CSCE 451/851
Operating Systems Principles

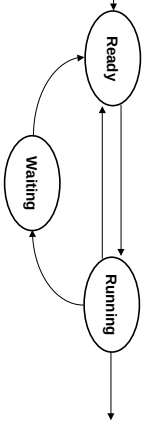
Processes

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE451

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

3

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

4

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
```

5

Process Creation Paradigms

- COBEGIN/COEND

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

- FORK/JOIN

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

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

- Explicit process creation

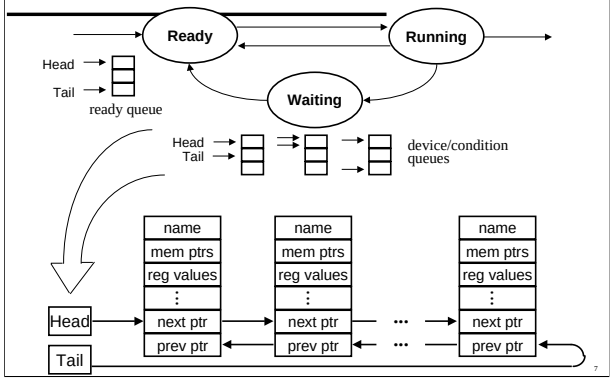
```
begin
  :
  P
  :
end
```

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

6

Process Scheduling

Implementing and managing state transitions



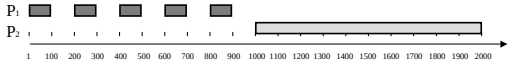
Why Schedule?

Scheduling goals

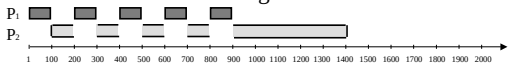
- Example: two processes execute concurrently

```
process P1                    process P2
begin                         begin
  for i := 1 to 5 do            <execute for 1 sec >
    <read a char>               end P2
    <process a char>
  end for
end P1
```

- Performance without scheduling



- Performance with scheduling



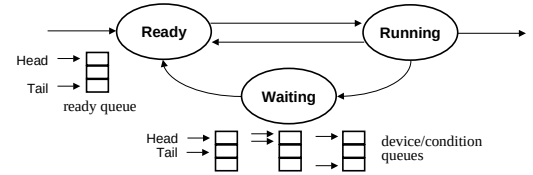
Types of Schedulers

- 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

9

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*

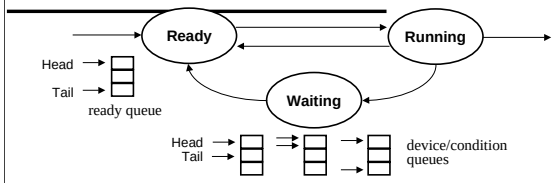
(3a. a process is *created*)

4. from *running* to *terminated*

10

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
```

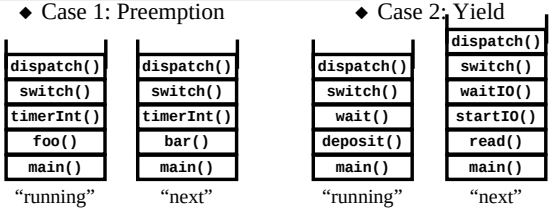
```
dispatch(proc : process_id)
begin
  <save memory image of runningProcess>
  <save processor state of runningProcess>
  <load memory image of proc>
  <load processor state of proc>
  runningProcess := proc
end dispatch
```

11

Implementing a Context Switch

Dispatching

- Case 1: Preemption
- Case 2: Yield



"running's" dispatch:

```
dispatch()
begin
  <save state of running>
    :
end dispatch
```

"next's" dispatch:

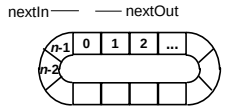
```
dispatch()
begin
  <save state of running>
  <load state of next>
end dispatch
```

12

Producer/Consumer Implementation

```
process Producer
  var c : char
begin
  loop
    <produce a character "c">
    while nextIn+1 mod n = nextOut do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
  end loop
end Producer
```

```
process Consumer
  var c : char
begin
  loop
    while nextIn = nextOut do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n
   <consume a character "c">
  end loop
end Consumer
```



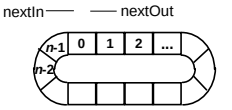

```
globals
   buf : array [0..n-1] of char;
   nextIn, nextOut : 0..n-1 := 0
```

13

Producer/Consumer Implementation with a shared counter

```
process Producer
  var c : char
begin
  loop
    <produce a character "c">
    while count = n do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
    count  := count + 1
  end loop
end Producer
```

```
process Consumer
  var c : char
begin
  loop
    while count = 0 do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n
    count   := count - 1
   <consume a character "c">
  end loop
end Consumer
```



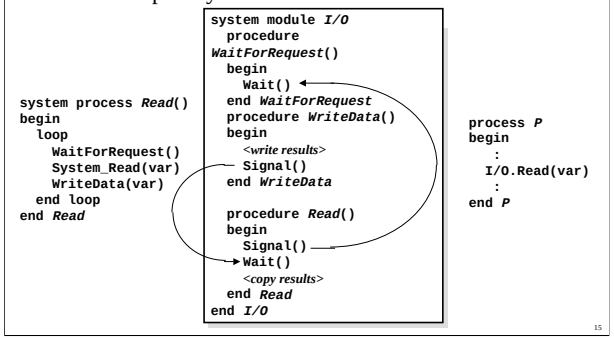

```
globals
   buf   : array [0..n-1] of char;
   nextIn,nextOut : 0..n-1 := 0
   count : integer := 0
```

14

Process Coordination

Producer/Consumer systems

- Example: Synchronous I/O



15