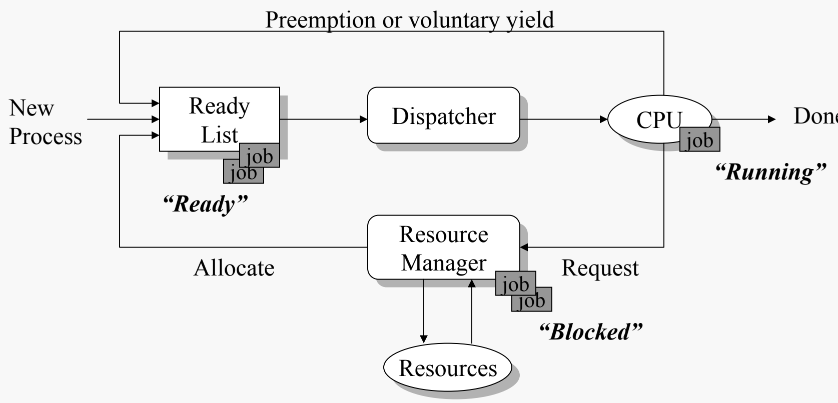
## Model of Process Execution

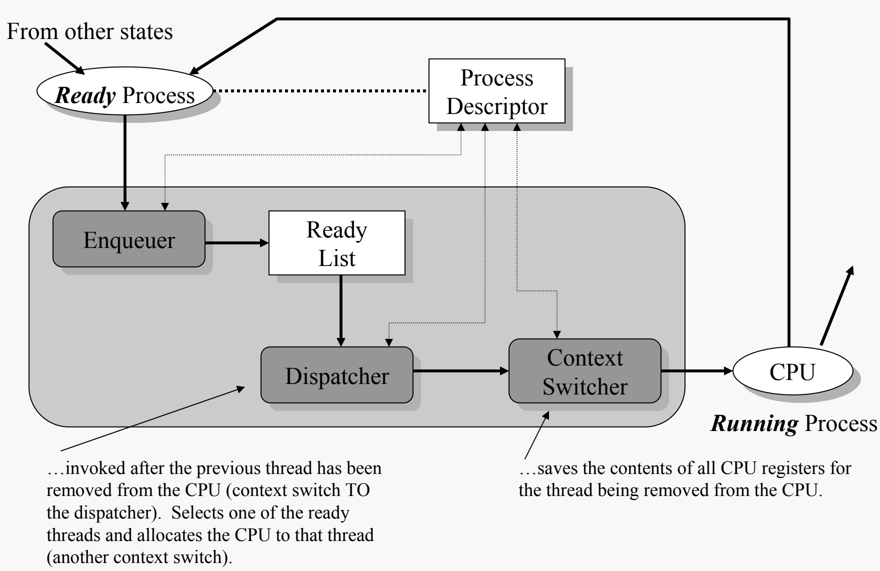







## The Scheduler

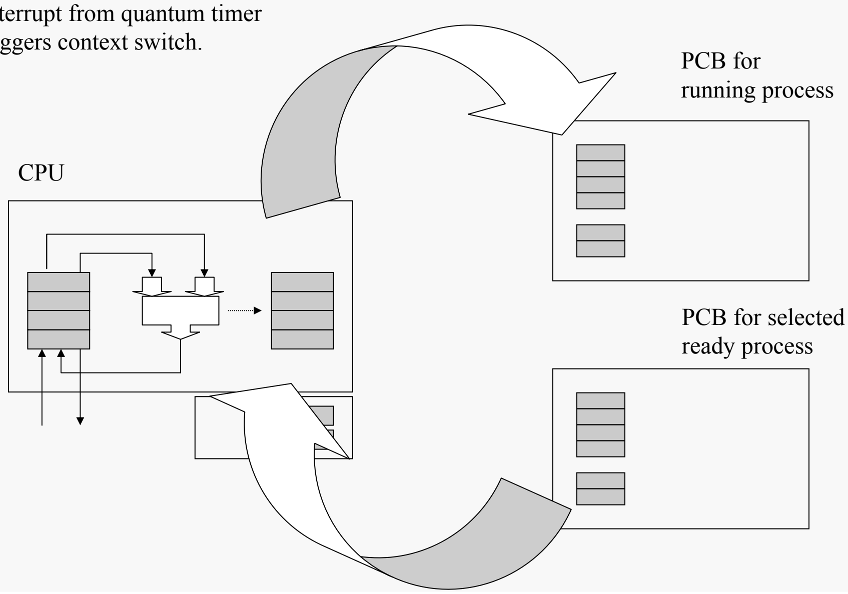




## Process Switching



Interrupt from quantum timer triggers context switch.

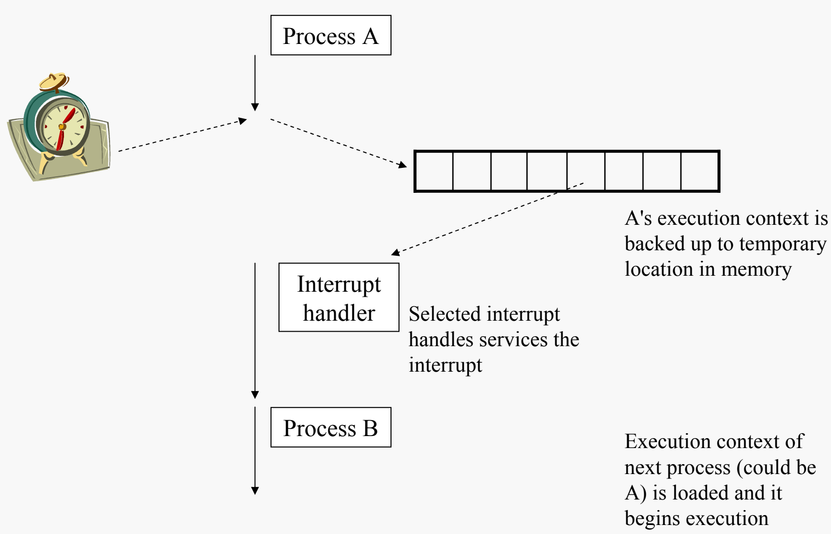






## Interrupt Processing



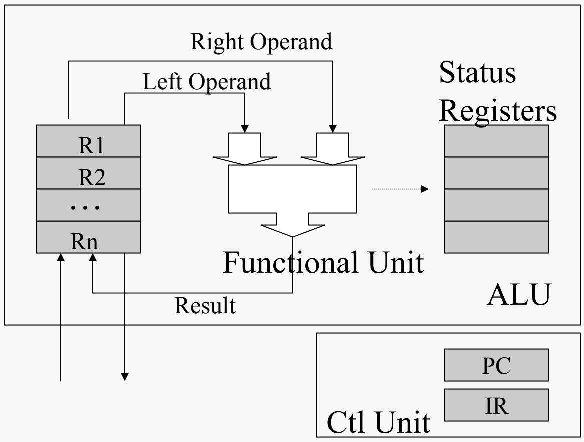


# Process Context

Context is saved in the PCB for the process.

Saving the context for "old" process might take about 2 microseconds.

Loading context for "next" process takes similar amount of time.




Execution of the dispatcher is not free.

So total time for performing a process switch might be 4+ microseconds.

1GHz processor might execute 2000 register instructions in time for a process switch… overhead!

Duplicate register sets for user and kernel mode exec can reduce cost by ½.





