Determine how the state of a process evolve

long term scheduling -> determines the number of active processes

a new process is inactive until the long-term scheduler activates it

- when can a new process can be activated?
- which process to activate?

more active processes increase the execution times of the active processes (and requires more virtual mem)

So the scheduler will monitor the idle time of the processor and activate processes when the idle time becomes too high it will activate a process.

- The type of process is also important.
- Too many I/O-bound processes can lead to all process being blocked.
- interactive processes are often allowed to be active immediately

medium term scheduling -> determines which processes are (partially) present in physical memory.

non real-time short-term scheduling -> which of the ready processes may use the processor

turn-around time -> T_q how long to be completely executed (from when it was started)

It's desirable that the processor is almost always working. High utilisation often results in larger turn around time. So we have to make a compromise.

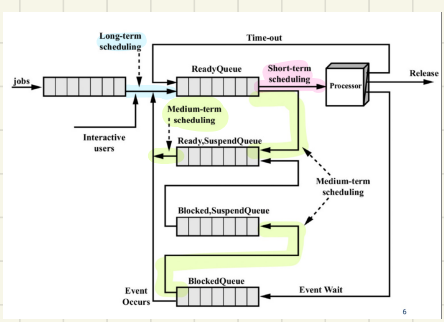
Fairness -> capacity of the resources is fairly distributed over the processes

predictability -> how sensitive the exec. time of a process is to the presence of other processes

Notation

$\alpha | \beta | \gamma$

- α: number of processes
- β: boundary conditions
- γ: function to minimize



- $\alpha = 1$ 1 processor $\alpha = Pm$ m identical processors
- β: r_j -> tasks with arrival times $\neq 0$
  pmtn -> tasks can be interrupted
  prec -> some tasks have to wait for others to finish
- γ: C_j -> completion time of task j
  we either consider $\gamma = \sum C_j$ or $\gamma = \sum w_j C_j$ <- weighted sum

$\sum U_j$ number of tasks that don't meet deadline

non real time short term scheduling algorithms

tasks can be interrupted -> preemptive systems.

First come first served (FCFS) (FIFO)

Serve the processes in the order they were placed in the queue

drawbacks:

- small tasks end up behind large processes
- I/O bound processes are disadvantaged, bcz everytime an I/O op. happens they go back to the queue

Round Robin (RR)

set a maximum time a process can have continuos access to the processor (quantum).

After this time is ended the tasks is interrupted and back to the ready queue

shorter quantum-> a lot of time spent on switching processes instead of executing them

Still disadvantage for I/O bound processes. A solution may be using a different queue for them.

the amount of time they are allowed is quantum - time they had access to the processor since the last time they were in the queue

Shortest process next (SPN)

choose the process that requires the least processor time.

optimal for $1||\Sigma C_j$ -> tasks arriving toghether at time t=0, no interruptions

not for $1|r_j|\Sigma C_i$

difficult to implement because requires prior knowledge of the running time of all processes.

Disadvantageus for large processes

proof of optimality

consider Turnaround time $T_q \sim \Sigma C_j$ p_j processing time of proc. j

$T_q = p_1 + (p_1+p_2) + \dots + (p_1+\dots p_N) = \sum_{i=1}^{N} (N-i+1)p_i$ optimal if $p_1 \leq p_2 \leq \dots \leq p_N$

↑ has to wait for p_1 to start

Weighted shortest process next (WSPN)

chooses processes according to p_j/w_j $w_j \geq 0$ is the weight

optimal $1||\Sigma w_j C_j$

aims at minimizing weighted turnaround time $T_q \sim \frac{1}{N} \Sigma w_j C_j$

