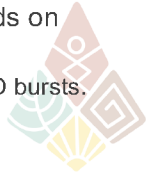
5 – Task Scheduling

5 – Task Scheduling

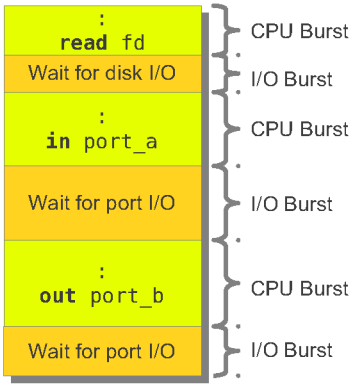
- Basic scheduling concepts
- Scheduling algorithms
- Selecting an algorithm for a particular system

Basic Concepts

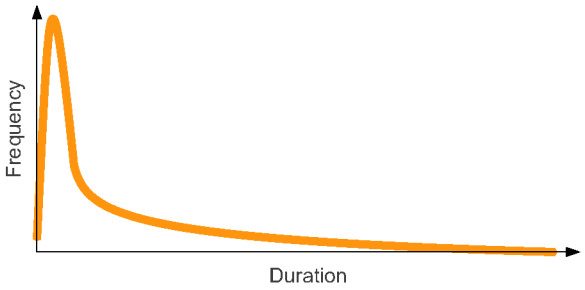
- Task scheduling is a fundamental function of every operating system.
- A process is executed until it must wait for I/O completion.
  - This means CPU is idle and an operating system should give the CPU to another process that want to execute the code.
- The success of task scheduling depends on property of processes.
  - e.g, alternate between CPU bursts and I/O bursts.



CPU-I/O Burst Cycle



(cont'd.)

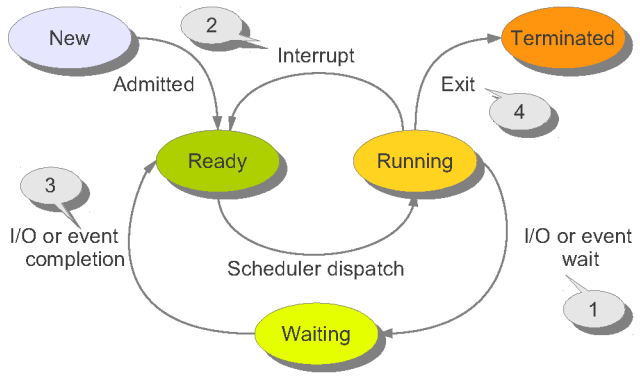


- Generally, CPU burst distribution is exponential or hyper-exponential.
  - Many short CPU bursts, few long CPU bursts

Task Schedulers

- Carried out when a computing resources is idle.
- Because there are many short bursts, it should be a short-term scheduler.
- Also, the algorithm should be optimized for those short bursts.
- But, before that, we need to know one more concept – **'When'** the task-scheduling decisions may take place ?

(cont'd.)



(cont'd.)

- For 1 and 4, no choice, the scheduler **must** select a process for execution.
  - This is called *non-preemptive*.
  - A process keeps the CPU until terminating or switching to the waiting, e.g. I/O.
  - No special hardware needed.
  - MS Windows 3.1, Older versions of Mac OS
- For 2 and 3, the scheduler **may** switch from one process to another. This is called *preemptive*.
  - Requires special hardware, e.g., timer interrupt.
  - May introduce deadlocks, inconsistencies, ...
  - What about processing of system calls in the kernel ?

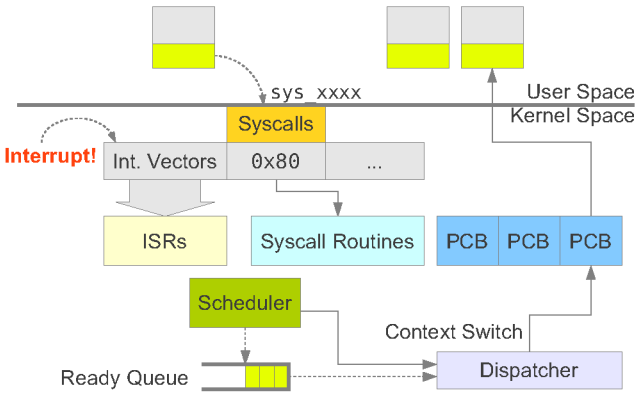
(cont'd.)

- To avoid any inconsistencies, most UNIX waits for a system call to complete, or for an I/O block to take place, before doing a context switch.
  - Kernel will not preempt the process while kernel data structure is in inconsistent state.
  - Safe, but bad for real-time computing.
- Another problem – an interrupt can occur at any time, and the kernel must service immediately.
  - Interrupt service routine must not be used simultaneously by several processes.
  - A simple solution is to disable the interrupt when enter the service routine, and re-enable when exit.
    - This can be slow.

Dispatchers

- A component to give control to the selected process.
- This involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart the program.
- Must be **very** fast
  - Remember ? context switch is totally wasteful.
- The time it takes to stop one process and start another is known as the *dispatch latency*.

Putting things together,



Scheduling Criteria

- Different CPU-scheduling algorithms have different properties. Many criteria have been suggested for comparing algorithms:
  - **CPU Utilization** – keep the CPU as busy as possible. In a real system, it should range from 40% to 90%.
  - **Throughput** – the number of processes completed per time unit.
  - **Turnaround time** – total time spent to complete a process.
  - **Waiting time** – total waiting time a process spends.
  - **Response time** – for interactive system, it is the amount of time it takes to **start** responding for a request or an event.

(cont'd.)