# Invoking the Scheduler

Need a <u>*mechanism*</u> to call the scheduler:

Voluntary call
- process blocks itself
- calls the scheduler

Every process periodically yields to the scheduler

Relies on correct process behavior
- malicious
- accidental

Prone to disruption by ill-behaved processes

Involuntary call
- external force (interrupt) blocks the process
- calls the scheduler

Interval timer
- device to produce a periodic interrupt
- programmable period

# Voluntary CPU Sharing

Currently running process P1 calls yield() to cede the processor to process P2:

```
// Machine instruction yield() saves contents of PC at r
// and loads the PC with contents at s

yield(r, s) {

    memory[r] = PC;
    PC = memory[s];
}
```

Address r will lie within the PCB for P1 (calling process) and can be determined implicitly at runtime.

Address s can be determined similarly if the identity of P2 is known.

Alternative model would place responsibility for choosing P2 on the scheduler.






# Involuntary CPU Sharing

Interval timer device handler

- keeps an in-memory clock up-to-date (see Chap 4 lab exercise)
- invokes the scheduler

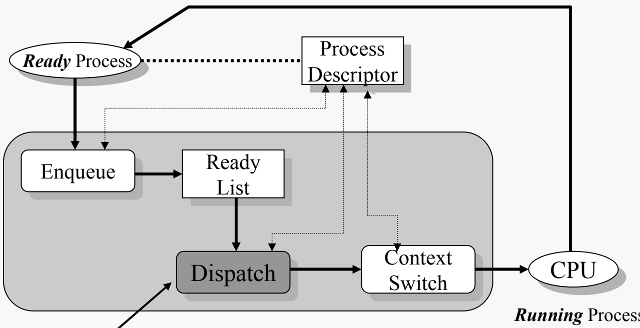
```
IntervalTimerHandler() {

    Time++;  // update the clock
    TimeToSchedule--;
    if(TimeToSchedule <= 0) {
        <invoke scheduler>;
        TimeToSchedule = TimeSlice;
    }
}
```

Involuntary CPU sharing – timer interrupts

- *time quantum* determined by interval timer – usually fixed size for every process using the system
- sometimes called the *time slice length*

## Choosing a Process to Run






Mechanism never changes

*Strategy* = policy the dispatcher uses to select a process from the ready list

Different policies for different requirements




## Policy Considerations



Policy can control/influence:

- CPU utilization
- average time a process waits for service
- average amount of time to complete a job

Could strive for any of:

- equitability (sounds good, vague)
- favor very short or long jobs (throughput vs response time)
- meet priority requirements (e.g., process control systems)
- meet deadlines (e.g., real-time systems)

## Optimal Scheduling



The *service time* $\tau(p)$ for a process is the amount of time the process requires in the running state (using the CPU) before it is completed.

Suppose the scheduler knows the $\tau(p_i)$ for each process $p_i$.

Policy can optimize on any criteria, e.g.,
- CPU utilization
- waiting time
- deadline

To find an *optimal schedule*:
- have a finite, fixed # of $p_i$
- know $\tau(p_i)$ for each $p_i$
- enumerate all schedules, then choose the best

Issues…?




## However ...



The $\tau(p_i)$ are almost certainly just estimates (at best).

General algorithm to choose optimal schedule is $O(n^2)$

Other processes may arrive while these processes are being serviced

Usually, optimal scheduling is only a theoretical benchmark – scheduling policies try to approximate an optimal schedule

# Talking About Scheduling ...



Let $P = \{p_i \mid 0 \le i < n\}$ = set of processes in system

Let $S(p_i) \in$ {running, ready, blocked} (the *process state*)

Let $\tau(p_i)$ = time process needs to be in running state (the *service time*)

Let $W(p_i)$ = Time $p_i$ is in ready state before first transition to running (*wait time*)

Let $T_{TRnd}(p_i)$ = Time from $p_i$ first enter ready to last exit run (*turnaround time*)

Let $W_{TRnd}(p_i) = T_{TRnd}(p_i) / \tau(p_i)$ (*weighted turnaround time*)

