Introduction to Operating Systems

Processor Scheduling

Contents

List of Figures

1 Acknowledgments

As with any Computer Science course, Introduction to Operating Systems is constantly evolving. This evolution does not happen in a vacuum. In addition to many outside resources being consulted, we have been very fortunate to have important contributions from our TAs. Ted Cooper and Jeremiah Peschka have been particularly valuable in improving these lecture notes, as editors and contributors. Ted and Matt Spohrer developed the quiz infrastructure and questions. The quiz questions form the basis of the study questions now included with each set of notes. Ted and Jeremiah have made significant contributions and improvements to the course projects which use the xv6 teaching operating system.

As always, any omissions and/or errors are mine, notifications of such are appreciated.

2 Learning Objectives

1. Explain the high-level goals for a scheduling policy in a multiprogramming system.
2. Explain the differences between "long running" and "interactive" processes.
3. Explain the different environments for which specific scheduling policies are optimal.
4. Explain why limited direct execution is necessary.
5. Explain in some depth, these scheduling algorithms
   - FIFO
   - SJF
   - SJN / SPN
   - STFC / SRT
   - RR
   - Priority
   - MLFQ
6. Explain, at a high level, these scheduling algorithms
   - Basic UNIX
   - Lottery
   - $\mathcal{O}(1)$
   - Completely Fair
7. Discuss basic issues with differences in scheduling algorithms with respect to constraints upon the computing environment. This is a complement to question 3.
8. Use basic queuing theory to gain insight into the behavior of systems.

3 Terms and Definitions

Concept	**Description**
Task	A unit of execution.
Job	Comprised of one or more tasks. Synonym for *process*.
Turnaround Time	The time a job takes to complete (exit the system) after it has initially entered the ready state.
Response Time	The time a task takes to be scheduled on a processor once it has entered the ready state.
Time-Slice	The maximum amount of time that a task may take on a processor.
Quantum	The length of a time-slice.
Schedule	The order of execution for a set of tasks. The sets of tasks is also called the *workload*. Note that the schedule may change dynamically.
Fairness	The concept that tasks receive approximately an equal share of resources over time. The time-slice is one tool that can be used to achieve fairness. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. If you are willing to be unfair, you can run, for example, shorter jobs to completion[1], but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time.
Throughput	The number of tasks completed per unit time.
Overhead	The latency for performing a specific action. The scheduler algorithm, which is run at least once per time-slice, is overhead that is applied to all tasks so is considered fair, even though it add to the completion time for any one task.
Predictability	Consistency over time. Usually applied to a metric such as response time. If the response time of a system is consistent over time, we say that the *response time is predictable*. Wide variations in predictability can make a system more difficult for users.
Processor-bound	A task or job is processor-bound if it does very little I/O.
I/O-bound, Interactive	A task or job is interactive if it is dominated by I/O requests. Note that an I/O-bound task may run for a very long time. Being I/O-bound does not imply that the job is short-lived.
Timer Interrupt	
Voluntary Yield	
Involuntary Yield	

[1]This is often called a *bias* where one category is given priority over another. We will see examples of priority-based algorithms that strive to achieve fairness, although they tend to be quite complex.

Note that most processes go through phases. At different times, a process, and thus some subset of its tasks, may be I/O-bound – getting user input or performing file or network I/O. At other times, the process may be processor-bound, performing calculations with very little I/O. The real world is often a messy combination of these two extremes.

4 Processor Virtualization

In order to simplify processor management, the operating system provides each process with its own *virtual processor*. This virtual processor is more limited than the real processor but comes with benefits that help make multiprogramming much easier to accomplish.

In order to virtualize a processor, the operating system needs to somehow share the physical processor among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By time-sharing the processor in this manner, virtualization is achieved. When the virtual processor for a specific process is mapped into the real processor, the process is directly executing on the processor without any intervention by the operating system. This is called *direct execution*.

Direct execution has the obvious advantage of being fast; the program runs natively on the hardware processor and thus executes as quickly as one would expect. But running on the processor introduces a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as processor or memory? In other words, how do we achieve *limited direct execution*?

The approach we take is to introduce processor *modes*: known as *kernel mode* and *user mode*. Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests directly to the device; doing so would result in the processor raising an exception which would likely cause the operating system to terminate the process. Instead, the process must request that the operating system perform the I/O operation on its behalf.

In contrast to user mode is *kernel mode*, in which the operating system (or kernel) runs. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

The process virtual processor can only execute in user-mode. When it attempts to execute kernel-mode instructions, the processor is programmed to yield control to the operating system in a very specific way. The simplest way for a virtual processor to gain access to services only available in kernel-mode is via a system call.

4.1 System Calls

We are still left with a challenge: *What should a user process do when it wishes to perform some kind of privileged operation?* To enable this, nearly all modern processors provide the ability for user programs to perform *system calls*.

System calls allow the kernel to carefully expose certain privileged functionality to user processes, such as accessing the file system or allocating additional memory.

To execute a system call, a program must execute a special instruction, called a “trap”. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work on behalf of the calling process. When finished, the

OS calls a special return-from-trap instruction which returns into the calling user program while simultaneously reducing the privilege level back to user mode.