- Generally, we want to ..
  - Maximize CPU utilization,throughput
  - Minimize turnaround time, waiting time, response time
- Most of the cases, we optimize the average measures. But, it is not all the cases, e.g.,
  - We might want to minimize the maximum response time for some system, e.g., soft/hard real-time.
  - For interactive system, some analysts suggest that minimizing the **variance** of response time is more important than minimizing average response time.
  - A system with more **predictable** response time may be desirable than a system that is faster but highly variable.

Scheduling Algorithms

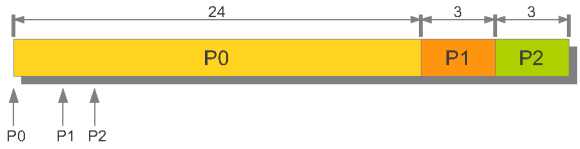
- First-Come, First-Served
- Shortest-Job-First
- Priority
- Round Robin
- Multilevel Queue
- Multilevel Feedback Queue

First-Come First-Served

- The simplest, by far
- The first process that requests the CPU first is allocated the CPU first.
- Simply implemented with a FIFO queue
- The average waiting time is often quite long, e.g.,

Process	Arrive.	Burst
P0	0	24
P1	2	3
P2	3	3

(cont'd.)



P0 arrives at T0, no wait.

P1 arrives at T2, waits until T24

P2 arrives at T3, waits until T27

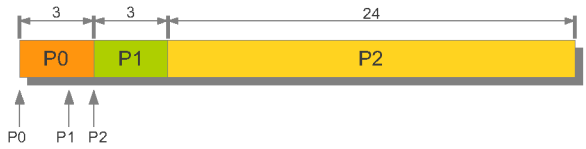
Average waiting time:

$$\frac{(0-0)+(24-2)+(27-3)}{3}=15.33$$

(cont'd)

- One more example:

Process	Arrive.	Burst
P0	0	3
P1	2	3
P2	3	24



- Average waiting time = 1.33

(cont'd.)

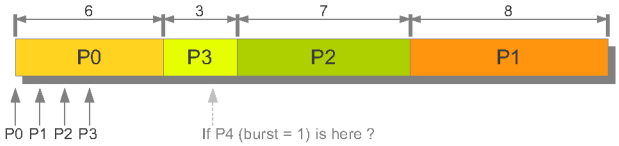
- *Convoy Effect* – all other processes wait for the one big process to get off.
- FCFS is non-preemptive.
  - A process keeps the CPU until it releases either by terminating or I/O.
  - Not suitable for time-sharing system.

Shortest-Job First

- Shortest CPU burst is chosen first
  - If two processes have the same length of CPU burst, use FCFS.

Process	Arrive	Burst
P0	0	6
P1	1	8
P2	2	7
P3	3	3

(cont'd.)



- At T6, there are 3 processes to be scheduled. So, P3 is chosen.
- At T9, 2 processes left, P2 is chosen.
- Finally, at T16, P1.
- Average waiting time:

$$\frac{(0-0)+(9-2)+(6-3)+(16-1)}{4}=6.25$$

(cont'd.)

- Optimal, give the minimum average waiting time
- Need to know length of the next CPU burst.
- Used frequently in long-term scheduling.
  - Users specify the length of the CPU burst.
- **Cannot** be implemented in short-term scheduling
  - There is no way to know the length of the next CPU burst accurately, but it can be predictable.
- Generally, the next CPU burst is predicted as an *exponential average* of the measured lengths of previous CPU bursts.

(cont'd.)

- Let t_n be the length of the nth CPU burst, then the predicted value, τ_{n+1}, is

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

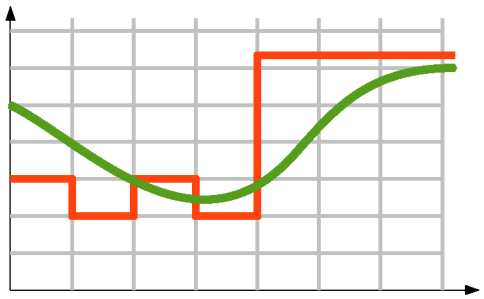
where $0 \leq \alpha \leq 1$

- This defines an *exponential average*.
  - t_n is the most recent actual/accurate information.
  - τ_n is the history.
  - α is the weight.

(cont'd.)

- e.g., $\alpha = ½$ and $\tau_0 = 10$

Burst	6	4	6	4	13	13	13
Predicted	8	6	6	5	9	11	12



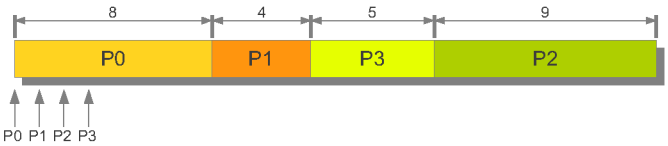
(cont'd.)

- SJF can be either *preemptive* or *non-preemptive*.
  - A newly arrived process may preempt the currently running process if the new one has a shorter burst.

Process	Arrive	Burst
P0	0	8
P1	1	4
P2	2	9
P3	3	5

(cont'd.)

- Non-preemptive

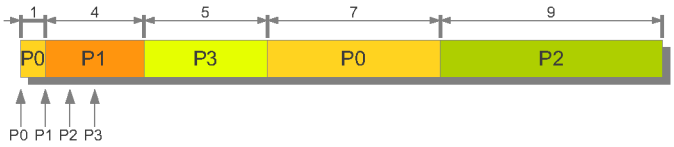


Average waiting time:

$$\frac{(0-0)+(8-1)+(17-2)+(12-3)}{4}=7.75$$

(cont'd.)

- Preemptive SJF is sometimes called *shortest-remaining-time-first*.