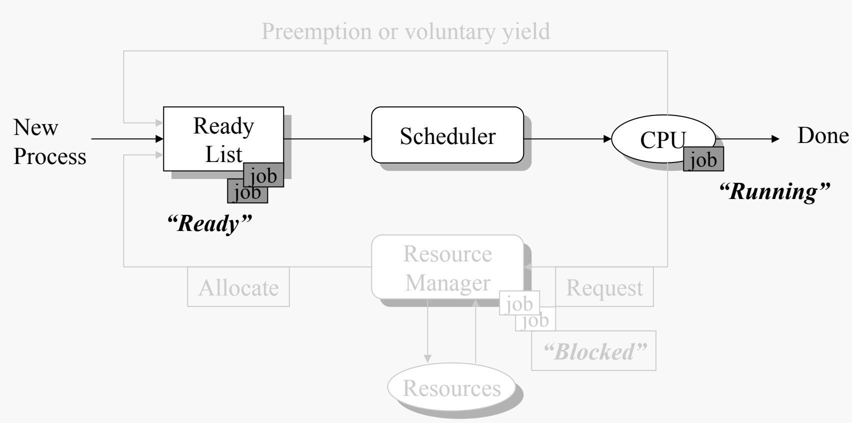
Batch *Throughput rate* = inverse of avg $T_{TRnd}$

*Timesharing response time* = $W(p_i)$ is of most interest to interactive users



# Simplified Model





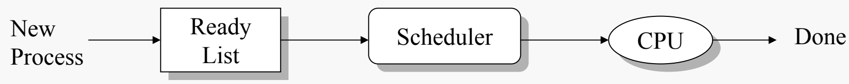

Simplified, but still provides analysis results

Easy to analyze performance

No issue of voluntary/involuntary sharing

# Estimating CPU Utilization



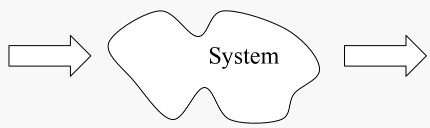



Let $\lambda$ = the average rate at which processes are placed in the Ready List, *arrival rate*

Let $\mu$ = the average *service rate*

$\therefore$ $1/\mu$ = the average $\tau(p_i)$

$\lambda$ $p_i$ per second

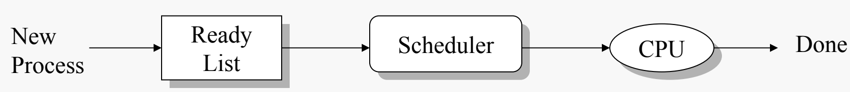



Each $p_i$ uses $1/\mu$ units of the CPU, on average




# Estimating CPU Utilization






Let $\lambda$ = the average rate at which processes are placed in the Ready List, *arrival rate*

Let $\mu$ = the average *service rate*

$\therefore$ $1/\mu$ = the average $\tau(p_i)$

Let $\rho$ = the fraction of the time that the CPU is expected to be busy. Then:

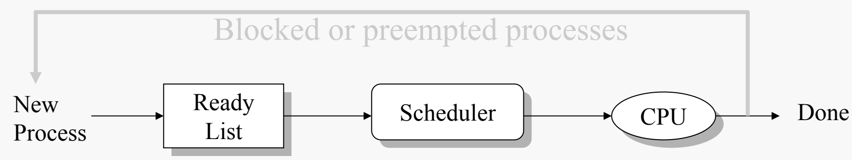
$\rho$ = # $p_i$ that arrive per unit time × avg time each spends on CPU

$$\rho = \lambda * 1/\mu = \lambda/\mu$$

Note: must have $\lambda < \mu$ (i.e., $\rho < 1$)
What if $\rho$ approaches 1?

# Nonpreemptive Schedulers






We can try to use the simplified scheduling model.

