SCHEDULING BASICS

SIMPLE ROUND-ROBIN-MULTIPROGRAMMING WAITING TIME vs UTILIZATION

PERFORMANCE ANALYSIS LITTLE’S FORMULA

SHORTEST RUN TIME FIRST (NEXT)- BATCH, CPU BURSTS TURNAROUND TIME
SHORTEST RUN TIME FIRST (NEXT)- BATCH, CPU BURSTS TURNAROUND TIME
SHORTEST RUN TIME FIRST-MULTIPROGRAMMING
  SIMPLE ROUND ROBIN TURNAROUND TIME

EFFECT OF FAVORING IO BOUND PROCESSES 1
EFFECT OF FAVORING IO BOUND PROCESSES 2

BASIC DYNAMIC PRIORITY USEFUL FOR ENFORCING FAIRNESS AND EFFICIENCY
MORE SOPHISTICATED USE OF PRIORITY IN PREDICTING CPU BURSTS
  - LEVELS AND USAGE (AGING) CALCULATION

UNIX-LIKE PRIORITY QUEUES

SCHEDULING SCHEMES FOR ASSIGNING TIME TO PROCESSES AIMED AT FAIRNESS
DEFINING PRIORITY LOTTERY SCHEDULING

REAL TIME SCHEDULING-ALGORITHMS
REAL TIME SCHEDULING-PERIODIC SCHEDULABILITY
**Units**

**Job:** A Job is associated with a program. It consists of the total of all CPU and I/O action necessary for completion of a given run of a program. The raw time for a Job is the time it would take if it alone had access to CPU and I/O.

**CPU Bursts:** The CPU receives a series of uses each called a burst. These come from different processes, but their time distribution, independent of where they come from, can be used in evaluation of different scheduling algorithms.

**Measures Of Performance**

Usually interest is in the average value of these Performance Measures.

**Turnaround:** The total time interval \([t_{\text{finish}(P_i)} - t_{\text{start}(P_i)}]\) between availability for running and completion of a job, including I/O, CPU and waiting time. Particularly Important In batch environment. Avg and Max.

It can be applied to CPU burst in which case it is the time interval from the instant the process generating the burst enters the ready state (queue) till it completes its burst in the CPU. (User Satisfaction)

\[
\text{Turnaround}_{\text{ave}} = \frac{\sum_{i=1}^{M} [t_{\text{finish}(P_i)} - t_{\text{start}(P_i)}]}{M} \quad M = \text{Number of Jobs}
\]

**Throughput:** The Number of Jobs completed per unit time

(Overall Efficiency)

\[
\text{Throughput} = \frac{\text{Number of Jobs}}{[t_{\text{finish(all Jobs)} - t_{\text{start(1st Job)}}}]}
\]

\[
(1/\text{Throughput} = \text{The average Time (Spent Computing) per Process.})
\]

**Response Time:** The CPU Interval Between request for attention and delivery of attention. Important in time sharing environments. (User Satisfaction)

**Waiting Time:** The time between entering the ready state, (entering the ready queue), and entering the active state. Applied to CPU bursts. (User Satisfaction)

**CPU Utilization:** The percentage or fraction of total time the CPU is in use excluding OS. If all processes other than OS are doing I/O the CPU is not in use. - This is effected by I/O usage, context switches and MM size. (Overall Efficiency)

---

**SCHEDULING BASICS**

Scheduling For Batch MultiProgramming Real time

© M C Paull 2005
Holding Number Of Processes and Context Switch time Constant, 
As Quanta Time Grows: 
1 **Waiting Time** increases. 
2 Fraction of time that processes, as opposed 
to OS, are in CPU (**Utilization**) increases.

**Waiting Time**

For 8 processes 
\[ t_{return} = 8t_o + 7t_p \]
For n processes: 
\[ t_{return} = nt_o + (n-1)t_p \]

\( t_{return} \) is the response time, the interval between a 
request for attention and getting that attention. To 
make this reasonably small it is desirable to 
keep \( t_p \), the quantum, **LOW**.