Average waiting time:

$$\frac{((0-0)+9)+(1-1)+(17-2)+(5-3)}{4}=6.5$$

- By the way, why is it exponential ?
  - Try to expand τ_{n+1} :)

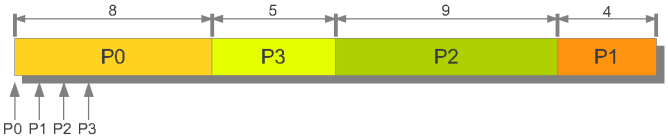
Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
  - SJF is a special case of priority-scheduling algorithm.
- Priorities are generally some fixed range of numbers.
  - e.g., 0 – 7, 0 – 4095
  - There is no general agreement about this.
  - Some systems use low number to represent low priority, the other use it for high priority.

(cont'd.)

- A

Process	Arrive	Burst	Priority
P0	0	8	4
P1	1	4	3
P2	2	9	2
P3	3	5	1



(cont'd.)

- Priority can be preemptive or non-preemptive.
  - Preemptive priority scheduler will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- The major problem of priority scheduling is *indefinite blocking* or *starvation*.
  - Lower-priority processes may have to wait indefinitely.
    - Rumor: when MIT shut down IBM 7094 in 1973, they found a low-priority process submitted in 1967 and had not yet been run.
  - *Aging* is a technique to solve this problem.
    - Increase priority of processes that wait in the system for a long time.

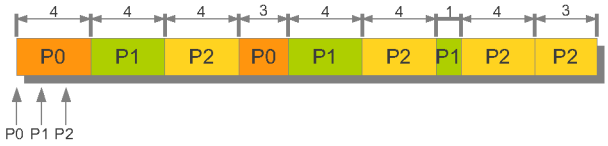
Round-Robin

- Specially-designed for time-sharing systems.
- FCFS + preemption to switch between processes.
- The ready queue is treated as a circular queue.
  - That's why each queue has the head and tail
- CPU is allocated for each process for a time up to 1 *time quantum* or *time slice*.
  - Time quantum is generally from 10 to 100 msec.
  - A process may have a burst less than 1 time quantum, and will release CPU voluntarily.
- Like FCFS, the average waiting time of RR is often quite long.

(cont'd.)

- Time quantum = 4

Process	Arrive.	Burst
P0	0	7
P1	1	9
P2	2	15



Average waiting time = 11.33

(cont'd.)

- Imply waiting time for each process $\leq (n-1) \times q$, where n is the number of running processes, and q is the time quantum
- In RR, the effect of context switching must also be considered.
  - It switches contexts frequently.
  - Time quantum >> context-switching time.
- Turnaround time depends on the time quantum.
  - Increasing the time quantum does not necessary decrease the average turnaround time.

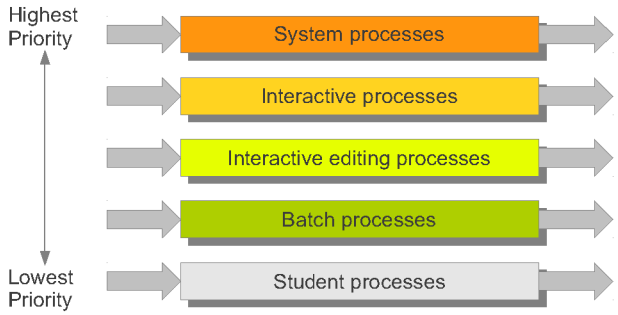
(cont'd)

- Exercise: find average turnaround time
  - Time quantum = 2, 3, 4, 5, 6

Process	Arrive	Burst
P0	0	6
P1	1	3
P2	2	1
P3	3	7

Multilevel Queue

- For situation in which processes can be classified into different groups, e.g.,

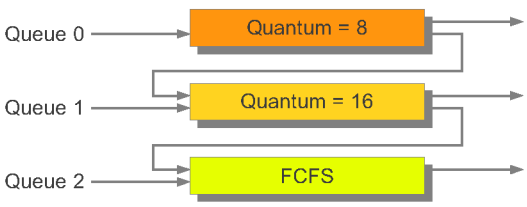


(cont'd.)

- Different scheduling/queue can be used for different group of processes.
- Each process is assigned permanently to one queue.
- There must be scheduling among these queues.
  - Generally, a fixed-priority preemptive scheduling.
  - Or, alternatively, time slice among the queues.

Multilevel Feedback Queue

- Allow processes to move between queues.
  - Separate processes with different CPU-burst characteristics, e.g., if a process uses too much CPU time, move to lower-priority queue, and vice versa.
    - A kind of aging.



(cont'd.)

- A scheduler executes all processes in queue 0.
  - If a process in queue 0 does not finish its job in 8 time units, it moves to queue 1.
    - Queue 1 will be executed only if queue 0 is empty.
  - If a process in queue 1 does not finish its job in 16 time units, it moves to queue 2.
    - Queue 2 will be executed only if queue 0 and 1 are empty.
- Processes of CPU bursts ≤ 8 time units can finish their job quickly, and go off to its next I/O.
- Processes of CPU bursts ≤ (8 + 16) time units can also finish their job quickly.
- Longer processes are sunken to queue 2.

(cont'd.)

- Multilevel queue is the most general CPU-scheduling algorithm.
  - It can be configured to match any system design.
  - It also requires some means of selecting values for all parameters to define the best scheduler.
    - Number of queues
    - Scheduling algorithm for each queue
    - Method to promote to higher-priority queue
    - Method to demote to lower-priority queue
    - Method to determine which queue a process will enter when it needs service.
    - etc.
  - Thus, it is also the most complex.