4.2 The Context Switch

Modern processors support a wide variety of *interrupts*. The one that we are most concerned with here is the *timer interrupt*. At boot time, the operating system initializes a timer, often via specialized hardware called a *programmable interrupt controller* (PIC). The PIC is usually a countdown timer that posts a timer interrupt to the processor when the count reaches zero and then immediately restarts the countdown without waiting for the processor to take action on the posted interrupt. This timer guarantees that the operating system will periodically regain control of the processor.

It is common for the operating system to define a time quantum called a *time-slice* that is essentially a period of time that equates to an integer multiple of the time between timer interrupts. When a time-slice expires, the scheduler will remove the currently executing process from the processor and allocate it to a (potentially) different process.

The combination of virtual processors and interrupts now give us *limited direct execution*. This approach also gives the application programmer a *simpler programming model* where the complexities of the processor, other hardware, and protected operations are managed by the operating system; to the benefit of all processes.

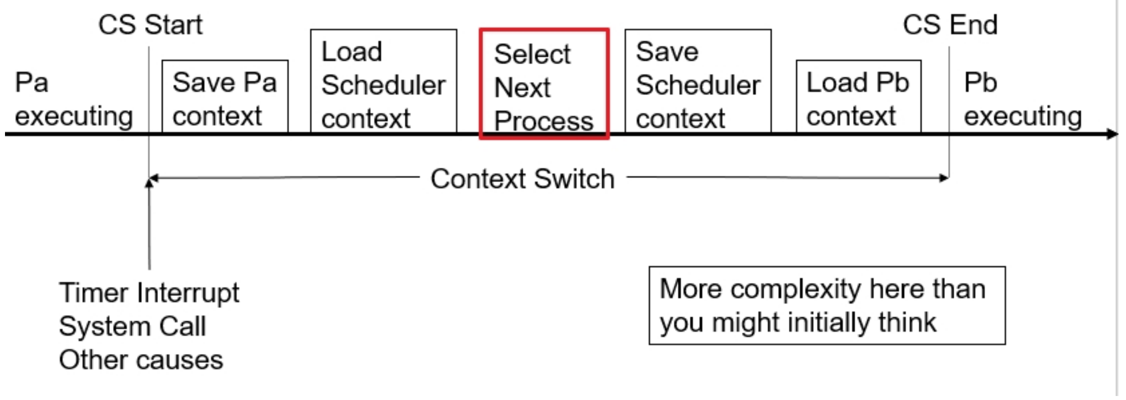

Figure 1: Steps of a Context Switch

A context switch can also be executed as a side-effect of a system call. The timer interrupt is an example of an *involuntary context switch* whereas a system call is an example of a *voluntary context switch*. Whenever a process requests a service from the operating system, the process "knows" that a context switch could occur (this is why it is called voluntary).

5 Uniprocessor Scheduling

5.1 First In, First Out (FIFO)

Also called **first come, first serve (FCFS)**. This algorithm is *optimal* for average turnaround time in an environment where all jobs arrive before the first job is run, all jobs have the same run time, and there is no preemption.

Since all the jobs are available to the scheduling algorithm before we run even the first job, the scheduler can establish a total ordering on the set of jobs a priori. The schedule can be set outside of the process of running specific jobs.

Job selection is very simple: any job will do. All jobs are equal, so even random selection is optimal. This is optimal for the average turnaround case; obviously individual jobs will have differing individual turnaround times. In addition to being optimal, FIFO is very easy to implement with very low overhead. Indeed, no computation required for scheduling.

5.2 SJF and SJN

For better turnaround time when jobs are of different lengths, we would choose the shortest job as the one to run next. It is provable that, in the absence of preemption and when all jobs arrive before any run, shortest job first is optimal. If we allow jobs to arrive late, then we select the shortest job each time another job ends; this is called shortest job next.

With SJF, we have an optimal schedule, but long running jobs are penalized to run last and thus have the worst turnaround time of all jobs in the schedule. However, things get much worse with SJN.

If there is always more than one process ready at each scheduling interval, then it is possible that there is a process that never gets to run. This is called *starvation.* Starvation is to be avoided. At the very least, it is unfair to the jobs that are subject to this unbounded delay.

5.3 STCF and SRT

For the rest of this lecture note, we will assume that preemption is allowed. That is that there is a periodic timer interrupt and that the system implements a time-slice, which upon expiration will cause an involuntary context switch for the processor.

If we know how much time a job will take and also track how much processor time the job has used so far, we can then choose to allocate the processor to the job that requires the least amount of processor time to complete. This is called *shortest-time-to-completion* or *shortest remaining time.* This approach is just SJN with interrupts, so we have not addressed the starvation issue. Sigh.

5.4 Round Robin

For SJF, SJN, STCF, and SRT, we must sort the ready queue every time we add a new task to the list. This is a lot of overhead! And we can have starvation!

