Uniprocessor Scheduling

Arnav Gupta

March 25, 2024

Contents

1 Types of Processor Scheduling **2**
- 1.1 Long-Term Scheduling . . . 2
- 1.2 Medium-Term Scheduling . . . 3
- 1.3 Short-Term Scheduling . . . 3

2 Scheduling Algorithms **3**
- 2.1 Short-Term Scheduling Criteria . . . 3
  - 2.1.1 User-Oriented, Performance Related . . . 4
  - 2.1.2 User-Oriented, Other . . . 4
  - 2.1.3 System-Oriented, Performance Related . . . 4
  - 2.1.4 System-Oriented, Other . . . 4
- 2.2 Use of Priorities . . . 4
- 2.3 Alternative Scheduling Policies . . . 4
  - 2.3.1 First-Come-First-Served . . . 5
  - 2.3.2 Round Robin . . . 5
  - 2.3.3 Shortest Process Next . . . 6
  - 2.3.4 Shortest Remaining Time . . . 6
  - 2.3.5 Higher Response Ratio Next . . . 7
  - 2.3.6 Feedback . . . 7
- 2.4 Performance Comparison . . . 7
  - 2.4.1 Queueing Analysis . . . 8
  - 2.4.2 Simulation Modelling . . . 9
- 2.5 Fair-Share Scheduling . . . 9

3 Traditional UNIX Scheduling **10**

1 Types of Processor Scheduling

Long-term scheduling: decision to add to the pool of processes to be executed

Medium-term scheduling: decision to add to the number of processes that are partially or fully in main memory

Short-term scheduling: decision as to which available process will be executed by the processor

IO scheduling: decision as to which process's pending IO request shall be handled by an available IO device

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives (response time, throughput, processor efficiency).

Scheduling affect system performance since it determines which processes wait and which progress. Scheduling involves managing queues to minimize queueing delay and to optimize performance in a queueing environment.

1.1 Long-Term Scheduling

Controls the degree of multiprogramming since it determines which programs are admitted to the system.

A newly created process can be added to a queue for a short-term scheduler (admitted) or a medium-term scheduler (swapped-out).

Newly submitted jobs are routed to disk and held in a batch queue, which the long-term scheduler creates processes from, based on 2 decisions:

1. when the OS can take on one or more additional processes
    - driven by the desired degree of multiprogramming
    - more processes created, smaller percentage of time each process can be executed
    - each time a job terminates, scheduler may add 1+ jobs
    - if the fraction of idle time exceeds a threshold, scheduler may be invoked
2. which job(s) to accept and turn into processes

- can be based on first-come-first-serve or include priority, expected execution time, and IO requirements

For interactive programs in a time-sharing system, the OS accepts all users until the system is saturated.

1.2 Medium-Term Scheduling

Part of the swapping decision, based on memory requirements (memory management) and the need to manage the degree of multiprogramming.

1.3 Short-Term Scheduling

Executes most frequently, is invoked whenever an event occurs that may lead to the blocking of the current process or provides an opportunity to preempt a currently running process in favour of another. Such events are:

- clock interrupts
- IO interrupts
- operating system calls
- signals (ex. semaphores)

2 Scheduling Algorithms

2.1 Short-Term Scheduling Criteria

Objective: allocate processor time to optimize 1+ aspects of system behaviour

User-oriented criteria relate to the behaviour of the system as perceived by a user or process. (ex. response time - time between request submission and response output) **System-oriented criteria** have the focus on effective and efficient processor utilization. (ex. throughput - rate at which processes are completed) User-oriented criteria always important, system-oriented criteria only important if more than 1 user.

Performance-related criteria are quantitative and can be readily measured (response time and throughput). **Other criteria** are qualitative or cannot be readily measured (predictability).

Criteria are interdependent, cannot all be satisfied.