Only consider running and ready states

Ignores time in blocked state:

- "New process created when it enters ready state"
- "Process is destroyed when it enters blocked state"
- Really just looking at "small phases" of a process

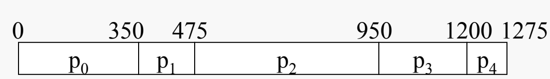


# FCFS Average Wait Time



Easy to implement
Ignores service time, etc
Not a great performer

| $i$ | $\tau(p_i)$ |
|---|---|
| 0 | 350 |
| 1 | 125 |
| 2 | 475 |
| 3 | 250 |
| 4 | 75 |




$T_{TRnd}(p_0) = \tau(p_0) = 350$ $\quad W(p_0) = 0$

$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$ $\quad W(p_1) = T_{TRnd}(p_0) = 350$

$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$ $\quad W(p_2) = T_{TRnd}(p_1) = 475$

$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$ $\quad W(p_3) = T_{TRnd}(p_2) = 950$

$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$ $\quad W(p_4) = T_{TRnd}(p_3) = 1200$

$$W_{avg} = (0+350+475+950+1200)/5 = 2974/5 = 595$$

# Predicting Wait Time in FCFS



In FCFS, when a process arrives, all in ready list will be processed before this job

Let $\mu$ be the service rate

Let L be the ready list length

$W_{avg}(p) = L*1/\mu + 0.5* 1/\mu = L/\mu+1/(2\mu)$

Compare predicted wait with actual in earlier examples

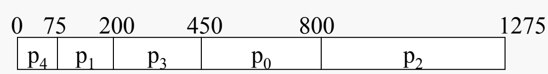



# Shortest Job Next



| i | $\tau(p_i)$ |
|---|---|
| 0 | 350 |
| 1 | 125 |
| 2 | 475 |
| 3 | 250 |
| 4 | 75 |

Minimizes wait time
May starve large jobs
Must know service times




$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 475+350+250+125+75 = 1275$

$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$

$T_{TRnd}(p_4) = \tau(p_4) = 75$

$W(p_0) = 450$
$W(p_1) = 75$
$W(p_2) = 800$
$W(p_3) = 200$
$W(p_4) = 0$

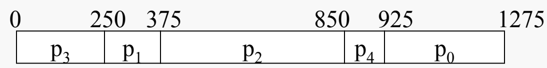
$W_{avg} = (450+75+800+200+0)/5 = 1525/5 = 305$

## Priority Scheduling



| i | $\tau(p_i)$ | Pri |
|---|---|---|
| 0 | 350 | 5 |
| 1 | 125 | 2 |
| 2 | 475 | 3 |
| 3 | 250 | 1 |
| 4 | 75 | 4 |

Reflects importance of external use
May cause starvation
Can address starvation with aging




$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_4)+\tau(p_2)+\tau(p_1)\ )+\tau(p_3) = 350+75+475+125+250$
$= 1275$
$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_3) = 125+250 = 375$
$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_1)+\tau(p_3) = 475+125+250 = 850$
$T_{TRnd}(p_3) = \tau(p_3) = 250$
$T_{TRnd}(p_4) = \tau(p_4)+ \tau(p_2)+ \tau(p_1)+\tau(p_3) = 75+475+125+250 = 925$