We would like to have the benefits of FIFO without the starvation issue. Enter *round robin.*

Round robin is a scheduler that has FIFO semantics but with a twist. Round robin uses preemption. Recall that a time-slice is the maximum amount of time that a job may be on the processor before a scheduling event takes place (context switch). Round robin will always select the task that is at the front of the ready list when scheduling the processor. When a time-slice expires, the task is removed from the processor and placed at the *back* of the ready list. In this way, we can see that no starvation can occur and that all tasks will eventually be scheduled, which will allow all jobs, absent bugs, to eventually complete.

What is the impact of RR on turnaround time and response time?

Response time is the time a task spends in the ready list waiting to run. With RR, we have an absolute bound on this time. The maximum time that any one task has on the processor is the time-slice, denoted by T. The number of tasks ahead of any task in the ready list is denoted by N. Thus, the worst case delay for any task is $delay = N * T$. Note that if a task is interactive, it will give up the processor before the end of the time-slice, so, based on the mix of interactive and long-running jobs, the delay could be much less. If all the tasks ahead of us are long-running, then we get worst case response time.

We'd like to have best case response time for our interactive jobs. Is there a way to get there with round robin? First, let's look at an example.

Let's assume that an interactive job uses only 10% of its time-slice. A long running jobs will use all of its time-slice except, perhaps, for its last time-slice.

If all the tasks ahead of us in the ready list are interactive, the delay becomes $delay = N * (T * 0.10)$. If all the tasks are long-running, the we use our original formula. This simple example shows that the delay could vary by a factor of ten! eek!

In general, long-running jobs have very little in the way of user interaction on the jobs have all their in initial inputs. Interactive jobs, however, are often characterized by interacting directly with a user, via keyboard, mouse, microphone, etc. Having a high variance in response time can be frustrating for users of interactive jobs.

Is there a way to get the benefits of round robin with a priority, also called a *bias*, for interactive jobs? Let's see.

5.5 Priority

A simple approach is to give each task a priority, based on some categorization. Let's use a simple priority wherein an interactive task receives a priority over a long-running task. This approach gives a bias towards interactive jobs and thus gives us a response time closer to $delay = N * (T * 0.10)$, the optimal round robin response time for our example, above.

But ... there is a problem. A priority scheme is just another form of shortest job next. In SJN, the priority was time to completion. So, while this approach may give us the correct bias, it also opens up the door to starvation.

Is there a way to get the "no starvation" quality of round robin and the bias of priority? The answer is yes, but why?

The true problem with priority scheduling is that the priorities are fixed, not dynamic. Rather than give a priority to a class of tasks, why not allow there to be multiple priorities and if a task appears to be long-running, we lower its priority. One way to do this is to take a look at how much of a time-slice a task uses. If it uses the entire time-slice, then demote it. Simple! Oh wait, we still have starvation. Can you see it? How can we address the starvation issue inherent in priority schemes?

The principle problem with priority scheduling is not that priorities are somehow bad, but rather that priorities can only be reduced. This means that, as long as there is a higher priority task ready to run, a lower priority task is subject to starvation. We could fix this by periodically increasing task priority, but how to do so with both fairness and a bias towards interactive jobs is tricky.

5.6 Multi-Level Feedback Queue (MLFQ)

In an ideal world, our scheduler would be able to learn about all the jobs and be able to favor interactive ones, for good response time, but not starve processor bound jobs. MLFQ to the

rescue!
Some observations

- Long running jobs don't care about response time
- Interactive jobs require good response time
- Implies a priority
- How would be implement priorities in RR?
  - Remember that we want to keep overhead low
  - Starvation must be avoided
- Design
  - Multiple queues based on priority
  - Serviced in high-to-low order
  - Each queue is RR

We now have the concept of priority. We want to prioritize interactive jobs without starving long-running jobs. Any algorithm that tackles this problem will need to be *adaptive.*

Initial Rules (flawed)

- 1: If Priority(A) > Priority(B), A runs (B doesn't).
- 2: If Priority(A) = Priority(B), A & B run in RR.
- 3: When a job enters the system, it is placed at the highest priority
- 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
- 4b: If a job gives up the processor before the time slice is up, it stays at the same priority level.

The issue

- Rules 4a and 4b are subject to gaming
- Remove dependence on time slice with a budget
- **New Rule 4:** When budget is used up, demote and provide a new budget
- Removes gaming based on time slice
- More fair and equitable

Our old friend starvation

- Lowering a priority means that jobs at higher priorities are given preference

  - ▷ What if there is always a higher priority job?

- In a busy system with many short-lived processes, a demoted process may be shut out from using a processor
  - ▷ Process starvation now possible
- How do we fix this problem?

Promotion to the rescue!

- Rule 5: After some time period S, promote all jobs
- Two strategies
  - ▷ Promote all to highest priority
  - ▷ Promote each job one level
- Which is simplest?
- Which better meets the goal of giving interactive jobs priority while not starving long running jobs?