**Utilization**

\[ f_{useful} = \frac{t_p}{(t_p + t_o)} = \frac{1}{1 + t_o/t_p} \]

\( f_{useful} \) is the fraction of the time that processes are 
active. To keep this reasonably high it is desirable to 
keep \( t_p \), the quantum **HIGH**. 
Also Turnaround improves with larger quanta.
**Little’s Steady State Formula:** \[ \lambda = \frac{n}{W} \]

**Ave arrival rate** (\(\lambda\)) = 
**Ave Number in queue** (\(n\)) / **Ave Time a Process Spends in Queue** (\(W\))

\[ = \text{Ave leaving rate} (\lambda) \]

[any 2 of the variables determines the 3rd].

**Explanation**

If there are an average of \(n\) in queue and the average time each spends (Waits) in the queue (buffer) is \(W\), then, on average, **all** \(n\) leave every \(W\) time units. So the average rate at which they leave is \(\lambda = \frac{n}{W}\).

---

**Example**

Death rate (leaving queue) = Number alive today (in queue) / Average time alive (in queue) = Birth rate

Rate people arrive at diner = Number in Diner (in queue) / Average time in Diner (in queue)

100 wait queue occupants (in Diner), average time in wait queue (stay in Diner) = 2 hours leaving rate (= arrival rate) = \(\frac{100}{2} = 50\) per hour. (If they leave uniformly? 25 will have left in 1/2 hour which means if there are 25 ahead of you you can expect to wait 1/2 hour.

---

**USES**

**PERFORMANCE ANALYSIS LITTLE’S FORMULA**

© M C Paull 2005
Turnaround Time For Job J is the time from its submission to its completion. For a CPU burst it is the time from entry in the Ready Q to entry into state Active.

In the following chart the time required to run the ith job, J_i, started in batch mode is t_i.

<table>
<thead>
<tr>
<th>Order Of Entry</th>
<th>Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>t_0</td>
</tr>
<tr>
<td>2nd</td>
<td>t_0 + t_1</td>
</tr>
<tr>
<td>3rd</td>
<td>t_0 + t_1 + t_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n-1st</td>
<td>t_0 + t_1 + t_2 + ... + t_{n-2}</td>
</tr>
<tr>
<td>nth</td>
<td>t_0 + t_1 + t_2 + ... + t_{n-2} + t_{n-1}</td>
</tr>
<tr>
<td>Total</td>
<td>n t_0 + (n-1) t_1 + ... + 2 t_{n-2} + t_{n-1}</td>
</tr>
</tbody>
</table>

Average Turnaround Time = Total (Turnaround Time) / t_n
This is clearly minimized if 0 ≤ t_1 ≤ ... ≤ t_{n-2} ≤ t_{n-1} ≤ t_n.

Another way to show this is:
To determine the order of submission of processes which will yield the minimum average turnaround time we will consider which should be the 1st, the 2nd, ... the ith to be submitted. If the job which takes the smallest time is the one submitted 1st that will result in the smallest possible turnaround time for the 1st job submitted. Then if the 2nd job submitted is that with the 2nd smallest time is submitted we will have the smallest possible turnaround times for the first two jobs submitted. This argument can be continued to show that if the ith smallest job is done ith then the smallest possible turnaround for all jobs from the first to the ith will be obtained.

Still another way to show this is: Suppose J_y requires more time than J_x, but J_y is done before J_x. Then the time to do J_y is included in the turnaround time of all jobs done after J_y. The time for both J_x and J_y are included in the turnaround times of jobs done after J_x. It is easy to see that by interchanging the time at which these two jobs are done with J_x before J_y now, the turnaround time of all jobs between the two will be decreased, while the times after J_y will be the same. So interchange in this case will decrease the total turnaround time.

**Examples**

**Shortest First Optimal**

<table>
<thead>
<tr>
<th>Arrivals</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_5</td>
<td>2+6+7+8+9/5=32/5=6+2/5</td>
</tr>
<tr>
<td>J_4</td>
<td></td>
</tr>
<tr>
<td>J_3</td>
<td></td>
</tr>
<tr>
<td>J_2</td>
<td></td>
</tr>
<tr>
<td>J_1</td>
<td></td>
</tr>
</tbody>
</table>

All arrive at time 0 Non-Optimal Ordering

<table>
<thead>
<tr>
<th>Arrival Rate</th>
<th>Ave # in Q</th>
<th>Ave Wait in Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Rate</td>
<td>Ave # in Q</td>
<td>Ave Wait in Q</td>
</tr>
</tbody>
</table>