$w_1 p_1 + w_2(p_1+p_2) \cdots + w_N(p_1+\dots p_N)$ $w_i(p_1+p_2 \cdots p_{i-1}+p_i)$

proof of optimality

consider order S optimal but not for WSPN, then $\frac{p_i}{w_i} > \frac{p_{i+1}}{w_{i+1}}$ -> $p_i w_{i+1} > p_{i+1} w_i$

turnaround time for i and i+1

$S = w_i(p_1+p_2+\dots p_{i-1}+p_i) + w_{i+1}(p_1+p_2+\dots+p_{i-1}+p_i+p_{i+1})$

now consider order S' obtained from switching processes i and i+1 in S

$w_{i+1}(p_1+p_2+\dots+p_{i-1}+p_{i+1})+w_i(p_1+p_2\cdots+p_{i-1}+p_{i+1}+p_i)$

⇓ switch for convenience

$S'=w_i(p_1+p_2\cdots+p_{i-1}+p_{i+1}+p_i)+w_{i+1}(p_1+p_2+\dots+p_{i-1}+p_{i+1})$

Now subtract : $S-S'$

$w_i(p_1+p_2+\cdots p_{i-1}+p_i)-w_i(p_1+p_2\cdots+p_{i-1}+p_{i+1}+p_i)=-w_i p_{i+1}$

$w_{i+1}(p_1+p_2+\cdots+p_{i-1}+p_i+p_{i+1})-w_{i+1}(p_1+p_2+\dots+p_{i-1}+p_{i+1})=w_{i+1}p_i$

$w_{i+1}p_i-w_i p_{i+1}>0$ → T_q decreased! ⇒ S was not optimal

Shortest remaining time next (SRTN)

tasks can be interrupted (assume int. itself doesn't waste time).

When new process becomes ready if it needs less running time than the remaining time of the current process, the current one gets interrupted and replaced with the new one.

optimal for optimizing turnaround time.

$1|r_j, pmtn|\sum C_j$

it requires perfect knowledge of the length of all tasks, also have to track the remaining exec. time. Starvation for longer tasks

proof of optimality

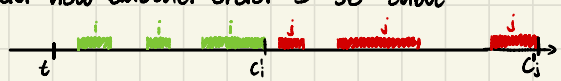
Consider order S optimal but not for SRTN, then

$\exists t$

$c'_j=\max(c_i,c_j)$

$c'_i<\min(c_i,c_j)$

consider now another order S' so that



Compare turnaround times

$$\sum_K C_K=\sum_{k\neq i,j} C_K+C_i+C_j>\sum_{k\neq i,j} C_K+C'_i+C'_j$$

T_q has decreased
S was not optimal

Nonpreemptive SRTN (nSRTN)

tries to minimize the running time when tasks are not interrupted

$1|r_i|\Sigma C_j$

given the optimal order $C_1 < C_2 \dots C_N$

construct same sequence with times $\bar{C}_1 + \bar{C}_2 + \dots \bar{C}_N$

possibly leave processor idle until next process arrives

its only possible to reach 2x of the optimal time (2x approximation)

proof

prove that $\Sigma_j \bar{C}_j \leq 2\Sigma_j C_j$

consider completition time C_j

$$\bar{C}_j \leq \max_{k \leq j} r_k + \sum_{k=1}^{j} p_k$$ maximal if the first task is the last of the j-task to arrive (has to wait)

for $k \leq j \rightarrow r_k \leq C_k \leq C_j \rightarrow \max_{k \leq j} r_j \leq C_j$ $\rightarrow \bar{C}_j \leq 2C_j$

Since tasks are processed in order $\rightarrow \sum_{k=1}^{j} p_k \leq C_j$

Highest response ratio next (HRRN)

each process has a weighted turnaround time equal to $RR = (w + p_s)/p_s$ w= time process has being waiting

The increase is proportional to the processing time p_s of the project (shorter increase faster)

When a process is released we choose the one with the highest RR. Short processes advantaged.

Requires at least estimates of times

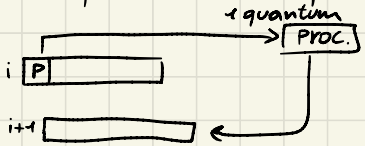
Multilevel feedback scheduling

look at how much time a process already had at its disposal

We allow processes that had access to the processor less frequently

An implementation uses n ready queues with a priority mechanism.

Process in queue i can access processor only if the i-1 queues are empty.



All queues FIFO except last (Round Robin)

Short tasks are favored. The quantum is increased as i increases.

I/o bound processes always in queue 1

Traditional Unix Scheduler

Similar to multilevel feedback queue.