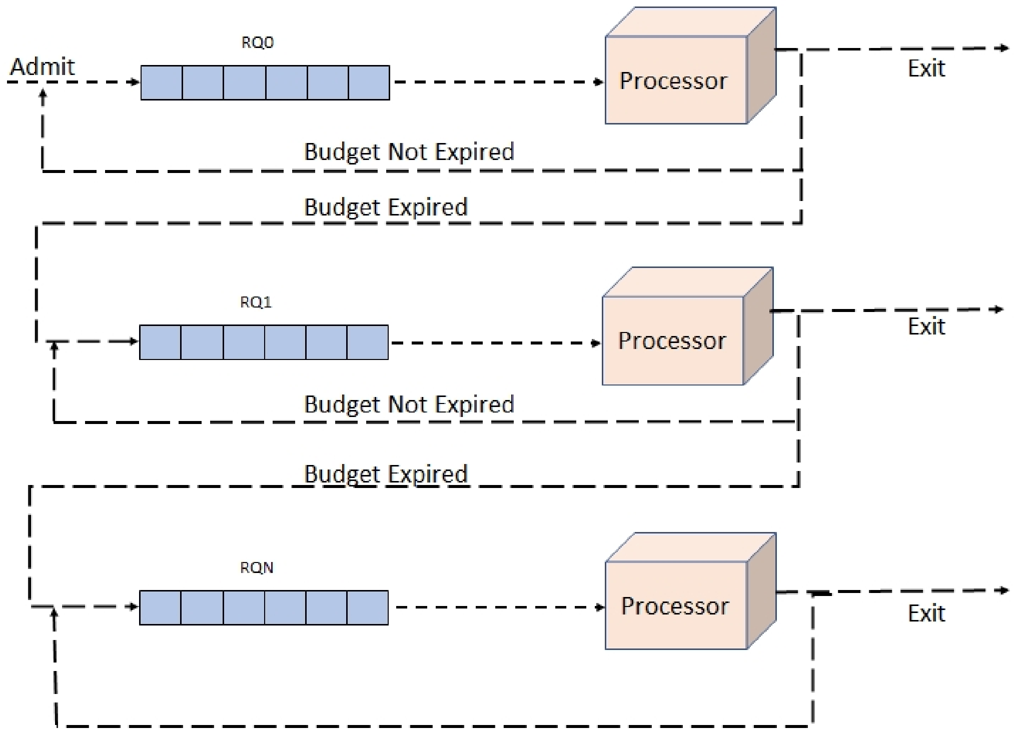

Figure 2: MLFQ Scheduler with Budget

When a process is removed from the processor, but hasn't called `exit()`, the process may be returned to the same queue or a lower priority queue, based on feedback via the *budget*. When a process moves to a lower queue, the priority is lowered by one level. This process is called *demotion*.

At some point, a promotion timer occurs that causes all processes to be moved one queue higher in priority. This process is called *promotion*[2].

Demotion at the lowest level and promotion at the highest level are ignored in this high-level description. However these *edge cases* cannot be ignored in the implementation.

We do not care how long a job will be in our system. We also do not care if the job is interactive, long running, or both. In fact, under MLFQ we do not distinguish between them and instead use the rate at which the time budget is used up to decide where in the hierarchy of priority queues the job will reside. However, our scheduling algorithms are more complex. They take more time. This overhead is taken from the start of every time slice and thus the user process.

Question: Is the result worth the overhead?

A Note on Complexity

- FIFO and MLFQ are both $\mathcal{O}(1)$
- MLFQ better tracks jobs that change back and forth between interactive and long running
  - ▷ Better represents the real world
  - ▷ Why?
- Promotion strategies – reset v. one-level
  - ▷ Would there be a noticeable difference between these two mechanisms in practice?
  - ▷ Why or why not?

MLFQ Final Rule Set

1. If Priority(A) > Priority(B), A runs (B doesn't).
2. If Priority(A) = Priority(B), A & B run in RR.
3. When a job enters the system, it is placed at the highest priority (the topmost queue).
4. Once a job uses up its time allotment (budget) at a given level (regardless of how many times it has given up the processor), its priority is reduced (i.e., it moves down one queue).
5. After some time period S, promote all jobs one level

6 Multi-(whatever) Scheduling

In this section, we will discuss scheduling approaches for multicore and multiprocessor architectures. First however, we need to know a bit more about caches.

[2]The priority could also be set to the highest priority. In this approach, the highest priority queue will periodically contain all processes from all ready queues. This approach is termed *priority reset*.

6.1 Cache Considerations

There is an inherent trade-off between size and speed. A larger resource implies greater physical distances[3] but also a trade-off between expensive, premium technologies (such as SRAM) vs cheaper, easily mass-produced commodities (such as DRAM or hard disks).

For uniprocessor systems, a cache is viewed as just a faster memory. Each level of cache is substantially faster, and smaller, than the level below. This is a boon.

For multiprocessor systems, the cache is both a boon and a bane. For uniprocessor systems, your data is in the cache or not. With a multiprocessor system, your data could be in *some cache*, just not in the cache for the processor on which you are scheduled. So while your data is cached, it could be cached in a place that does you no good.

Figure 3 illustrates the issue. We can see that each processor has one or more private caches, a shared cache, and then access to a shared main memory. The issue for our process is the private cache(s). What if we could remember which processor ran our process last? Could this be useful? The answer is, of course, yes. Modern caches are quite large and there is a good chance that data for a specific process will still be in the cache if the process is again assigned to the same processor later. As a result, we can get improved performance by accounting for *processor affinity*, also called *cache affinity*. Allocating the same processor to a task based on affinity is known as *pinning* that task to that processor.

