Round-Robin (Time Slicing) Scheduling

In Round-Robin (RR) scheduling, each ready process is run for a specific time slice (or quantum), then execution switches to the next ready process in a ready list (or queue), even if the first process has not ended.

The quantum (q), determined by a clock interrupt is a critical parameter in the algorithm. It should be neither too small nor too large (typical compromise value is 10 to 100 ms).

If there are n ready processes, each process is assigned 1/n of the CPU time.

Round-Robin (Time Slicing) Scheduling

To solve the examples of previous lecture using RR scheduling, we assume the following:

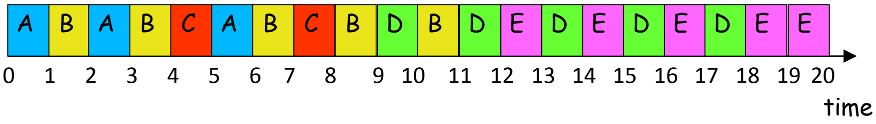
- Ready processes are placed in a linked list with last node pointing to the first node. A moving pointer determines the next process to run. New ready process is inserted at the end of the list.
- At the end of a quantum, updating of list (removing or inserting nodes) is done before deciding the next process to run.
- q=1 and time of context switching is negligible.

Round-Robin (Time Slicing) Scheduling

Example 1 of previous lecture

Process	a_i	p_i	t_i	w_i
A	0	3	6	3
B	1	5	11	5
C	3	2	8	3
D	9	5	18	4
E	12	5	20	3

$w_{av} = 3.6 \quad (w/p)_{av} = 0.98$



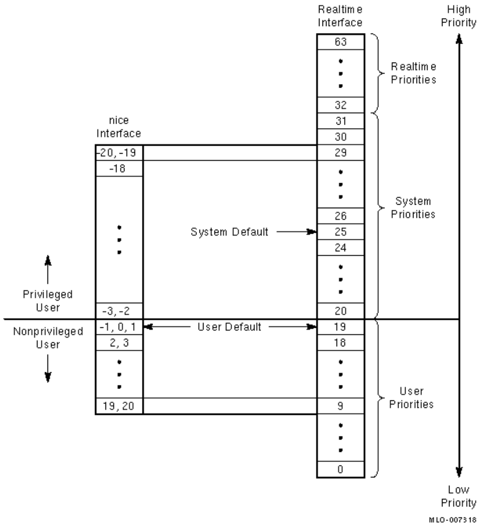
Priority Scheduling

In any system, some processes must be given a higher priority. For example:

- ❑ System (kernel) processes.
- ❑ Real-time processes.
- ❑ Processes holding many resources.

A priority level is associated with each process (thread). The priority levels should be reflected in the scheduling decisions.

Priority Scheduling



Priority Scheduling

Priority scheduling can be either:

- Non-preemptive: e.g. add a new high priority process at the head of the ready queue.
- Preemptive: e.g. an arriving high priority process causes a preemption of currently running low priority process.

A potential problem will be the “*starvation*” (indefinite postponement) of low priority processes. This can be avoided by allowing variable or *dynamic* priorities. For example:

- As process waits more, its priority level increases.
- Alternatively, as process consumes more CPU time, its priority level decreases.

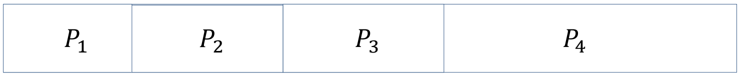
Multilevel Queue Scheduling

- n priority levels with n corresponding ready queues.
- Processes with highest priority run first.
- If several processes have the same priority level, they are scheduled using round-robin (or other suitable algorithm).
- Priorities are changed dynamically: process moves between queues according to its recent CPU behavior (multilevel feedback queue scheduling).

Shortest Process Next (SPN) Scheduling

Start with the process that has the shortest process time. Run it until it is terminated or blocked (i.e. SPN is non-preemptive).

For a set of processes that arrive at the same time, SPN minimizes the average waiting time.



$$w_{av} = \frac{3P_1 + 2P_2 + P_3}{4}$$

Which order will minimize w_{av}?

Shortest Process Next (SPN) Scheduling

However, for normal interactive systems, process time cannot be known a-priori.

One possible approach is to approximate SPN by trying to estimate the length of the next process *CPU burst*. For example:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$$

where t_n is the measured length of the nth CPU burst.

τ_n is the estimated length of the nth CPU burst.

α is a weighting factor, $0 \le \alpha \le 1$

Shortest Remaining Time (SRT) Scheduling

A preemptive version of SPN is the Shortest Remaining Time (SRT) Next scheduling algorithm.

An arriving short process may preempt a process with longer remaining processing time.

This algorithm minimizes the average waiting time for any values of arrival time (if context switching time is neglected).

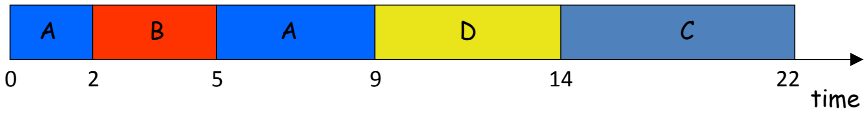
At which instants will the system need to decide which process to run next?