2.1.1 User-Oriented, Performance Related

Turnaround time: interval between submission of a process and completion (execution time plus time spent waiting for resources

Response time: time from submission of a request and when the response is received (request could still be being processed)

Deadlines: when process completion deadlines can be specified, scheduling should maximize the percent of deadlines met

2.1.2 User-Oriented, Other

Predictability: a job should run in the same amount of time and cost regardless of system load

2.1.3 System-Oriented, Performance Related

Throughput: maximize number of processes completed per unit time (how much work is being performed)

Processor utilization: percent of time the processor is busy

2.1.4 System-Oriented, Other

Fairness: processes should be treated the same, none starving unless otherwise specified

Enforcing priorities: scheduling policy should favour high-priority process

Balancing resources: scheduling policy should keep the resources of the system busy

2.2 Use of Priorities

Priority can be implemented with multiple queues in descending order of priority. Lower-priority processes may suffer starvation, which can be solved by having the priority of a process change with its age or execution history.

2.3 Alternative Scheduling Policies

Selection function: determines which process is selected next for execution, could depend on priority, resource requirements, or execution characteristics of the process.

- w is the time spent in system so far waiting
- e is the time spent in execution so far
- s is the total service time required by the process including e

Decision mode: specifies the instants in time when the selection function is exercised:

- **nonpreemptive**: a running process continues to execute until it terminates or blocks itself to wait for IO or some OS service
- **preemptive**: the currently running process may be interrupted by the OS, done when a new process arrives, when an interrupt occurs that readies a blocked process, or on a clock interrupt

Preemptive policies have more overhead but could have better service. Cost can be kept low with efficient process-switching and large main memory.

2.3.1 First-Come-First-Served

As each process becomes ready, it joins the ready queue. When the current process ceases to execute, the process that has been the queue the longest is selected for running.

Performs better for long processes than short ones. Favours processor-bound processes over IO-bound processes.

With priorities, can create an effective scheduler.

2.3.2 Round Robin

When an interrupt occurs, the current process is placed in the ready queue and the next ready job is selected on a FCFS basis.

Time-slicing: each process given a slice of time before being preempted

Principle design issue is length of time quantum: very short means too much overhead. Time quantum should be slightly greater than the time required for a typical interaction or process funciton.

Particularly effective in a general-purpose time-sharing system or transaction processing system. Drawback is that processor-bound processes tend to receive an unfair portion of processor time.

Virtual round robin uses an auxiliary queue for processes that are moved after being released from an IO block. Processes in the auxiliary queue get preference over those in the main ready queue. It then runs no longer than a time equal to the basic time quantum minus the total time spent running since it was last selected from the main ready queue.

2.3.3 Shortest Process Next

Nonpreemptive policy where the process with the shortest expected processing time is selected next.

Overall performance is improved in terms of response time, though response times vary more so less predictability. Further, difficult to know actual required processing time.

For interactive processes, the OS keeps a running average of each burst for each process. The calculation used is

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n$$

where T_i is the processor execution time for instance i of the process (total exec time for batch job, burst time for interactive job) and S_i is the predicted value for instance i

Preference is giving weight to more recent instances so **exponential averaging** is used:

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n$$

where α is a constant weighting factor. If α close to 1, the average will quickly reflect a rapid change in the observed quantity.

Exponential averaging tracks changes in process behaviour faster than simple averaging.

SPN could starve longer processes if enough shorter processes. Though, reduces bias in favour of longer jobs. Undesirable for certain environments because of lack of preemption.

2.3.4 Shortest Remaining Time

Scheduler always chooses the process that has the shortest expected remaining processing time.

Scheduler may preempt the current process when a new process becomes ready. Must have an estimate of processing time to select, and still a risk of starvation of longer processes.

Less overhead than round robin since no additional interrupts, though elapsed service time must be recorded. Faster turnaround time since shorter jobs can preempt longer ones.

2.3.5 Higher Response Ratio Next

Based on the ratio

$$R = \frac{w + s}{s}$$

where R is the **response ratio**, w is the time spent waiting for the processor, and s is the expected service time.

Scheduling rule is that when the current process completes or is blocked, choose the ready process with the greatest R value (accounts for process age).

Shorter jobs are favoured but longer processes will also eventually get past competing shorter jobs.

2.3.6 Feedback

SPN, SRT, HRRN cannot be used if no indication of relative process length.