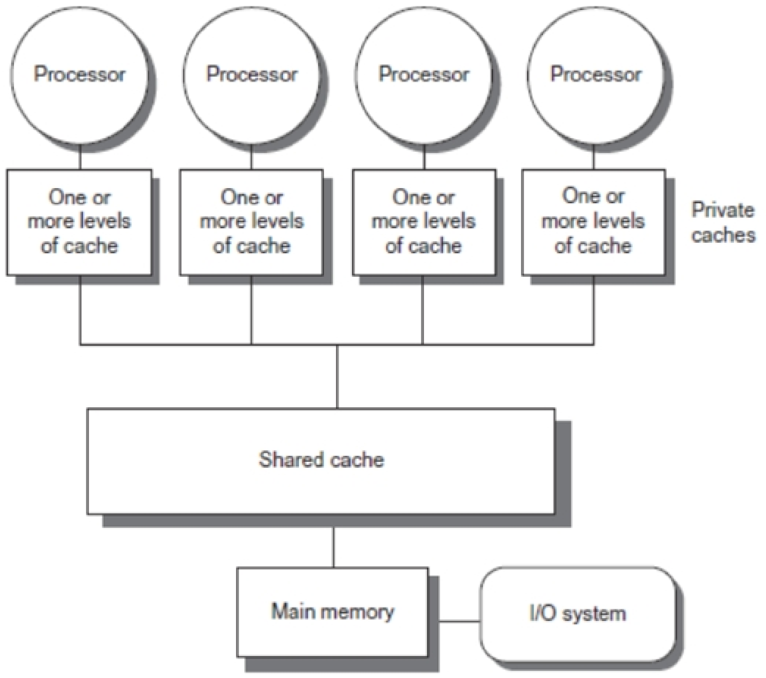
Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 3: SMP Layout

Inclusion policies Whether a block present in the upper cache layer can also be present in the lower cache level is governed by the memory system's inclusion policy, which may be inclusive, exclusive or non-inclusive non-exclusive (NINE)[4].

[3]Nothing is instantaneous, not even light – the jury is still out on gravity. This is an important fact to know.

[4]See the L2 caches in Figure 4.

With an inclusive policy, all the blocks present in the upper-level cache have to be present in the lower-level cache as well. Each upper-level cache component is a subset of the lower-level cache component. In this case, since there is a duplication of blocks, there is some wastage of memory. However, checking is faster.

Under an exclusive policy, all the cache hierarchy components are completely exclusive, so that any element in the upper-level cache will not be present in any of the lower cache components. This enables complete usage of the cache memory. However, there is a variable and sometimes high memory-access latency.