Algorithm Evaluation

- First, what criteria are we considered ?
  - CPU utilization
  - Response time
  - Throughput
  - etc. etc. etc.
- Next, what values do we focus ?
  - Average
  - Max
  - Min
  - etc. etc. etc.

(cont'd.)

- Methodologies – What model do we use to evaluate ?
  - Deterministic
  - Queuing
  - Simulation
  - Implementation

Deterministic Models

- A kind of *analytical evaluation*.
- Take a predefined workload, then define the performance of each algorithm for that workload.
  - It's what we did so far.
- Deterministic models
  - Simple and fast
  - Exact values for a particular input
  - Good to describe scheduling algorithm
  - Its answers apply to only those cases.
    - Too specific to be useful.

Queuing Models

- Model the system into services and queues
- Determine the arrival rates and service rates
- Use *queuing-network analysis* for evaluation
  - CPU / ready queue
  - I/O / device queue
  - etc. etc.
- *Little's theorem*: let n be the avg. queue length, W be the avg. waiting time, and λ be the avg. arrival rate. Thus,

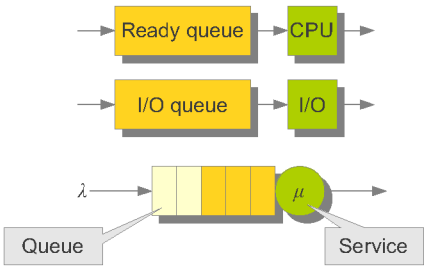
$$n = \lambda \times W$$

(cont'd.)

- Little's theory can be used to determine one of the three variables if we know the other two, e.g.,
  - Avg. arrival rate = 7 processes/sec
  - Avg. queue length = 14 processes
  - Then, avg. waiting time = 14/7 = 2 sec.
- The Little's theorem shows that relationships among the three variables are independent from any statistical distributions.
  - e.g., regardless of arrival behaviors of processes, how processes are enqueued.
  - This greatly simplifies queuing analysis, but, what if we want to know more that just these three ?

(cont'd.)

- A more complex queuing analyses can be done by employing the *queuing theory*.



(cont'd.)

- The queuing theory can be used to analyze scheduling algorithms, but
  - It is difficult to work with mathematics of complicated algorithms.
  - The arrival and service distributions are often defined in unrealistic ways.
  - Still, assumptions and/or approximations have to be made to simplify the analysis.
    - Accuracy ?

Simulations

- Program the model
- Feed a large number of workloads
  - Random data
    - Uniform, Exponential, Poisson distribution ?
  - Collected actual data
- Collect the results
- Statistically determine the results
  - Quite acceptable accuracy if it is done properly.
- Expensive
  - Program development
  - Time to simulate
  - Storage to maintain feeds and results

Implementation

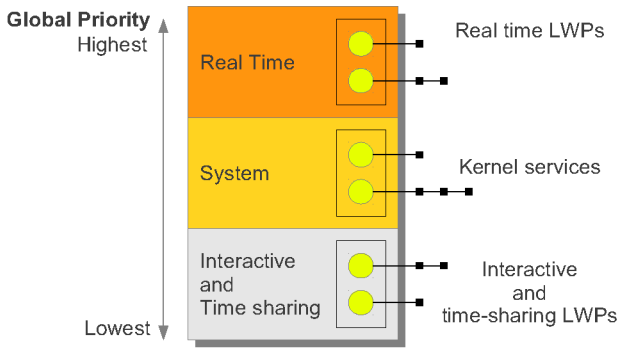
- Still, the simulation is of limited accuracy.
- The most accurate way to evaluate the system performance.
- Very expensive

Solaris 2

- 4-class priority scheduling
  - Real time – response within a bounded period of time.
  - System – kernel processes, e.g., paging daemon
  - Time sharing – default
  - Interactive
- Each of these 4 classes includes a set of priorities. The scheduler converts the class-specific priorities into global priorities, and select to run the threads with the highest priority until
  - It blocks
  - Its time slice expired
  - It is preempted by a higher-priority thread

(cont'd.)

- Multiple threads have the same priority → RR.



MS Windows 2000/XP/...

- Priority-based preemptive scheduling algorithm.
- Ensure that the highest priority threads will always run.
- The *dispatcher* handles scheduling.
- 32-level priority scheme, divided into
  - Real-time class (16 – 31): soft real-time.
  - Variable class (1 – 15): priority can be changed.
  - Memory management runs at priority 0.
- Dispatcher traverses the set of queues from the highest to the lowest until it finds a thread that is ready to run.

(cont'd.)

- If there is no ready thread, the *idle thread* is executed.
- In Win32 API there are 6 priority classes
  - `REALTIME_PRIORITY_CLASS`
  - `HIGH_PRIORITY_CLASS`
  - `ABOVE_NORMAL_PRIORITY_CLASS`
  - `NORMAL_PRIORITY_CLASS`
  - `BELOW_NORMAL_PRIORITY_CLASS`
  - `IDLE_PRIORITY_CLASS`

(cont'd.)

- Each class has a relative priority:
  - TIME_CRITICAL
  - HIGHEST
  - ABOVE_NORMAL
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE
- Priority class + relative priority can be converted to the 32-level priority.

(cont'd.)

- To give a good response time for interactive threads
  - When a thread is interrupted, and is in the variable-priority class, its priority is lowered.
  - When a thread is released from wait, its priority is boosted.
    - The amount of boosts depends on what the thread was waiting for, e.g., keyboard I/O gets a large boost while disk I/O gets a moderate one.
- When a process becomes a *foreground process,* the quantum is increased typically by the factor of 3.
  - Three times longer to run before preemption.