Shortest Remaining Time (SRT) Scheduling

Example using SRT

Process	a_i	p_i	t_i	w_i
A	0	6	9	3
B	2	3	5	0
C	5	8	22	9
D	7	5	14	2

$w_{av} = 3.5$



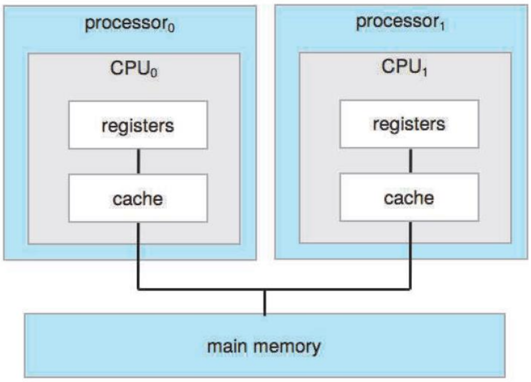
Multiprocessor Scheduling

Currently, architectures in most applications include multiple processors. In the following, we assume that processors are identical, running at the same clock frequency, and have access to shared main memory.

Multicore processor chips became widely used when it was not possible to increase the speed of single processors without consuming too much power. Having a multicore architecture will not speed up the execution of a given program unless it is divided into parallel threads.

Multiprocessor Scheduling

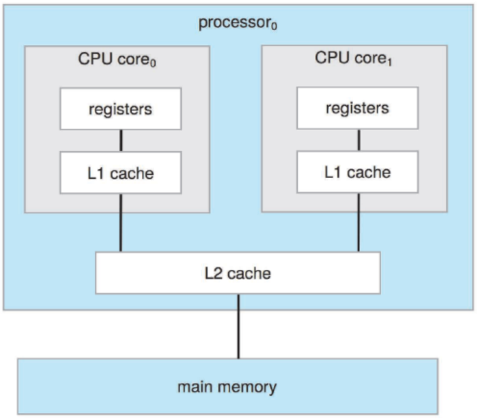
Symmetric Multiprocessing (SMP) architecture



Source: [Silberschatz 18]

Multiprocessor Scheduling

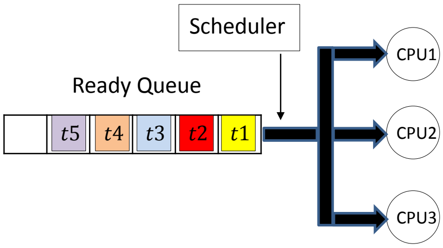
A dual core chip design



Source: [Silberschatz 18]

Single Scheduler – Single Ready Queue

Single CPU scheduler can be extended by having a single scheduler that assigns threads from a global ready queue to CPUs.



Advantages:

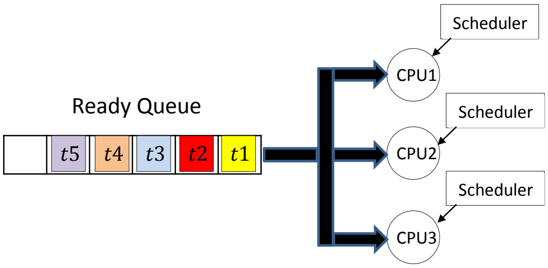
- Single CPU scheduling algorithms can be directly extended to the case of multiple CPUs.
- We can ensure the best use of the processing capacity. E.g. No CPU remains idle while there are some ready tasks.

Single Scheduler – Single Ready Queue

Disadvantages:

- As number of processors increase, load on single scheduler will be higher.
- A thread will repeatedly become blocked or preempted then will run again. *Migrating* a thread from one CPU to another usually causes the overhead of refilling the cache.

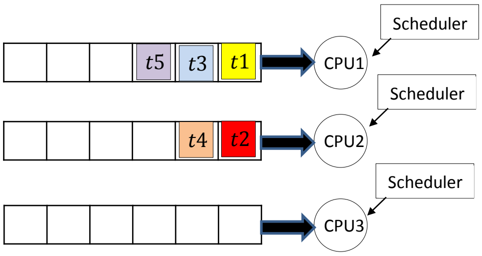
Multiple Schedulers – Single Ready Queue



Having multiple schedulers will reduce the load of a single common scheduler.

However, the single ready queue is a shared resource and conflicts of its use will result in longer delays as number of schedulers increase.

Multiple Schedulers – Multiple Ready Queues



In this partitioned scheduling approach, each new thread is assigned to a particular CPU during its lifetime. No migration takes place.

System will have a particular assignment policy: e.g. assign a new thread to the least loaded CPU (not always possible to determine accurately).

Multiple Schedulers – Multiple Ready Queues

Advantages:

- Less overheads on schedulers and hence faster operation.
- Processor *affinity* (i.e. thread linked always on the same CPU) will reduce time spent in cache refilling.

Disdvantages:

- Difficult to achieve load balancing: one CPU may be overloaded while another one is idle. Optimal use of processing capacity will not be achieved.