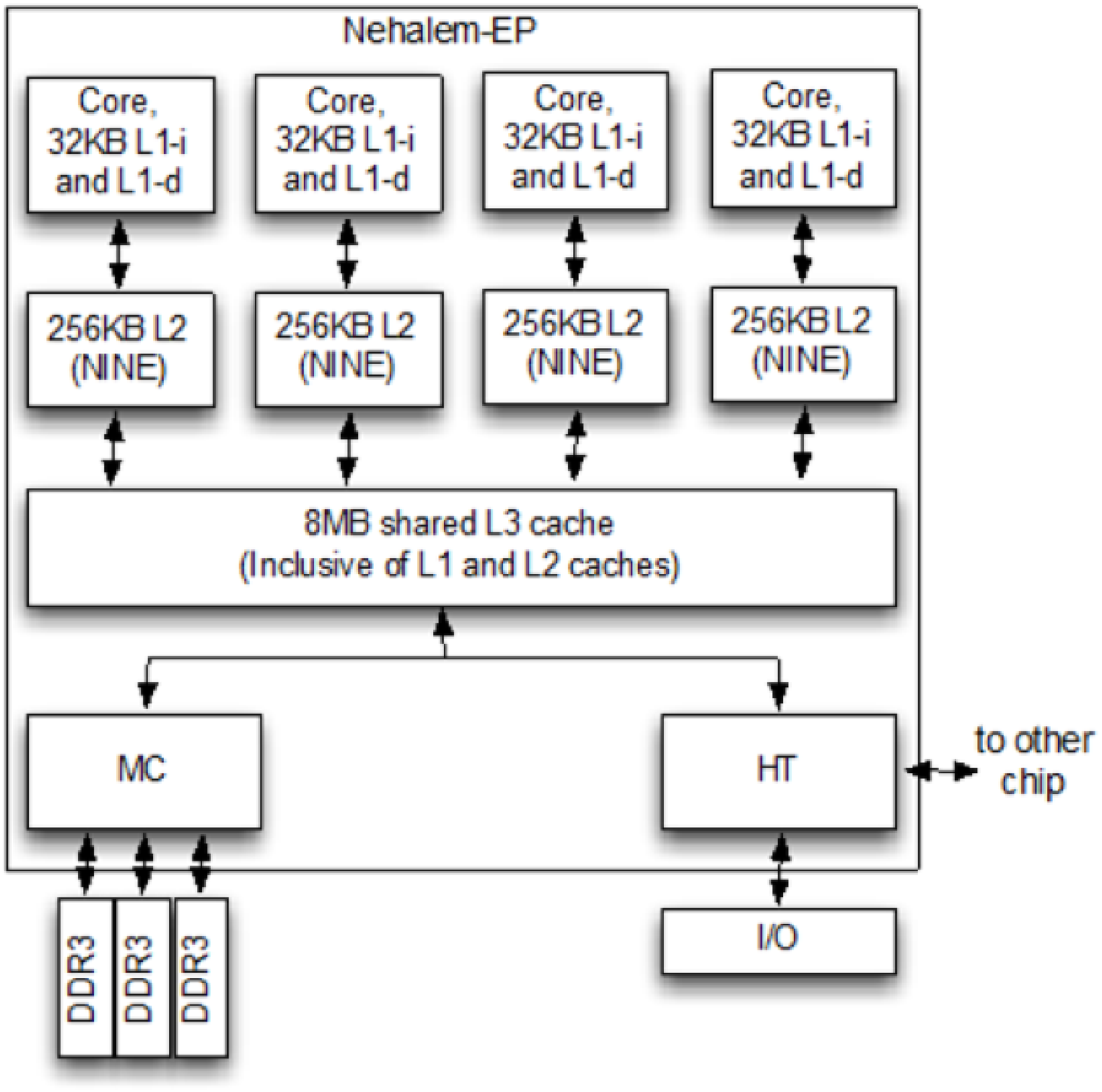

Figure 4: Intel Nehalem Cache Layout

Writing to the cache There are two policies[5] which define the way in which a modified cache block will be updated in the main memory: write-through and write-back. We will discuss this with regards to an inclusive cache design.

In the case of write-through policy, whenever the value of the cache block changes, it is further modified in the lower-level memory hierarchy as well. This policy ensures that the data is stored safely as it is written throughout the hierarchy.

However, in the case of the write-back policy, the changed cache block will be updated in the lower-level hierarchy only when the cache block is evicted. A "dirty bit" is attached to each cache block and set whenever the cache block is modified. During eviction, blocks with a set dirty bit will be written to the lower-level hierarchy. Under this policy, there is a risk for data-loss as the most recently changed copy of a datum is only stored in the cache and therefore some corrective techniques must be observed.

In case of a write where the byte is not present in the cache block, the byte may be brought to the cache as determined by a write allocate or write no-allocate policy. Write allocate policy states that in case of a write miss, the block is fetched from the main memory and placed in the cache before writing. In the write no-allocate policy, if the block is missed in the cache it will write in the lower-level memory hierarchy without fetching the block into the cache.

6.2 Multiprocessor Scheduling

We will tackle the easier case of multiprocessor scheduling first. Figure 5 shows an example system architecture with multiple, discrete, processors. These processors are in their own package and communicate via the memory bus. Each processor now has a private cache hierarchy and communications between processor caches occurs at main memory speeds.

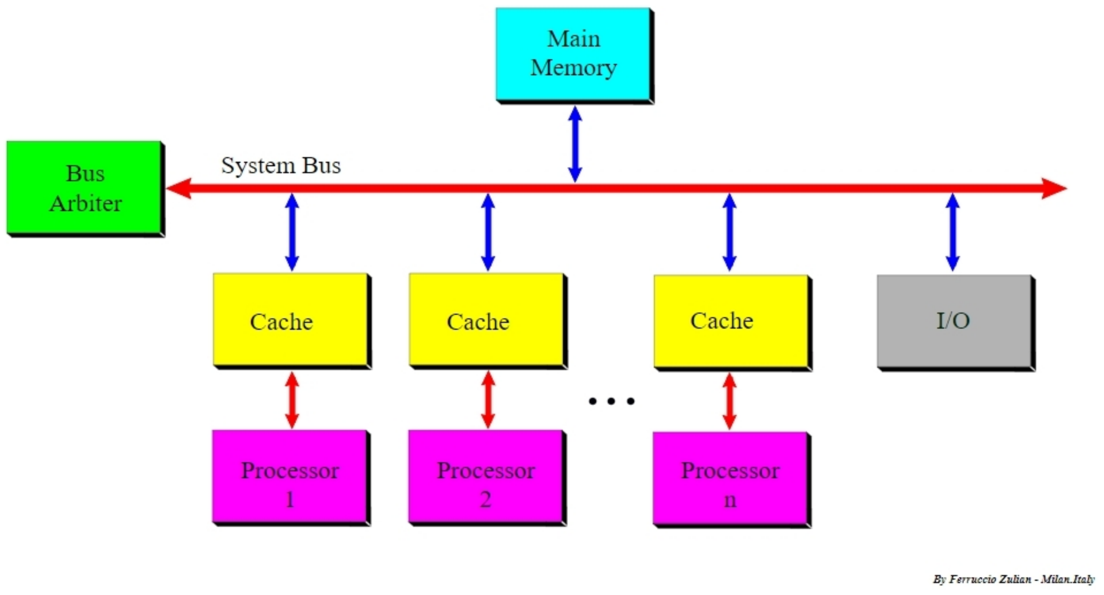

Figure 5: Multiprocessor Layout

For some processor X and some process P_A, the only cache likely to have any cached data for

[5]See Wikipedia.