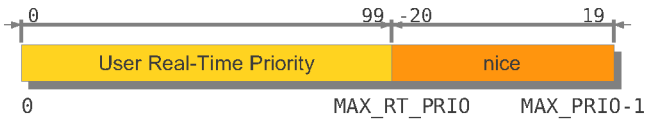
Linux

- Priority-based scheduling
- Two levels of priority schemes
  - Nice: -20 to +19
    - Nice means .. nice :)
  - User real-time priority: 0 to 99
    - Based on POSIX.1b
  - Configurable/controllable via system calls.
- From `include/linux/sched.h`

```
#define MAX_USER_RT_PRIO 100
#define MAX_RT_PRIO      MAX_USER_RT_PRIO
#define MAX_PRIO         (MAX_RT_PRIO + 40)
```

Linux – Priority

- The two schemes are combined to a single priority scheme for scheduling in kernel space.
  - `0` to `MAX_RT_PRIO – 1` to map user real-time priority
  - `MAX_RT_PRIO` to `MAX_PRIO – 1` to map nice.



- Linux combines these in `effective_prio()`

Linux – Scheduling Policies

- From `include/linux/sched.h`

```
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
```

- Normal processes uses `SCHED_NORMAL`
- (Soft) Real-time processes explicitly specify `SCHED_FIFO` or `SCHED_RR`
- Batch processes are treated as `SCHED_NORMAL` that never sleep, i.e, no I/O wait.

(cont'd.)

- The `SCHED_RR` and `SCHED_FIFO` will always be scheduled before `SCHED_NORMAL` and `SCHED_BATCH`.
- The `SCHED_RR` uses round robin.
- The `SCHED_FIFO` uses FIFO.
  - Basically, it is identical to `SCHED_RR` without time slices.
- Round robin is used to resolved processes with the same priority.

Linux – More about priority

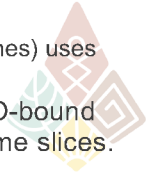
- The `effective_prio()` is also a wrapper to dynamically adjust scheduling priority based on the policies and process behaviors.
- If a process spends more time in I/O wait, then it might be I/O bound.
  - I/O-bound cycle is usually alternations of a long I/O wait followed by a short CPU burst, e.g., to process the I/O data.
  - Linux will increase scheduling priority of such processes
- If a process spends more time on CPU, then it is CPU bound. Linux will decrease its priority.

(cont'd.)

- This means I/O-bound processes would get CPU more than CPU-bound processes ? Is it a good approach ?
  - Well, it is reasonable since I/O-bound processes take very short CPU burst. So, the rest of CPU time can be given to those CPU-bound lower-priority processes.
  - CPU-bound processes want longer CPU time, not be scheduled more frequent.

Time Slices

- Linux takes time-sharing approach, so time slice must be defined.
- It is hard to define a time slice that fits all the cases.
  - Longer time slices provide better utilization but sacrifice the responsiveness.
  - Shorter time slices help to get better responses, but bad utilization.
  - Still, most OSes (especially for desktop ones) uses short time slices, e.g. 20 msec.
- Linux already increases priorities for I/O-bound processes. So, it uses relatively high time slices.



(cont'd.)

- Linux adjusts time slices dynamically based on nice value:

Nice	Time Slice
-20	800 msec (`MAX_TIMESLICE`)
0	100 msec (`DEF_TIMESLICE`)
19	5 msec (`MIN_TIMESLICE`)
Child	Parent/2

- Additionally, a process does not need to use a given time slice at once. e.g., 100 msec can be used as 5 x 20 msec. This is good for interactivity.

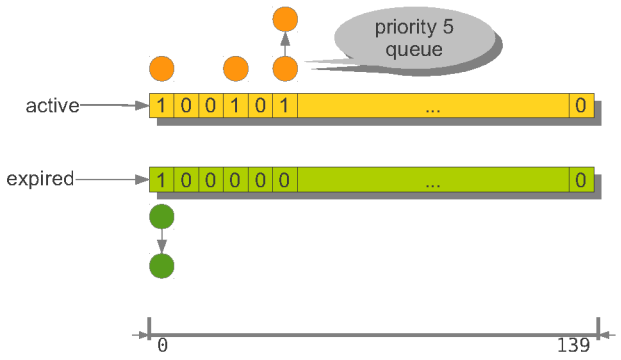
Linux – The *O*(1) Scheduler

- Implemented by Ingo Molnár in 2002
- All algorithms used run in constant time.
- Also, designed for SMP from the ground.
  - Linux virtually supports unlimited number of CPUs.
- The basic data structure for scheduling is called *runqueue*.
  - Each CPU has its own runqueue.
  - Each process is assigned to a single runqueue.
  - So, a process (or thread) runs on a single CPU.
    - Good, we want to use CPU's cache.
  - Linux *heuristically* adjusts workload for each CPU.

(cont'd.)

- Each runqueue consists of two 1-D arrays.
  - Active array
  - Expired array
- The arrays are bitmapped. Each bit corresponds a scheduling priority.
  - So, the size is 140-bit long, or 5 × 32-bit words.
- Both arrays are initialized to 0.
- For active array, a bit is set if there is a *runnable* (i.e., ready) process at corresponding priority.
  - e.g., if there is a runnable process of priority = 10, then bit 10 is set.

(cont'd.)



