CSCE 351
Operating System Kernels

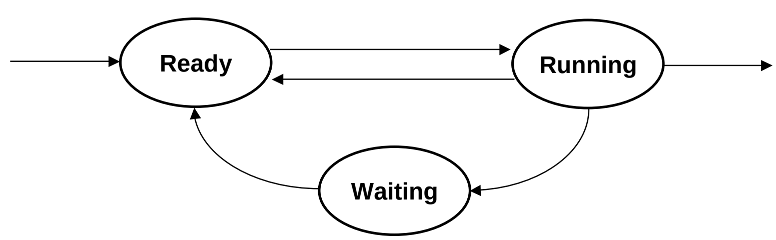
Processes, Context Switches and Interrupts

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE351

Processes

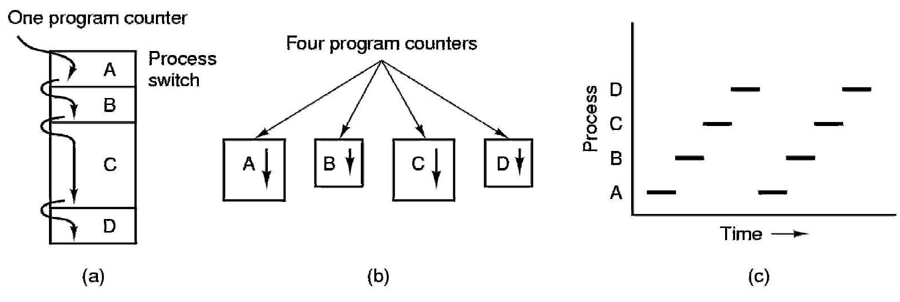
- The basic agent of work, the basic building block
- Process characterization
  - » Program code
  - » Processor/Memory state
  - » Execution state
- The state transition diagram



Process Actions

- Create and Delete
- Suspend and Resume
- Process synchronization
- Process communication

Multiprogramming



Physical v. Logical Concurrency
Why is logical concurrency useful?

- Structuring of computation
- Performance

```
process P
begin
  :
  Read(var)
  :
end P
```

```
system call Read()
begin
  StartIO(input device)
  WaitIO(interrupt)
  EndIO(input device)
  :
end Read
```

» Single process I/O

Physical v. Logical Concurrency
Performance considerations

- Multithreaded I/O

```
process P
begin
  :
  StartRead()
  <compute>
  Read(var)
  :
end P
```

```
system call StartRead()
begin
  RequestIO(input device)
end StartRead

system call Read()
begin
  SignalReader(input device)
end Read
```

```
system process Read()
begin
  loop
    WaitForRequest()
    System_Read(var)
    WaitForRequestor()
     :
  end loop
end Read
```

Process Creation Paradigms

- COBEGIN/COEND

```
cobegin
   S1 ||
   S2 ||
     :
   Sn
coend
```

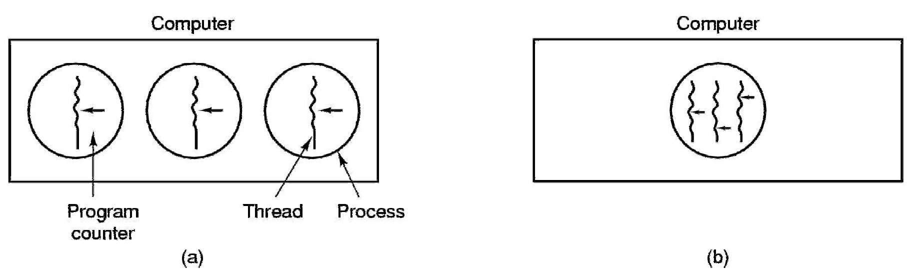
- FORK/JOIN

```
begin                procedure foo()
   :                 begin
  fork(foo)             :
   :                    :
  join(foo)          end foo
   :
end
```

- Explicit process creation

```
begin                process P
  :                  begin
  P                     :
  :                     :
end                  end P
```