**Shortest First Without Preemption**

<table>
<thead>
<tr>
<th>Arrivals2</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_5</td>
<td>2+6+4+5+6=23/5</td>
</tr>
<tr>
<td>J_4</td>
<td></td>
</tr>
<tr>
<td>J_3</td>
<td></td>
</tr>
<tr>
<td>J_2</td>
<td></td>
</tr>
</tbody>
</table>

More than 1 arrival time

**Shortest First With Preemption**

<table>
<thead>
<tr>
<th>Arrivals1</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_1</td>
<td>2 +1+2+3+9=17**5</td>
</tr>
<tr>
<td>J_2</td>
<td></td>
</tr>
<tr>
<td>J_3</td>
<td></td>
</tr>
</tbody>
</table>

More than 1 arrival time

© M C Paull 2005
Job completed

The difference in average turnaround between the best and worst ordering is low compared to that difference when the same number of jobs is run in batch mode.

\[ \sum_{j=1}^{n} d_j = (n + n-1 + \ldots + 1) = n(n-1)/2 = (n^2-n)/2 \]  
**Worst Case**

\[ \sum_{j=1}^{n} d_j = (1 + 1 + \ldots + 1) = n \]  
**Best Case**

\[ (n^2-n)/2 - n = (n^2-3n)/2 = O(n^2) \]  
**Difference**

The total of all turnaround times is the sum of the quanta \( Q_i \) executed before job \( J_i \) is completed.

\[ Q_{i+1} = [n]N_1 + [n-1]D_{2,1} + [n-2]D_{3,2} + \ldots + [n-i](D_{i+1,i} - 1) + d_{i+1} \]

So for example:

\[ Q_2 = [n](N_1) + [n-1] [D_{2,1} - 1] + d_2 \]

\[ Q_3 = [n](N_1) + [n-1] D_{2,1} + [n-2][D_{3,2} - 1] + d_3 \]

The total of all turnaround times is the sum of the quanta \( Q_i \) executed before job \( J_i \) is completed. The only variation in the result due to reordering of the jobs is the value of \( d_i \). \( d_i = 1 \), for all \( i \), is the smallest it can be. It takes this value if the jobs are ordered so that the quanta for the \( i \)th smallest job is done first.

**Example**

<table>
<thead>
<tr>
<th>Job</th>
<th>Completion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>4</td>
</tr>
<tr>
<td>J2</td>
<td>4</td>
</tr>
<tr>
<td>J3</td>
<td>5</td>
</tr>
<tr>
<td>J4</td>
<td>6</td>
</tr>
</tbody>
</table>

\[ t_1 = 4 \]

\[ t_1 + t_2 = 4 + 3 \]

\[ t_1 + t_2 + t_3 = 4 + 3 + 4 \]

\[ \text{Sum} = 22 \]

\[ t_1 = \text{time to complete J1 when running round robin} \]

\[ t_1 + t_2 = \text{time to complete J2 when running round robin} \]

\[ t_1 + t_2 + t_3 = \text{time to complete J3 when running round robin} \]
Assuming a process which blocks on IO system call indicates it is IO bound. Increasing its priority will produce more parallelism - CPU and IO, IO and IO.

**EFFECT OF FAVORING IO BOUND PROCESSES**

© M C Paull 2005
Suppose a Process, P1, consistently blocks early in its Quanta on a Semaphor (down). This implies it has a fast response to whatever its partner Process, P2, did when it ran or perhaps P2 did not run at all since P1 last left. Decreasing P1’s and/or Increasing P2’s Priority should allow P1 to run for a longer time when it is made active since the partner P2 will have more time to produce work for P1.
Priority is designed to give smaller CPU bursts (I/O bound jobs) high priority. If smaller CPU bursts are done earlier then the average queue waiting time is minimized, as is the average (CPU burst) turnaround time. Future use of IO is assumed proportional to last use of I/O. The nature of jobs is that some are I/O bound and others CPU bound.

Usage can be computed so as to give more weight to current usage.

The priority calculation based on usage can also take account of how much usage one is entitled to. This can be based on equal entitlement for each user dividing the entitlement equally among all the processes of that user which are running.

Also instead of running the highest priority process one may let the priority determine the probability that a process will run.