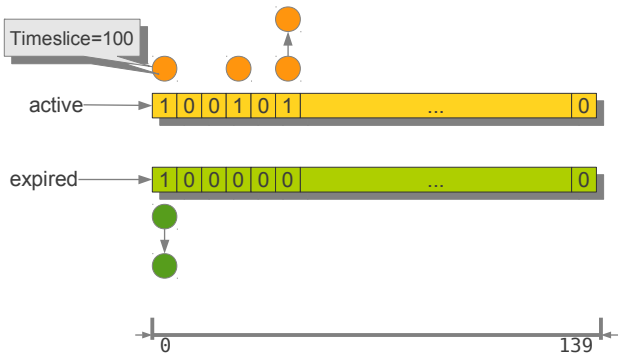
(cont'd.)

- The `schedule()` is called when
  - A process wants to *sleep* (i.e., wait)
  - Preemption
- It finds the first bit set in the active array.
  - Using, e.g., `bsfl` on x86 or `cntlzw` on PPC.
- Then, it select the first process in the queue to run.
  - If the process does not currently hold the CPU, then switch context.
- A process will be scheduled to run until time slice expired.

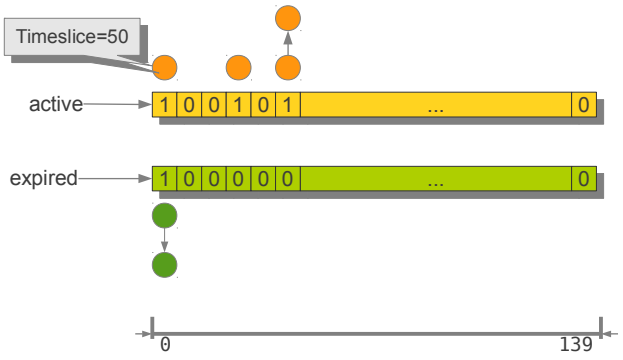
(cont'd.)

- When time slice expired, `schedule()` recomputes priority and time slice for the process, then moves it to the expired array.
  - Priority is adjust in range of -5 to +5, depends on nice and its behavior (e.g., I/O bound or CPU bound).
- Eventually, all the processes will spend their time slice. So, all process will be moved to the expired array, and the active array will be totally reset.
- Then, `schedule()` switches the expired array to the active array, and vice versa.

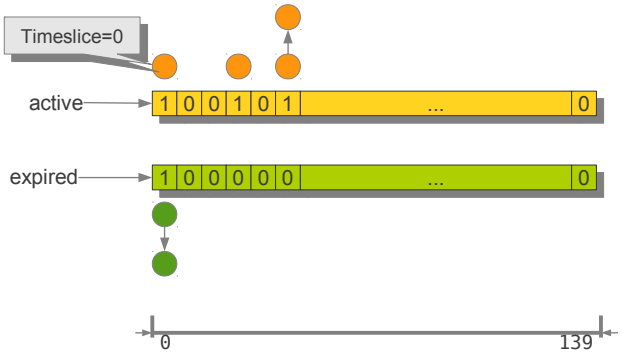
(cont'd.)



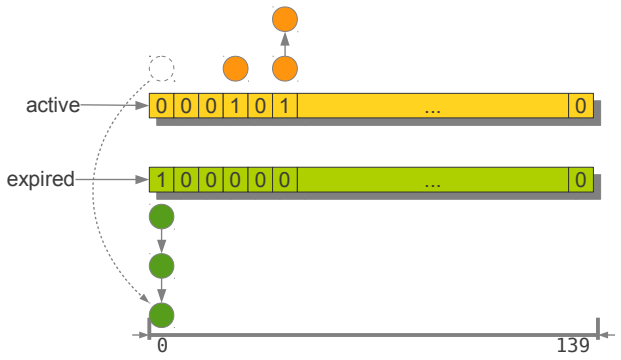
(cont'd.)



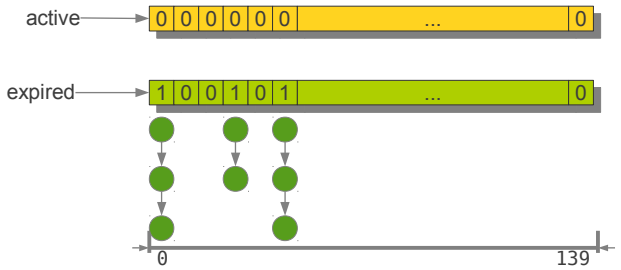
(cont'd.)



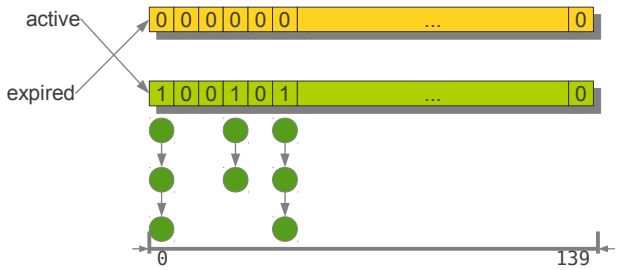
(cont'd.)



(cont'd.)



(cont'd.)



(cont'd.)

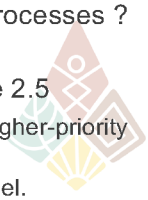
- On SMP, `load_balance()` is called every 200 ms. to balance processor workload.
- Tasks **may** be migrated from one processor to another.
  - This is bad for per-CPU code/data caching.
  - So, they defines a concept of *processor affinity*.
- Still, keeping processor busy might be more important.

Linux – More for Interactivity

- If a process is explicitly interactive, Linux may recompute time slice and reinsert to the active array, given more chance to run.
- An interactive process will be moved to expired array when `expired_starving(rq)` returns true, indicating that expired array is starving.