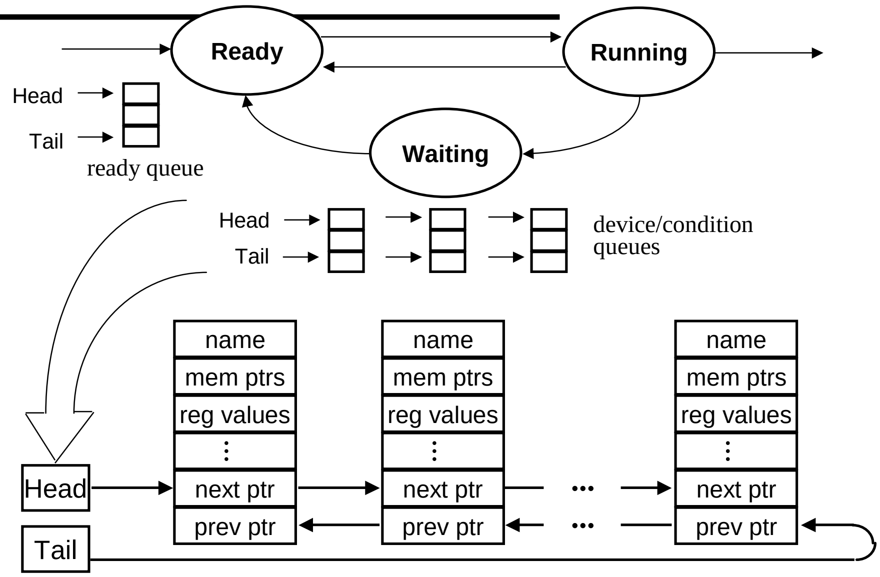

Threads



- 3 processes
  - » Each with one thread
- 1 process
  - » Three threads

Process Scheduling

Implementing and managing state transitions



Why Schedule?

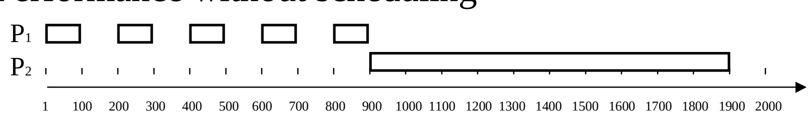
Scheduling goals

- Example: two processes execute concurrently

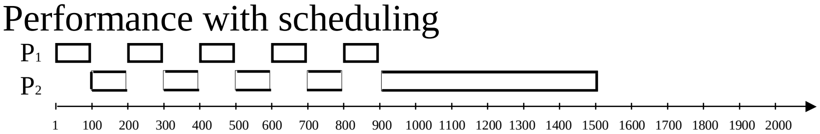
```
process P1
begin
  for i := 1 to 5 do
    <read a char>
    <process a char>
  end for
end P1
```

```
process P2
begin
  <execute for 1 sec >
end P2
```

- Performance without scheduling



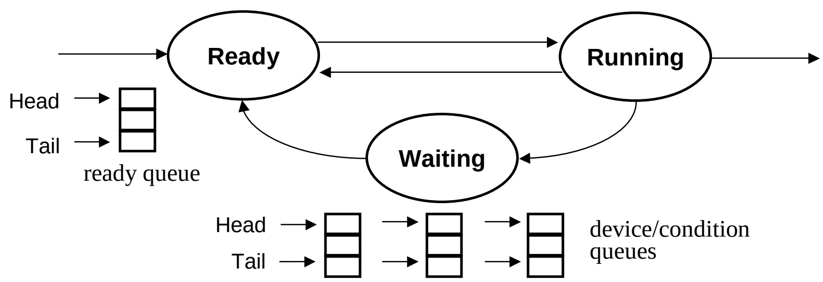
- Performance with scheduling



Types of Schedulers

- Long term schedulers
  - » adjust the level of multiprogramming through admission control
- Medium term schedulers
  - » adjust the level of multiprogramming by suspending processes
- Short term schedulers
  - » determine which process should execute next

Short Term Scheduling
When to schedule



When a process makes a transition...