$W(p_0) = 925$
$W(p_1) = 250$
$W(p_2) = 375$
$W(p_3) = 0$
$W(p_4) = 850$

$W_{avg} = (925+250+375+0+850)/5 = 2400/5 = 480$

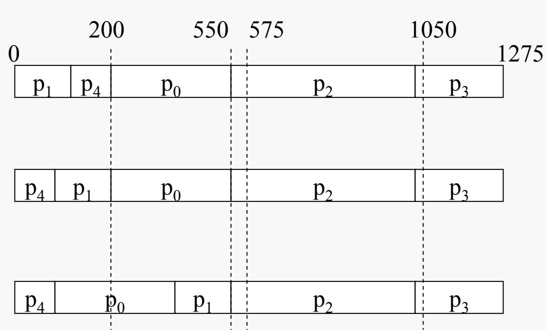


## Deadline Scheduling

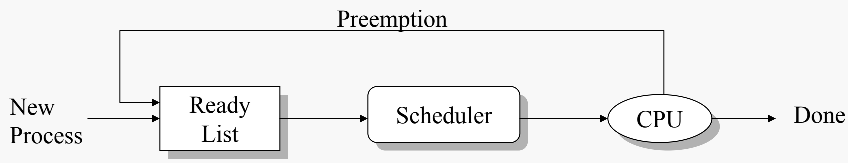


| i | $\tau(p_i)$ | Deadline |
|---|---|---|
| 0 | 350 | 575 |
| 1 | 125 | 550 |
| 2 | 475 | 1050 |
| 3 | 250 | (none) |
| 4 | 75 | 200 |

Allocates service by deadline
May not be feasible



# Preemptive Schedulers




Highest priority process is guaranteed to be running at all times, or at least at the beginning of a time slice

Dominant form of contemporary scheduling

But complex to build and analyze

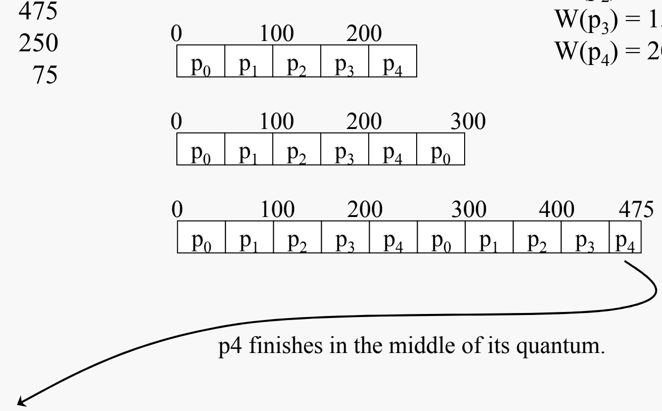




# Round Robin (TQ = 50)

| i | $\tau(p_i)$ |
|---|---|
| 0 | 350 |
| 1 | 125 |
| 2 | 475 |
| 3 | 250 |
| 4 | 75 |

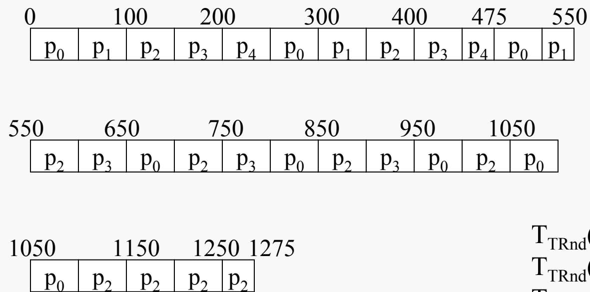
$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$




$T_{TRnd}(p_4) = 475$

# Round Robin (TQ = 50)

| i | $\tau(p_i)$ |
|---|---|
| 0 | 350 |
| 1 | 125 |
| 2 | 475 |
| 3 | 250 |
| 4 | 75 |




$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$T_{TRnd\_avg} = (1100+550+1275+950+475)/5 = 4350/5 = 870$

$W_{avg} = (0+50+100+150+200)/5 = 500/5 = 100$

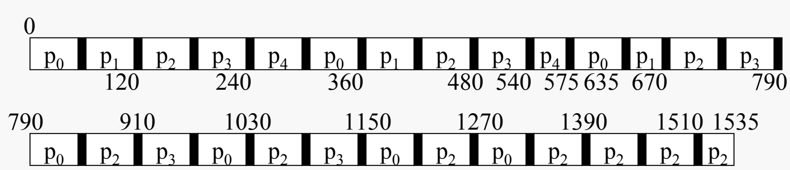





# Round Robin with Overhead = 10 (TQ = 50)

Overhead must be considered

| i | $\tau(p_i)$ |
|---|---|
| 0 | 350 |
| 1 | 125 |
| 2 | 475 |
| 3 | 250 |
| 4 | 75 |



