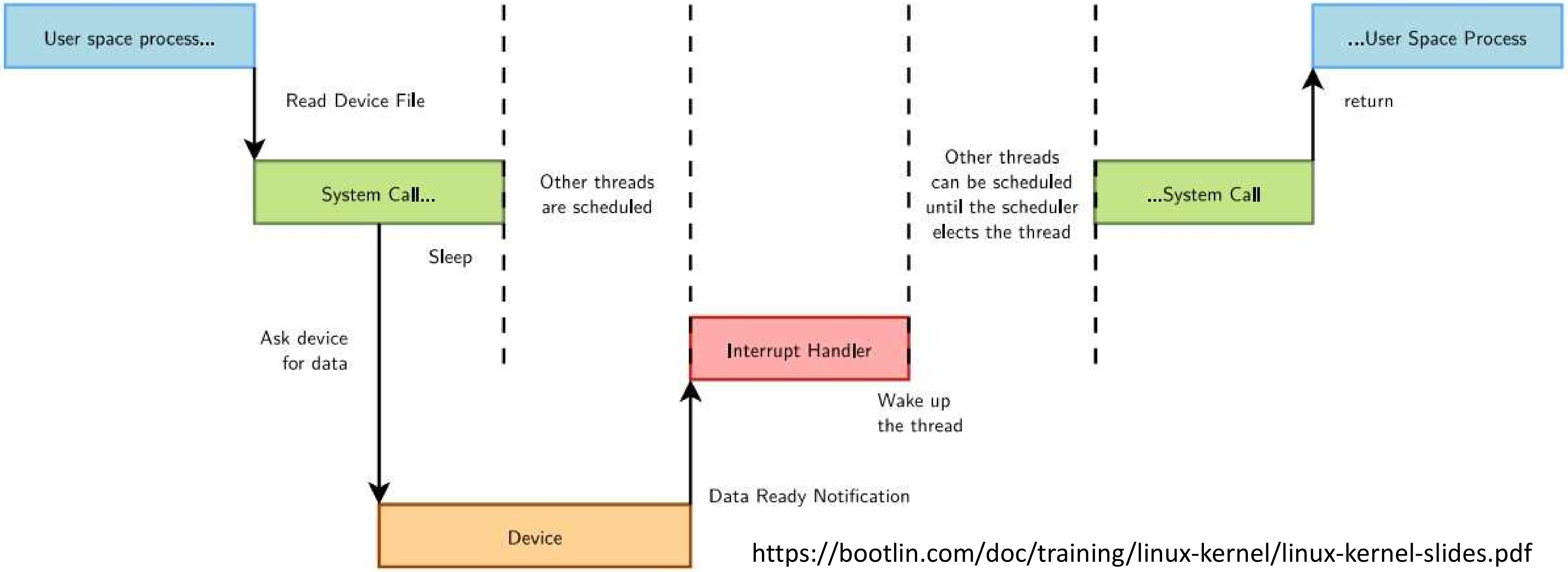
A thread life



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Sleeping

- Sleeping is needed when a process (user space or kernel space) is waiting for data



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

How to sleep with a wait queue ?

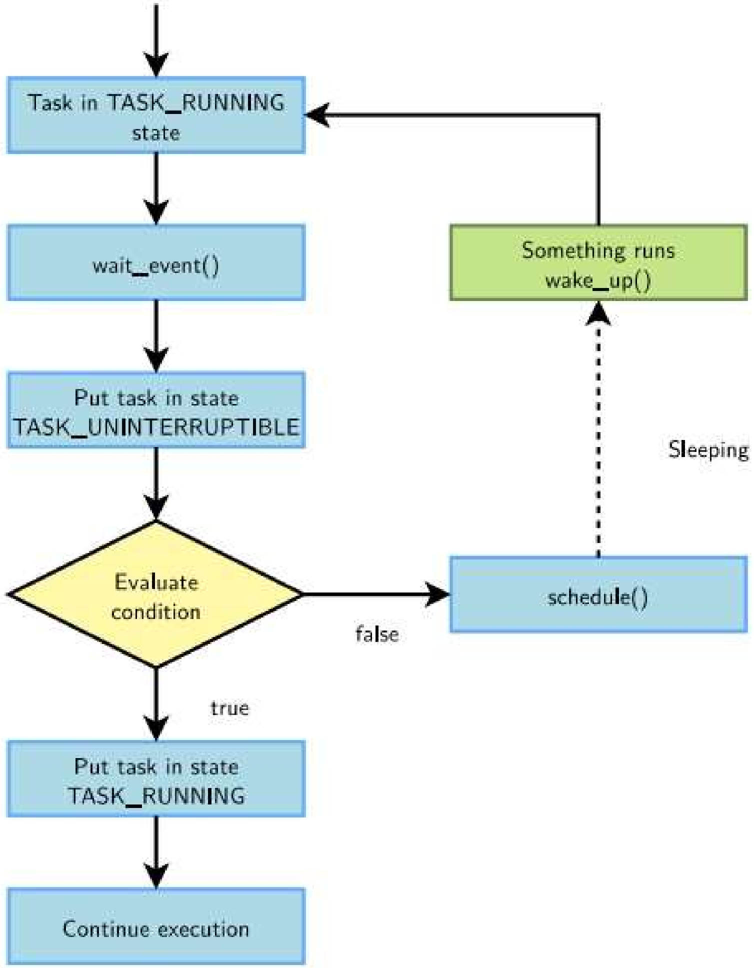
- **A wait queue**
  - stores the list of threads waiting for an event
- Several ways to make a kernel process sleep
  - void wait_event(queue, condition);
  - int wait_event_killable(queue, condition);
  - int wait_event_interruptible(queue, condition);
  - int wait_event_timeout(queue, condition, timeout);
  - int wait_event_interruptible_timeout(queue, condition, timeout);

Waking up!

- Typically done by interrupt handlers when data sleeping processes are waiting for become available
  - **wake_up(&queue)**
    - Wakes up all processes in the wait queue
  - **wake_up_interruptible(&queue);**
    - Wakes up all processes waiting in an interruptible sleep on the given queue

Waking up -- implementation

- **wait_event(queue, cond);**
  - The process is put in the TASK_UNINTERRUPTIBLE state
- **wake_up(&queue);**
  - All processes waiting in queue are woken up
  - They get scheduled later and have the opportunity to evaluate the condition again
  - Go back to sleep if it is not met



Summary

- Processes contains **process states** including running, ready to run, and block
- OS can switch from running the current process to a different one known as **context switch**
- OS uses **timer interrupt** to ensure the user program does not run forever
- **Process** means a program is in execution, whereas **thread** means a segment of a process.