Linux – Preemption

- Linux supports preemption in user space since very beginning of the kernel development.
- A process may be preempted only at
  - Ending of system call
  - Ending of interrupt handler
- So, basically, higher-priority **user** processes cannot preempt lower-priority **kernel** processes ?
  - Well, not anymore :)
- Linux supports kernel preemption since 2.5
  - Kernel processes can be preempted by higher-priority user processes.
  - This is hard, especially for monolithic kernel.



Linux – Kernel Preemption

- Initially, there was a low-latency patch for kernel 2.2.12 by Ingo Molnár.
  - It is a kind of kernel preemption.
- Then, in 2.4 era, Andrew Morton wrote another low-latency patch for kernel 2.4.x.
  - Basically, this patch uses Ingo's approach.
  - Extremely low latency, very popular among real-time systems, and digital audio workstations.
- During development of kernel 2.5, Robert Love modified the entire kernel to be *preemptible*.
  - This has finally been merged into the 2.6 kernel.
  - Fully preemptible – very good responsiveness.

(cont'd.)

- Later, Ingo introduced the *voluntary kernel preemption*.
  - Allow each kernel process decide whether it should be preempted or not.
  - Still very good interactivity.
- Today's Linux provides options for user to choose preemption model.
  - `CONFIG_PREEMPT_NONE`: good for server
  - `CONFIG_PREEMPT_VOLUNTARY`: good for desktop
  - `CONFIG_PREEMPT`: for low-latency desktop

(cont'd.)

- Linux also allows to tune timer interrupt frequency
  - 100 Hz – good for server
  - 250 Hz
  - 300 Hz – good for digital video editing
  - 1000 Hz – good for desktop
  - No Hz – good for notebook
- Recently, Linux begins to support real-time.
  - Not in the vanilla, but an official-maintained real-time patch for the vanilla.
  - There are also variants of RT-Linux available.
    - Some of them are commercial products.

Linux's Completely Fair Scheduler

- The $O(1)$ scheduler is very nice. Still, it has some deficiencies.
  - A part of the scheduler is interactivity estimator, a kind of heuristics to determine whether a process is interactive.
  - There are certain attacks against the heuristics, e.g., `fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c`.
- Some of users complained about desktop interactivity.
  - Con Kolivas implemented RSDL/SD scheduler in the `-ck` patchset but the scheduler has never been merged into Linux.

(cont'd.)

- Ingo Molnár rewrote the CPU scheduler to maximize CPU utilization as well as interactivity. It has been finally named "*Completely Fair Scheduler (CFS)*".
  - Based on the fair queue idea of RSDL/SD.
  - First patch: Apr 11 2007 08:47 4230 bytes
  - First public release: Apr 13 2007 21:05 101011 bytes
  - By-product: nanosecond granularity, modular scheduler core.
- CFS has been merged into 2.6.23 and is now the default CPU scheduler of Linux.

(cont'd.)

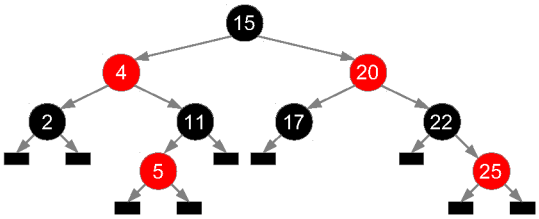
- Basically, it is an implementation of a fair queue.
  - Each of n running (ready) processes gets $1/n$ of CPU time
  - This implies *runtime* fairness.
- A runqueue is associated to each processor.
- The runqueue maintains scheduling entity of running processes.
- Each scheduling entity contains the *virtual runtime* variable that represents amount of time (in nsec.) the process executed.

(cont'd.)

- To maintain fairness, CFS picks the process with the smallest virtual runtime to run.
  - Dequeue from the runqueue
  - Add up execution time
  - If the process is still ready, its scheduling entity is reinserted (enqueue) into the runqueue.
- **Question**: What kind of data structure and algorithm should we use for the runqueue ?

(cont'd.)

- The CFS runqueue is a *red-black tree*.
  - A red-black tree, like AVL tree, is a self-balanced BST.
    - Due to less strict in balance, a r-b tree is faster insertion/deletion but slower retrieval compared to AVL.
  - Imply $O(\log n)$.
  - Key is the virtual runtime of each process



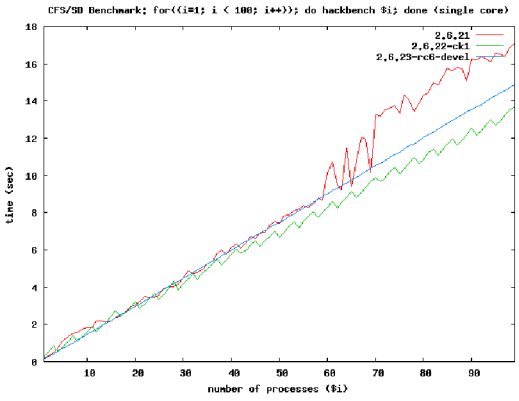
(cont'd.)

- The R-B tree represents timeline of execution.
  - No starvation
- Sleeping processes also get the same amount of CPU time as running processes.
  - Since a sleeper does not spend its time, CFS typically runs it immediately after wake up to maintain fairness.
  - Good for interactivity, no heuristics required.
- The `/proc/sys/kernel/sched_min_granularity_ns` is the tunable parameter.
  - How quickly the scheduler will switch processes in order to maintain fairness.
  - No jiffies, no HZ, no time slices.