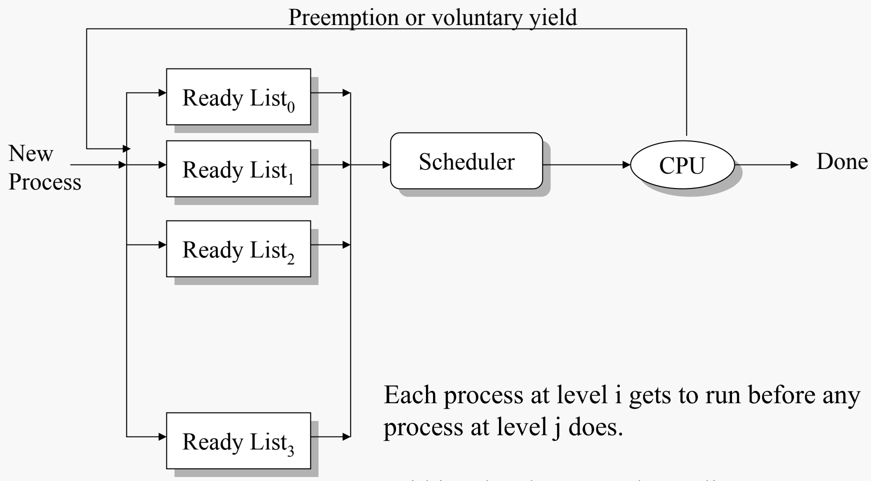

$T_{TRnd}(p_0) = 1320$
$T_{TRnd}(p_1) = 660$
$T_{TRnd}(p_2) = 1535$
$T_{TRnd}(p_3) = 1140$
$T_{TRnd}(p_4) = 565$

$W(p_0) = 0$
$W(p_1) = 60$
$W(p_2) = 120$
$W(p_3) = 180$
$W(p_4) = 240$

$T_{TRnd\_avg} = (1320+660+1535+1140+565)/5 = 5220/5 = 1044$

$W_{avg} = (0+60+120+180+240)/5 = 600/5 = 120$

## Multi-Level Queues




Each process at level i gets to run before any process at level j does.

Within a level, use another policy, e.g. RR





## Multilevel Feedback Queues

Different processes have different needs

- short I/O-bound interactive processes should generally run before processor-bound batch processes
- behavior patterns are not immediately obvious to the scheduler, but can be deduced from process behavior

Multilevel feedback queues

- arriving processes enter the highest-level queue (or based on initial priority) and execute with higher priority than processes in lower queues
- long processes repeatedly descend into lower levels
  - gives short processes and I/O-bound processes higher priority
  - long processes will run when short and I/O-bound processes terminate
- processes in each queue are serviced using round-robin
  - process entering a higher-level queue preempt running processes

Algorithm must respond to changes in environment

- move processes to different queues as they alternate between interactive and batch behavior
- adaptive mechanisms incur overhead that often is offset by increased sensitivity to process behavior

## Contemporary Scheduling

Involuntary CPU sharing -- timer interrupts

- *time quantum* determined by interval timer -- usually fixed for every process using the system
- sometimes called the *time slice length*

*Priority*-based process (job) selection

- select the highest priority process
- priority reflects policy

With *preemption*

Usually a variant of *multi-level queues*






## Contemporary Scheduling

BSD 4.4 Scheduling

- Involuntary CPU Sharing
- Preemptive algorithms
  - 32 Multi-Level Queues
  - queues 0-7 are reserved for system functions
  - queues 8-31 are for user space functions
  - `nice` influences (but does not dictate) queue level

Windows NT/2K Scheduling

- Involuntary CPU sharing across threads
- Preemptive algorithms
- 32 multi-level queues
  - highest 16 levels are “real-time”
  - next lower 15 are for system/user threads
    - range determined by process base priority
  - lowest level is for the idle thread