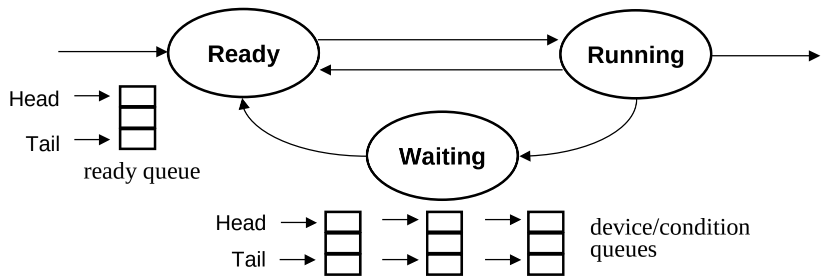
1. from *running* to *waiting*
2. from *running* to *ready*
3. from *waiting* to *ready*

(3a. a process is *created*)

4. from *running* to *terminated*

Short Term Scheduling

How to schedule — Implementing a context switch



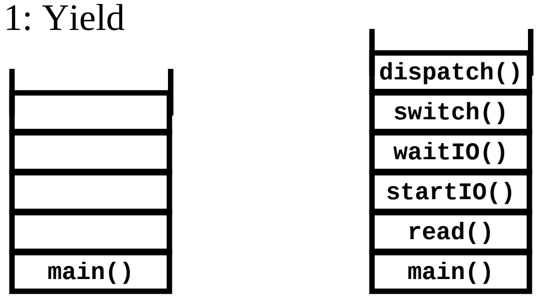

```
context_switch(queue : system_queue)
var next : process_id
begin
  DISABLE_INTS
  insert_queue(queue, runningProcess)
  next := remove_queue(readyQueue)
  dispatch(next)
  ENABLE_INTS
end context_switch
```

```
dispatch(proc : process_id)
begin
  <save memory image of runningProcess>
  <save processor state of runningProcess>
  <load memory image of proc>
  <load processor state of proc>
  runningProcess := proc
end dispatch
```


Implementing a Context Switch

Dispatching

- Case 1: Yield



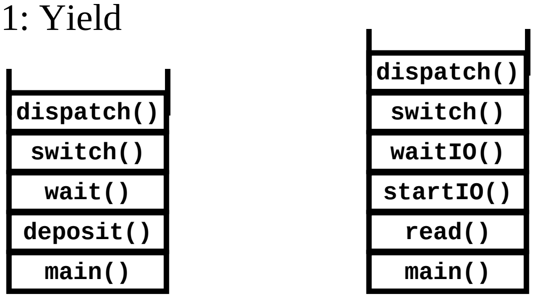
"P1" "P2: running"

P2's dispatch:

```
dispatch()
begin
  <save state of P2>
  <load state of P1>
       :
end dispatch
```

Implementing a Context Switch
Dispatching

- Case 1: Yield



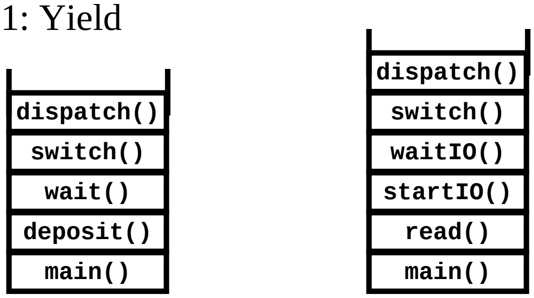
"P1: running" "P2"

P1's dispatch:

```
dispatch()
begin
   <save state of P1>
   <load state of P2>
          :
end dispatch
```


Implementing a Context Switch
Dispatching

- Case 1: Yield



"P1" "P2 : running"

P1's dispatch:

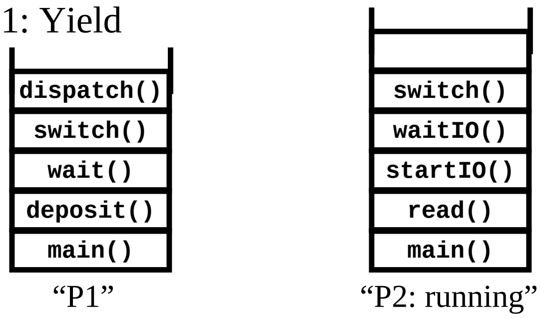
```
dispatch()
begin
   <save state of P1>
   <load state of P2>
          :
end dispatch
```