Time divided into equal intervals, each process j has a priority $P_j(i)$ during interval i.

The lower the value the higher the priority.

The scheduler resolution indicates how often a clock interrupt is generated per second (typically 60/s)

For each process j a counter $U_j(i)$ is kept, it indicates how often the process was active during i

at each interrupt the counter of the current process increased by 1.

At the end of each interval we calculate the average CPU usage

$$CPU_j(i) = U_j(i)/2 + CPU(i-1)/2$$

then we adjust the priority

$$P_j(i) = Base_j + CPU_j(i-1)/2 + nice_j$$

Fair-share scheduler

Add another $GU_K(i)$ per group K. Increase as before if a process from group K was active.

$$GCP\ U_K(i) = G\,U_K(i)/2 + GCPU_K(i-1)/2$$

$$P_j(i) = Base_j + CPU_j(i-1)/2 + GCP\,U_k(i-1)/4W_K + nice_j$$

W_k is a group weight, $\Sigma W_K = 1$ percentage of use of CPU for group K

Real time short time scheduling

deadline d_i for process i

Divided in static and dynamic → online

↳ offline (taken when system not operational)

1) static table-driven solutions: table created in advance that perfectly represents which tasks will be executed when. If not all can meet their deadlines some processes will not be executed

2) static priority-driven solutions: each task is given a priority as a result of a scheduling algorithm

With dynamic solutions decisions regard the acceptance of new tasks, this can be done "best effort"

Not all tasks actually have a deadline.

Some deadlines may be hard (never to be crossed) or soft (small crossing is solvable)

Deadline Scheduling

Assume we know all processes, their duration p_i and deadline d_i.

C_j → completition time of j

$U_j = 1$ if $C_j > d_j$ task j not completed in time, otherwise $U_j = 0$

We are interested in the number of tasks that fail the deadline ($\sum U_j$)

or the maximum degree to which a task is late ($L_{max} = \max_{j=1}^{n}(C_j - d_j)$)

those add up to Y (function we wish to minimize)

Earliest deadline first (EDF) - Earliest Due Date (EDD)

schedule tasks according to ascending deadline

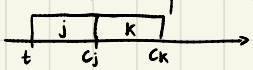
task j has deadline d_j

$L_{max} = \max_j(C_j - d_j)$ $1||L_{max}$

if deadlines are feasible ($L_{max} < 0$) → EDF algo will meet all of them

proof of optimality

consider order S optimal but not for EDF, then



with $d_K < d_j$ (not optimal for EDF)

then → $C_j = t + p_j$, $C_K = t + p_j + p_K$

Consider order S' switching j and K, then

$C'_K = t + p_K$, $C'_j = t + p_K + p_j$

considering $d_K < d_j$ $C_j = t + p_j$ // $C'_j = t + p_K + p_j$

$C_K = t + p_j + p_K$ $C'_K = t + p_K$

then: since $d_j > d_K$ it's less if subtract

$C'_j - d_j = C_K - d_j < C_K - d_K$ simply added this

$C'_K - d_K = t + p_K - d_K <$ then obviously it's greater $t + p_K + p_j - d_K = C_K - d_K$

in general $\max_i C'_i - d'_i \leq \max_i C_i - d_i$ → switching i and j did not increase $\max_i C_i - d_i$

S' closer to EDF, can be better with different j, K

proof of deadline feasibility

All processes are realizable if $L_{MAX} = \max_j (C_j - d_j) \leq 0$

every deadline is feasible IFF

$$U_i = \frac{1}{d_i} \sum_{k:d_k \leq d_i} \rho_k \leq 1 \quad \text{with } U = \max U_i \text{ (load factor)}$$

$\frac{1}{d_i}$ · sum of exec time of processes before p_i

if $U_i \leq 1$ -> all deadline feasible

else if $U > 1$ -> minimize total of unmet deadlines ΣU_j

Moore-Hodgson (MHA) $1||\Sigma U_j$

tries to minimize the number of tasks that don't meet deadlines ΣU_j

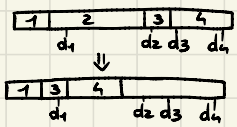
1) Sort tasks J_i based on EDF order

2) check if there is task that doesn't meet deadline, if yes continue

3) J_α first late task

remove J_β with $P_\beta = \max_{j \leq \alpha} p_j$

return to 2)