P_A is processor X. Any other cache is unlikely to have cached data for P_A so scheduling P_A on some other processor than X would result in a *cold cache* and all data and instruction accesses would occur at main memory speeds until the process working set was cached.

This argues for a scheduling policy that is based on *processor affinity.* To use this approach, we would need some way to track which processor P_A was on when it was context-switched out. There are two approaches that we will discuss: a unified ready list and a per-processor ready list.

Unified Ready List One approach is to add a field to the process control block that indicates the processor on which the job last executed. Most processors have an instruction to obtain the unique processor identifier. Then, when a processor wants to schedule a task, it locks the ready list and removes the next task that is marked (has affinity) for it. Making this efficient is tricky. Processor affinity is a form of priority scheduling, and if we aren't careful, performance could suffer.

However, the main performance issue is *lock contention.* Locking the entire ready list, not just a portion, does not scale well. The lock is a point of *serialization* and the scheduler is run *often.* The more processors involved, the more likely that contention will be an issue. Careful choice of the data structure for the ready list could help here; e.g., a hash table where each bucket corresponds to an individual processor and each bucket has its own lock, so that the lock on the entire hash table can be released once a specific bucket is locked.

Another issue is load balancing. For example, in the case of a two-processor system, it could occur that all the jobs that have affinity for processor 1 complete while no jobs for processor 2 complete. Without intervention, our multiprocessor system will devolve to single processor performance. How can we *steal* tasks from one processor and allocate them to another processor? Even with a cold cache the performance will be better than that of a single processor. Of course, one hard question here is: *how do we know when to steal processes from one processor and reallocate them to other processors?* This implies a search and that means even more lock contention. Note that the hash table approach described above can help here: we only need an approximation of the bucket size, so counting the tasks in each bucket without first acquiring the lock may be acceptable.

Per-Processor Ready list If we have multiple processors, each with one core, then allocating a ready list per processor and using processor affinity in scheduling is a good approach. A hash table approach could be used wherein each hash bin corresponds to an individual processor. Alternately, a separate ready list, such as a linked list, could use used. Rebalancing the ready lists can be made easier if both a head and tail pointer were used.

6.3 Multicore Scheduling

With modern processors, it is much more likely that a system will have multiple cores and perhaps even multiple processors. As can be seen in Figure 6, such a system would have affinity at the level of the core-private cache(s), shared cache(s) within the processor, and perhaps (not shown) a shared cache between processors (sometimes called a "level 4" cache).