P2's dispatch:

```
dispatch()
begin
  :
  RunningProcess:= P2
end dispatch
```

Implementing a Context Switch
Dispatching

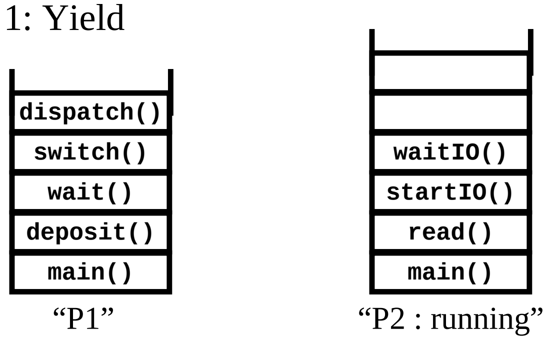
- Case 1: Yield




```
context_switch(queue : system_queue)
var next : process_id
begin
  DISABLE_INTS
  insert_queue(queue, runningProcess)
  next := remove_queue(readyQueue)
  dispatch(next)
  ENABLE_INTS
end context_switch
```

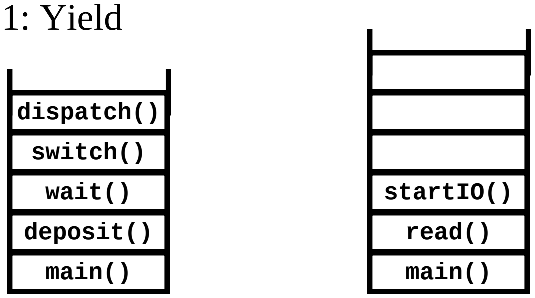

Implementing a Context Switch
Dispatching

- Case 1: Yield



Implementing a Context Switch
Dispatching

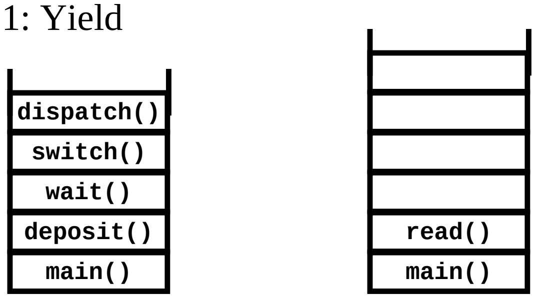
- Case 1: Yield



“P1” “P2 : running”

Implementing a Context Switch
Dispatching

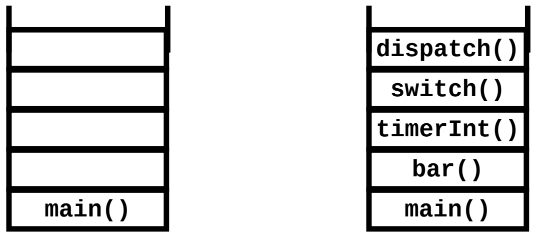
- Case 1: Yield



“P1” “P2 : running”

Implementing a Context Switch
Dispatching

- Case 2: Preemption



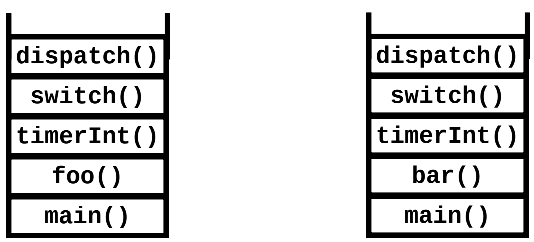
"P1" "P2: running"

P2's dispatch:

```
dispatch()
begin
   <save state of P2>
   <load state of P1>
            :
end dispatch
```


Implementing a Context Switch
Dispatching

- Case 2: Preemption



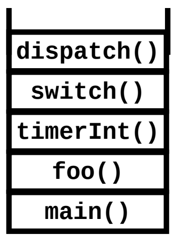
"P1: running" "P2"

P1's dispatch:

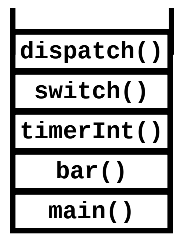
```
dispatch()
begin
   <save state of P1>
   <load state of P2>
            :
end dispatch
```

Implementing a Context Switch
Dispatching

- Case 2: Preemption



“P1”