**BASIC DYNAMIC PRIORITY** USEFUL FOR ENFORCING FAIRNESS AND EFFICIENCY

© M C Paull 2005
Priority Levels: Queues
(Assumes groups of Processes have essentially same Priority)

UNIX Numbering
(So Usage Can Be Added To Priority and Result In Lower Priority)

Level Increases After I/O expecting I/O bound
Round Robin At Each level
Level Decreases as Use Increases

usage(n) = \( f(usage(n-1)) + T \) recursive definition

usage(n) = \( \frac{(1-f)n+1}{(1-f)}\) as \( n \to \infty \) closed form

Proof

\[
(1-f)^{n+1} / (1-f) = f * [(1-f)nT / (1-f)] + T \\
(1-f)^{n+1}T = f * (1-f)nT + T(1-f) = fT - f^{n+1}T + T - Tf = f^{n+1}T + T = (1-f)^{n+1}T
\]

usage(n) = \( f(usage(n-1)) + T \) (a constant value of ticks) recursive definition

usage(0) = estimated usage = T
usage(n) = \( \frac{1}{2}(usage(n-1)) + \frac{1}{2}(ticks(n-1-to-n)) \) per process
usage(1) = \( \frac{1}{2}(usage(0)) + \frac{1}{2}(ticks(0-to-1)) = \frac{1}{2}T + \frac{1}{2}(ticks(0-to-1)) \)
usage(2) = \( \frac{1}{2}(usage(1)) + \frac{1}{2}(ticks(1-to-2)) = \frac{1}{4}T + \frac{1}{4}(ticks(0-to-1)) + \frac{1}{2}(ticks(1-to-2)) \)
usage(3) = \( \frac{1}{2}(usage(2)) + \frac{1}{2}(ticks(2-to-3)) = \frac{1}{8}T + \frac{1}{8}(ticks(0-to-1)) + \frac{1}{4}(ticks(1-to-2)) + \frac{1}{2}(ticks(1-to-2)) \)

Usage Calculations
['Discounts Past']

More Sophisticated Use of Priority in Predicting CPU Bursts
- Levels and Usage (Aging) Calculation

© M C Paull 2005
1. Reset priorities-Priority 1 level has most usage decrease priority (increase number).
2. Priority 0 level runs for some time, but one goes to IO so others in priority level 0 get lots of usage.
3. When I/O completes returning process gets high priority -2.
4. At next Priority Reset priorities many changes in priority levels of processes.

UNIX-LIKE PRIORITY QUEUES

priority = 0 + 4/2 (usage per proc) +0 = 2

Processes blocked waiting for an event usually get negative priority when the event occurs:
- Terminal output gets -2
- Input gets -3
- Disk buffer gets -4
- I/O gets -5

priority = -2 + 5 + 0 = 3

priority of P1 > priority of P2 iff priority number of P1 < priority number of P2.

More Usage->Higher Priority#->lower Priority

*Assume base = current priority# if its negative, 0 otherwise

^nice can be set negative by system admin, or positive by user-assumed 0
Scheduling By Owner: (Each owner may have many processes)

Example:
Two Owners each given 1/2 of CPU time: So if Total CPU = 32 Quanta.
Each Owner is given 16 quanta. If owner₁ has 4 processes and owner₂
has 8 processes, each of owner₁’s processes gets 4 quanta and each of
owner₂’s processes gets 2 quanta.

In general: $T = \text{Total CPU time in quanta available}$
$N = \text{Number of Owners active.}$
$n_i = \text{Number of Processes belonging to owner}_i$

$\frac{T}{N} = \text{Total time in quanta for each owner}$
$(\frac{T}{N})n_i = \text{Time in quanta for each process belonging to owner}_i$

**FAIR SHARE SCHEDULING**
Each User Is given equal time Decide on how much to give each Process

**GUARANTEED SCHEDULING**
Each User Is Entitled to Equal Fraction of time present
-Method For assigning priorities to enforce entitlement.
All processes are given a number of tickets. When the OS is to choose a member of the Ready Queue to run. Assume \( P_j \) has \( x_j \) tickets. These are tickets are now numbered for each \( P_j \). Processes are ordered so that \( P_1 \) has \( x_1 \) tickets numbered 1 to \( x_1 \) \( P_2 \) has \( x_2 \) tickets numbered \( 1 + x_1 \) to \( x_1 + x_2 \) etc. Then OS chooses a number in the range of the assigned numbers \textbf{randomly}. The process assigned that number is given control of the CPU next.