## Scheduling Criteria



Processor-bound processes
- use all available processor time

I/O-bound processes
- generates an I/O request quickly and relinquishes processor

Batch processes
- contains work to be performed with no user interaction

Interactive processes
- requires frequent user input, rapid response times are important



## Real-Time Scheduling



Static real-time scheduling
- does not adjust priorities over time
- low overhead
- suitable for systems where conditions rarely change
  - hard real-time schedulers
- rate-monotonic (RM) scheduling
- process priority increases monotonically with the frequency with which it must execute
- deadline RM scheduling
- useful for a process that has a deadline that is not equal to its period

Dynamic real-time scheduling
- adjusts priorities in response to changing conditions
- can incur significant overhead, but must ensure that the overhead does not result in increased missed deadlines
- priorities are usually based on processes' deadlines
- earliest-deadline-first (EDF)
  - preemptive, always dispatch the process with the earliest deadline
- minimum-laxity-first
  - similar to EDF, but bases priority on laxity, which is based on the process's deadline and its remaining run-time-to-completion

# Scheduling Levels

Short-term scheduling

- the decision as to which available process will be assigned the processor next
- known as the dispatcher
- executes most frequently
- invoked when an event occurs (clock interrupts, I/O interrupts, operating system calls, signals)

Medium-term scheduling

- the decision to add to the number of processes that are partially or fully contending for the processor
- part of the swapping function
- based on the need to manage the degree of multiprogramming

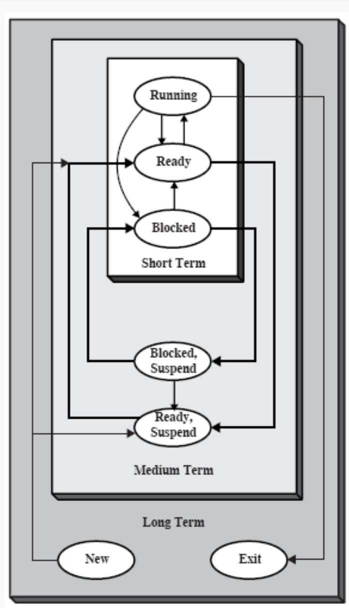
Long-term scheduling

- the decision to add to the pool of processes which will eventually be executed
- determines which programs are admitted to the system for processing
- controls the degree of multiprogramming
- more processes, smaller percentage of time each process is executed





# Scheduling Levels in the State Diagram



## Scheduling Criteria



### User-oriented, performance related criteria

Turnaround time
- interval of time between the submission of a process and its completion
- appropriate measure for a batch job

Response time
- time from the submission of an interactive request until the response begins to be received
- better measure than turnaround for an interactive process
- goal is low response time and maximization of the number of interactive users receiving acceptable response time

Deadlines
- only applicable when completion deadlines can be specified
- subordinate other goals to that of maximizing the percentage of deadlines met




## Scheduling Criteria



### User-oriented, not performance related

Predictability
- a given job should run in about the same amount of time regardless of the system load
- wide variation in response time or turnaround time is distracting to interactive users

### System-oriented, performance related

Throughput
- the number of processes completed per unit time
- measure of how much work is being performed
- clearly depends upon the average service time, but also on scheduling policies

Processor utilization
- percentage of time that the processor is busy
- must be considered in relation to the number of processes that are ready but not running
- less important on real-time systems

# Scheduling Criteria

## System-oriented, not performance related

Fairness

- processes should be treated the same, and no process should suffer starvation, in the absence of contradictory guidance from the user or other system components

Enforcing priorities

- when priorities are used, the scheduling policy should favor higher-priority processes

Balancing resources

- system resources should be kept busy, if there is sufficient demand to do so
- processes that will underutilize stresses resources should be favored
- relates also to medium- and long-term scheduling