“P2 : running”

P1’s dispatch:

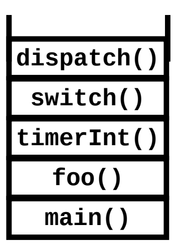
```
dispatch()
begin
   <save state of P1>
   <load state of P2>
          :
end dispatch
```

P2’s dispatch:

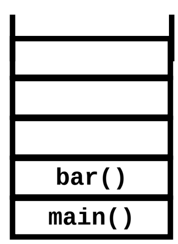
```
dispatch()
begin
  :
  RunningProcess:= P2
end dispatch
```


Implementing a Context Switch
Dispatching

- Case 2: Preemption



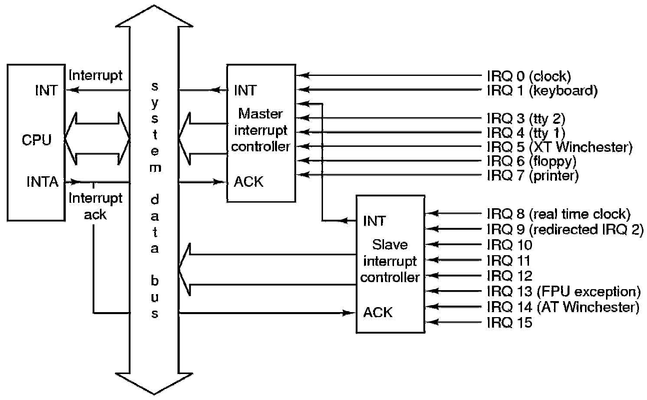
“P1”



“P2 : running”

Interrupts

- Device sends a signal to an interrupt controller
- Controller interrupts the CPU via the INT pin



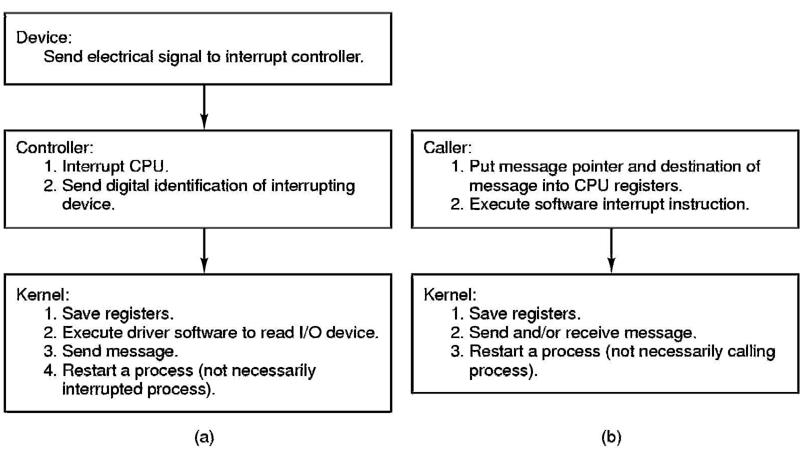
Kernel response to an Interrupt - sketch

- CPU stacks PC and other key registers
- CPU loads new PC from interrupt vector table
- Assembly language procedure saves registers
- Assembly language procedure sets up INT stack
- C ISR runs (usually reads and buffers input)
- Scheduler marks any newly ready tasks
- Scheduler decides which process will run next
- C procedure returns to the assembly code
- Assembly language procedure switches to new current process

Response to an Interrupt - details for Intel processors

- Controller interrupts the CPU via the INT pin
- CPU disables interrupts and pushes PC and other key registers onto the current process stack
- CPU signals the controller via INTA (interrupt acknowledge) signal to put interrupt number on the system data bus
- CPU reads the system data bus and uses that value as an index into the interrupt vector table to find the pointer of the interrupt handler, which is an assembly routine wrapper for the ISR (i.e., an indirect jump)
- The interrupt handler fills out the stack frame with general registers, switches to an interrupt stack and calls the C ISR
- When the ISR completes, the handler switches to a process stack frame, pops the general registers, and executes the `iretd` (return from interrupt) instruction to pop the remaining instructions in the stack frame to restore the system state

Interrupts vs System Calls



Interrupt   System Call