\[
\text{Priority} = \text{Probability Of } P_j \text{ Getting Control} = \frac{x_j}{\sum_{i=n}^{i=n} x_i}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>( P_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2/10</td>
<td>1/10</td>
<td>3/10</td>
<td>1/10</td>
<td>2/10</td>
<td>1/10</td>
<td>Probability Of ( P_j )</td>
</tr>
</tbody>
</table>

**Lottery Scheduling**
A Probabilistic Way Of Specifying Priority
Gives Precise Meaning To Priority

To multiply the probability of being chosen by \( k \) it is necessary to know how many additional tickets are necessary. So suppose process \( P \) has \( x \) tickets and there are a total of \( S \) tickets. So \( P \)'s probability of being chosen when ready = \( x/S \). Now we wish to change the probability to \( kx/S \). We have to decide the number of tickets, \( t \), to be added or subtracted:

\[
\frac{kx}{S} = \frac{(x+t)}{(S+t)}
\]

\[
Sx + St = kxS + kxt
\]

\[
t(S-kx) = xS(k-1)
\]

\[
t = x(k-1) \left(1 - \frac{kx}{S}\right) \text{ Note } k-1 \text{ is positive if } k>1, \text{ otherwise negative}
\]

But now there are a different number of tickets, so one whose previous probability was \( y/S \) now has probability \( y/(S+t) = qy/S \) (\( q \) times its previous probability) and \( q = S/S+t \). This is inevitable since the total of all probabilities must be 1.

**Note On Changing Probabilities**
Increasing and Decreasing Priority-Give and Take a Tickets

---

**DEFINING PRIORITY LOTTERY SCHEDULING**

© M C Paull 2005
(Can Usually Be Handled by a General System)

HARD REAL-TIME: Must Guarantee: Each Process Gets The Needed Time (Often Requires Special Hardware- In a general System Page Replacements, etc Are Not Sufficiently Predictable

Hard Real Time Scheduling: The scheduling of multiple competing processes some or all of which have **deadlines**.

Examples:  
A real-time multi-media scheduler each running periodically-maybe sending out successive frames of a movie-sent out at different rates because of the terminal facilities for handling different rates (different band-widths), and the fact that sound and video require different rates, and because video compression-decompression is often involved.

So what we have n periodically active processes. The key aspects are the periods or times from the start of one burst of activity to the next of each process. These are $P_1, P_2, ..., P_n$, and the length of the bursts; $L_1, L_2, ..., L_n$. From these we calculate a deadline for each process, that is the maximum time after the start of a processes burst by which it must again start another burst.

With pre-emption for the $i$th process this is $t_{current} + P_i - \text{remainder of } (L_i)$.

**Schedulable**
Here is a necessary, not itself sufficient condition for periodic processes to all be able to run on a single processor, $M$ in an acceptable order, but sufficient to all fit in the allotted time.

If there are $n$ real time processes to be run on processor $M$, and the $j$th must be active a fraction $F_j$ of each of its period (interval) $P_j$, then they cannot all be handled unless

$$\sum_{j=1}^{n} F_j \leq 1$$

If the $j$th real time process is periodic with a period $P_j$ during which time it must uses processor $M$ for time $C_j$, then $F_j = C_j / P_j$.

So

$$\sum_{j=1}^{n} \left[ C_j / P_j \right] \leq 1$$

This is a necessary, but not sufficient condition. Sufficiency depends on the extent to which the $C_j$ s can be broken up.
So it could all be done if, for example, the distributions above were OK

Hard Real Time Scheduling With Preemption

**Rate Monotonic Scheduling (RMS)** (For Periodic Processes): **Priorities** = 1/Period = Frequency
Always Run Highest **Priority** Available

**Earliest Deadline First (EDF)**: Works for Periodic and Non Periodic Deadline is the time at which an operation must be completed to avoid disaster. In the case of Periodic computations it would be the start of next cycle. Always run the one available with the closest deadline.
Unbounded Priority Inversion

Priority Inheritance Protocol
When Hi=Priority Process H Blocks on down, if its partner process P in its critical region is normally Lo priority, then P should have its priority increased until its up is executed so that H can be unblocked soon which would be consistent with H’s Hi priority.

PRIORITY INVERSION
Soft Real Time Problem

© M C Paull 2005 17