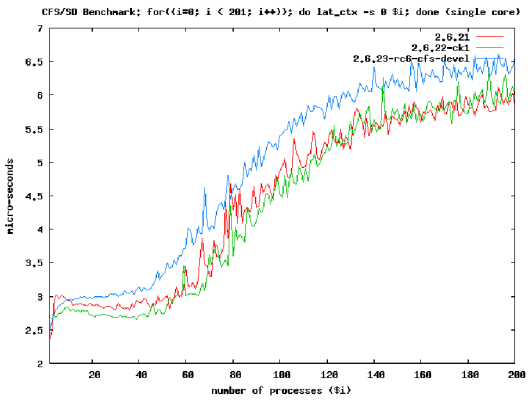
(cont'd.)

- Is CFS better than the $O(1)$ scheduler ?
  - Theoretically, operations on r-b tree is $O(\log n)$, but with 32k-limited of PIDs, CFS is **practically** $O(15)$.
    - Even with the theoretically-limited 1G PIDs of 2.6 kernel, it will be about $O(30)$.
  - The original $O(1)$ scheduler is actually $O(140)$.
- Blind tests suggested that CFS interactivity is as good as SD, and both are definitely better than the former $O(1)$.
- Under CPU intensive tasks, CFS acts slightly better than SD.

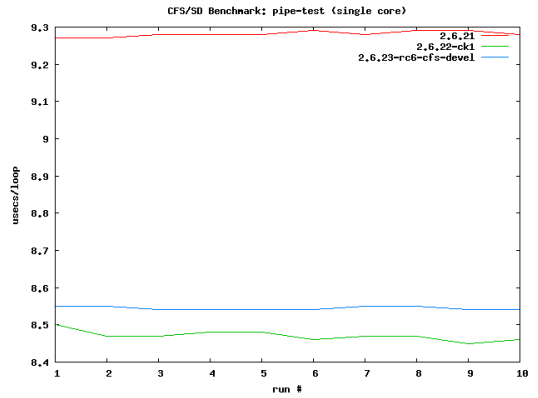
(cont'd.)



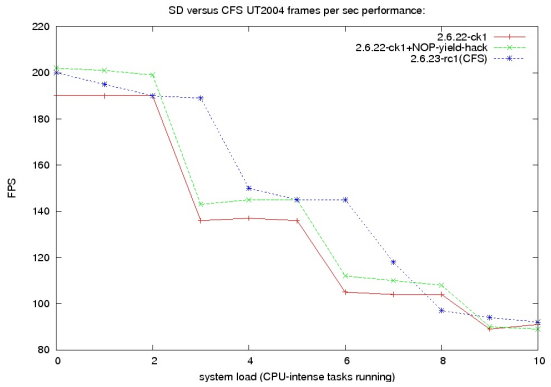
(cont'd.)



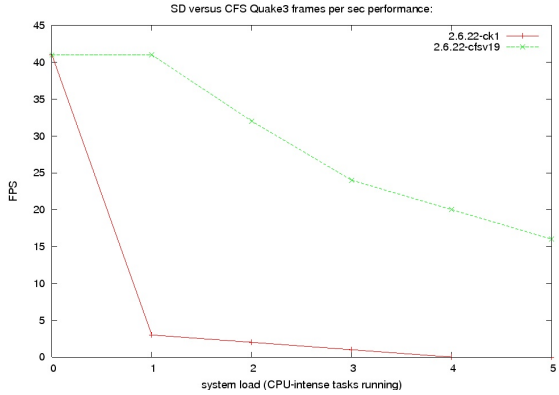
(cont'd.)



(cont'd.)



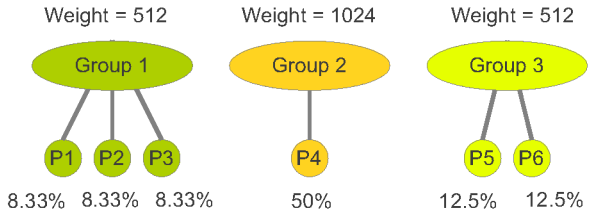
(cont'd.)



Linux – Group Scheduling

- Written by Srivatsa Vaddagiri as an extension of CFS. It has been merged in CFS v17.
- Allow to distribute CPU time among groups of processes
  - e.g., users may get exact share of CPU time to run their processes.
  - Similar to weighted fair queue.
- Weights can be controlled via `/sys`.
- Merged into kernel tree since 2007-07-01, released with 2.6.24.

(cont'd.)



- Group can be UID or cgroup.
  - See `Documentation/scheduler/sched-design-CFS.txt`

Brain Fuck Scheduler

- In August 31 2009, Con Kolivas came back with a simple scheduler to minimizing latency - *BFS: the Brain Fuck Scheduler*.
- It loosely bases on the *Earliest Eligible Virtual Deadline First* (EEVDF) algorithm and the Staircase Deadline.
  - Conceptually, EEVDF is very similar to CFS but provide (virtual) deadline fairness instead of (virtual) runtime fairness.
- Some distro. use BFS as the default, e.g.,
  - Zenwalk 6.4, PCLinuxOS 2010
  - CyanogenMod

(cont'd.)

- But, BFS comes with prices:
  - Sacrifice throughput for latency, lead to larger turnaround time.
  - Not scalable
    - Need to globally lock the global runqueue across processors
    - CK suggests BFS is for systems with processors < 16.
    - So, this will never be merged into the mainline kernel.

Automatic task group creation

- In 2010-10-19, Mike Galbraith wrote a small patch to improve desktop responsiveness.
  - Based on the discussions and Linus's suggestion about automatically create task groups per `tty`.
  - First patch: 8 files changed, 186 insertions, 1 deletion
- In 2010-11-15, the version 3 of this patch released. It's been reviewed, tested (by phoronix), and finally slashdotted.
  - 9 files changed, 224 insertions, 9 deletions
  - Interactivity performance is comparable to BFS, but with very small performance/scalability penalties.
  - Merged to 2.6.38.