Scheduling is preemptive and dynamic priority is used. Every time a running process is preempted, it is demoted to the next lower-priority queue. Within each queue other than the lowest-priority one, FCFS is used, otherwise round robin used. Known as **multilevel feedback**.

Turnaround time can stretch out alarmingly. Can cause starvation if new jobs enter system frequently, can compensate by varying time allowed to execute before preemption for each queue. Another remedy is to promote a process to a higher-priority queue after it spends some time in its current queue.

2.4 Performance Comparison

Relative performance depdends on distribution of service times, efficiency of scheduling and context switching mechanisms, nature of IO demand, and performance of IO subsystems.

2.4.1 Queueing Analysis

Any scheduling discipline that chooses the next item independent of service time obeys

$$\frac{T_r}{T_s} = \frac{1}{1-\rho}$$

where T_r is the turnaround time, T_s is the average service time, and ρ is the processor utilization.

For scheduling algorithms that account for service time, compared to FCFS, relative performance can be found by considering priority scheduling in which priority is based on service time.

For queues with 2 priority categories, assume:

1. poisson arrival rate
2. priority 1 items are services before priority 2 items
3. FCFS dispatching for items of equal priority
4. no item interrupted while being served
5. no items leave the queue (lost calls delayed)

λ is the arrival rate. The general formulas are:

$$\lambda = \lambda_1 + \lambda_2$$

$$\rho_1 = \lambda_1 T_{s1}; \rho_2 = \lambda_2 T_{s2}$$

$$\rho = \rho_1 + \rho_2$$

$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$

$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

If no interrupts, with exponential service times:

$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1+\rho_1}$$

$$T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1-\rho}$$

With preemptive-resume queueing discipline and exponential service times:

$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1}}{1-\rho_1}$$

$$T_{r2} = T_{s2} + \frac{1}{1-\rho_1}\left(\rho_1 T_{s2} + \frac{\rho T_s}{1-\rho}\right)$$

2.4.2 Simulation Modelling

Discrete-event simulation allows a wide range of policies to be modelled, but the results for a given run only apply to that particular collection of processes under a particular set of assumptions.

Results are presented by grouping processes into service-time percentiles so effects of policiies can be viewed as a function of process length.

2.5 Fair-Share Scheduling

In a multiuser system, it is attractive to make scheduling decisions based on sets of processes that compose an application which is **fair-share scheduling**. Similar for time-sharing systems, except with groups of users.

Each user assigned a weighting of some sort that defines the user's share of system resources as a fraction of total resource usage (a share of the processor).

The fair-share scheduler considers the execution history of a related group of processes and each process's execution history to make scheduling decisions. The system divides users into groups and allocates each group some processor.

Scheduling done on the basis of priority and accounts for the underlying priority of the process, its recent processor usage, and the recent processor usage of the group to which the process belongs. For a process j in group k:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

where $CPU_j(i)$ is the processor utilization by process j through interval i, $GCPU_k(i)$ is the processor utilization of group k through interval i, $P_j(i)$ is the priority of process j at the beginning of interval i (lower value is higher priority), $Base_j$ is the base priority of process j, and W_k is the weighting assigned to group k.

Each process has a base priority that drops as the process uses the processor and the group uses the processor. Greater the weight assigned to the group, the less its utilization affects its priority.

3 Traditional UNIX Scheduling

Primarily for time-sharing interactive environments, to provide good response time for interactive users while ensuring low-priority background jobs do not starve.

Uses multilevel feedback using round robin within each priority queue. System uses 1 second preemption, so if a running process does not block or complete within 1 second, it is preempted. Priority based on process type and execution history with the following formulas:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where *nice* is a user-controllable adjustment factor.

Priority of each process recomputed once per second. Band priority divides all processes into fixed bands of priority levels. CPU and *nice* are restricted to prevent a process from migrating out of assigned band. In decreasing order of priority, bands are:

- swapper
- block IO device control
- file manipulation
- character IO device control
- user processes

Provides most efficient use of IO devices. For user-processes band, execution history penalizes processor-bound processes, improving efficiency.