Real time short time scheduling algorithms

release/arrival time

in general $1|r_j|\Sigma U_j$ is NP-hard

EDF optimal for $1|r_j, pmtn|\Sigma U_j$

$U_i = \frac{1}{d_i} \sum_{k:d_k \leq d_i} \rho_k \leq 1$ not usable anymore $\Longrightarrow$ $U_{[t_1,t_2]} = \frac{1}{t_2 - t_1} \sum_{\substack{j: t_1 \leq r_j \\ d_j \leq t_2}} \rho_j$

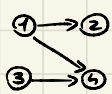
load $U = \max U_i \Longrightarrow U = \sup_{t_1 < t_2 \leq \infty} U_{[t_1,t_2]}$

load $U \leq 1 \Longleftrightarrow$ all deadlines feasible

precedence -> **Lawler's algorithm**

applicable when precedences in graph $G(V,E)$

optimal for $1|prec|L_{Max}$



$p_1 = 1$ $p_2 = 2$ $p_3 = 2$ $p_4 = 1$ Proc. times

$d_1 = 5$ $d_2 = 2$ $d_3 = 3$ $d_4 = 4$

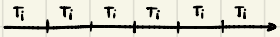
1) Assume

- G = precedence conditions, num = $|V|$
- $U(G)$ = nodi pozzo
- $T = \Sigma p_j$ total execution time

2) choose task j in $U(G)$ with d_j maximal (or $T - d_i$ minimal)

- task j gets number num
- num--, $T = T - p_j$ remove j's node from G
- redefine $U(G)$, repeat

scheduling periodic tasks



period of task i is T_i

each task ∞ instances with $r_j = jT_i$

instance has relative deadline T_i

EDF can handle n periodic tasks IFF

- $U = \sum_i \frac{\rho_i}{T_i} \leq 1$ ($\frac{\rho_i}{T_i}$ = utilisation of task i)

prove it by showing that $v = U$, EDF optimal for periodic tasks

proof

consider $v = \sup_{t_1 < t_2 \leq \infty} v_{[t_1,t_2)}$, $v_{[t_1,t_2)} = \frac{1}{t_2 - t_1} \sum_{\substack{j: t_1 \leq r_j \\ d_j \leq t_2}} \rho_j$

$U = \sum_{j=1}^{n} \rho_j / T_j \leq 1$ — it's this but whithout $\lfloor\ \rfloor$, so larger

show that $v \leq U$ for $t_1 < t_2$

$\sum_{j=1}^{n} \frac{t_2 - t_1}{T_j} \rho_j$

how many tasks i can fit in period

$v_{[t_1,t_2)}(t_2 - t_1) = \sum_{\substack{j: t_1 \leq r_j, d_j \leq t_2}} \rho_j \leq \sum_{j=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_j} \right\rfloor \rho_j \leq (t_2 - t_1) U$ (period)

show that $v \geq U$ by replacing $t_1 = 0$ and $t_2 = H$ (H smallest number divisible by T_j)

$v_{[0,H)} H = \sum_{\substack{j: 0 \leq r_j \\ d_j \leq H}} \rho_j = \sum_{j=1}^{n} \frac{H}{T_j} \rho_j = HU$ since v is the suppremum $\rightarrow v \geq U$

Rate monotonic scheduling

combining periodic real-time tasks with non real-time ones

Handle classical Unix scheduler with added priorities (real time ↑, each one fixed)

organize tasks so that $T_1 \leq T_2 \leq \dots T_n$ (more frequent)

returns tasks with period T_i priority i (max 1)

RMS respects all deadline certainly if

$U = \sum_i \frac{\rho_i}{T_i} \leq n(2^{1/n} - 1)$ $\quad n \to \infty$ $\quad 2^{1/n} - 1$ drops to $\ln 2 = 0.693$

RMS optimal in static priority-driven systems (relative deadlines = period)

RMS will meet all deadlines if all periods are multiple of each other and $U \leq 1$