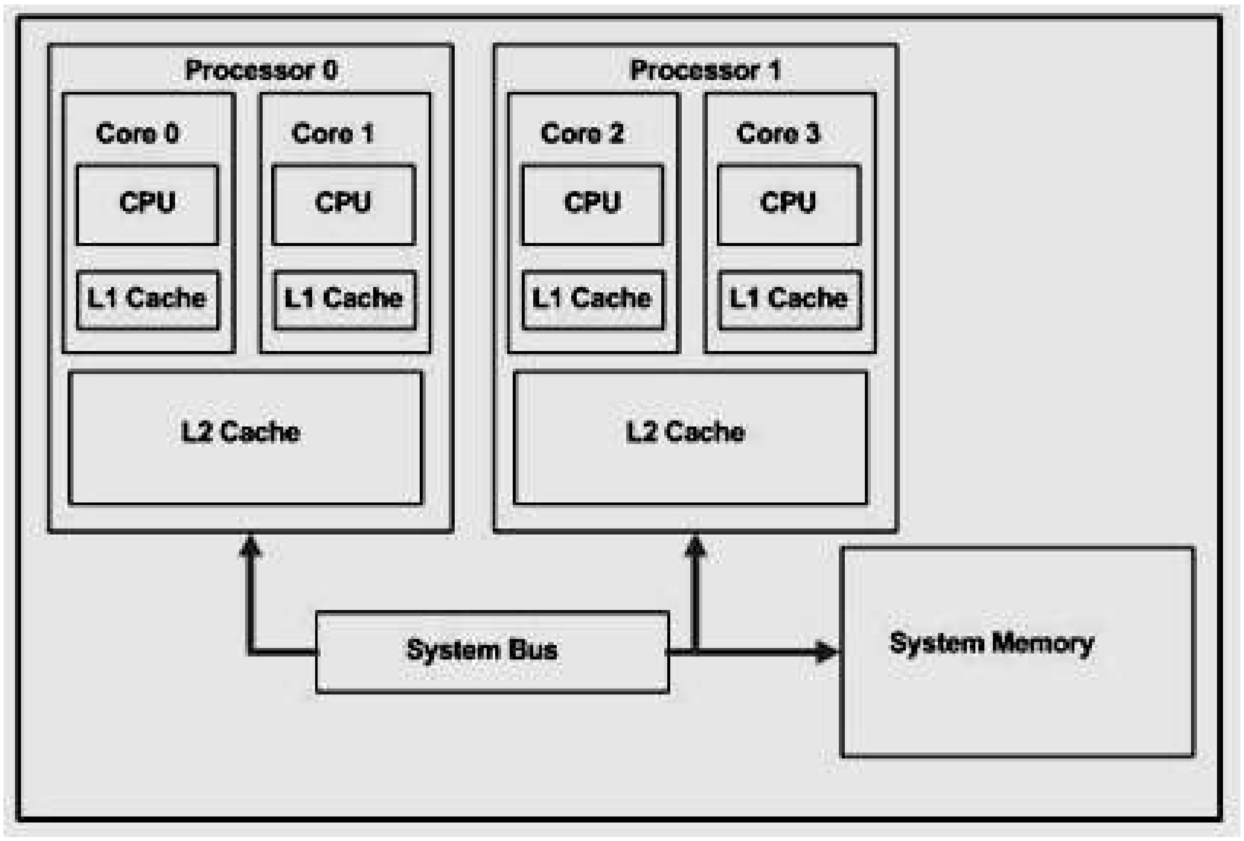

Figure 6: Multiprocessor with Multiple Cores

An architecture such as this gives rise to several possible approaches, of which we will describe but one.

Suppose that the system uses a per-processor ready list, implemented in an efficient manner, perhaps using a hash table approach already outlined. The choice of ready list at this level would use *processor affinity* as previously discussed. Each processor's ready list would be shared by more than one core. At this level, one type of rebalancing could occur when a task is being assigned to a processor. If another processor has an empty list, the task could be reassigned to that processor, foregoing any benefit of a warm cache but perhaps providing better overall performance.

Now, let's assume that each hash bucket can be further divided into a list for each core. For tasks arriving in the hash bucket that have no recent affinity (now tasks or rebalanced tasks), either core will do. For arriving tasks that would show a *core affinity*, the task can be placed on the appropriate list within the bucket. Rebalancing among the cores of an individual processor would only require a lock on the hash bucket.

There are other approaches, but this gives you a flavor for the options.

There are other, more advanced approaches that take into account interactions between threads of a process and other issues that can lead to even better performance. One such approach is called *gang scheduling*, which will not be discussed here.

7 Real-Time Scheduling

A *real-time constraint* is a computation must be completed by a deadline to have value. There are two types of real-time constrains: hard and soft.

A hard real-time constraint is a computation that *must* be completed by a strict deadline. Examples of such systems are airline traffic control, video conferencing, and certain medical devices.

A *soft real-time constraint* is a computation that, if the deadline is missed, perhaps only degrades system operation. Examples of such systems are media streaming services and computer games.

7.1 Over-provisioning

One approach is to ensure that the tasks in the system only use a fraction of the available resources. This helps to ensure that real-time tasks can be scheduled quickly, without having to wait for other tasks in the system.

7.2 Earliest deadline first

Another approach is a form of priority. Tasks are scheduled using an *earliest deadline first* approach. The ready list would be maintained in deadline order. Admission control for the system would not admit any task that would cause an already admitted task to miss its deadline.

A better, but more complex, approach might be to allow a new task to run when a task blocks on some event, such as a disk read. This would make more efficient use of the processor and could allow the task schedule to be completed earlier than it otherwise would. However, now we have the issue of what to do once the event completes. A simple approach would be to preempt the task currently on the processor and reinstate the task with the higher priority. Careful measurement would be needed to see if the overhead was worth the extra processor efficiency. Note that even with this new approach, admission control is necessary.

7.3 Priority Inversion

The interaction of shared data structures, priorities, and deadlines can lead to subtle problems. One well-known problem is known as *priority inversion.* This occurs when a higher priority task is prevented from making progress because a lower priority task holds a shared resource. It does not matter if the higher priority task is scheduled immediately, it is blocked from making progress by being serialized on the shared resource. We have a situation where the lower priority task starves the higher priority task. The lower priority task cannot run because there is a higher priority task available and the higher priority task cannot make progress because a needed resource is held by the lower priority task.

One possible solution is called *priority donation* where any task holding a shared resource runs with the highest priority among the set of tasks waiting on that resource. In a sense, the higher priority task has loaned the lower priority task its priority so that the lower priority task can run and release the resource. Once the resource is released, the task assumes its original priority.

8 Some Queuing Theory

Eventually, the designer of the scheduler algorithm must take into account the *rate* at which new jobs / tasks arrive in the runnable state. Failure to take this into account could be catastrophic for system performance. For example, a system that expects new jobs to be of the same type and to arrive at a rate less than the service time can use a simplified approach to scheduling. Systems that must support a variety of job types and arrival profiles must be more complex.

In this section, we take a simplified look at queuing theory, which can be quite useful in terms of understanding job loads, even if they vary over time.

Term	**Description**
Queuing delay (W)	Wait time. The total time that a task must wait to be scheduled.
Tasks queued (Q)	The total number of tasks in the ready list.
Service Time (S)	Execution time. The time to complete a task, assuming no waiting.
Response time (R)	Service time plus wait time. $R = W + S$.
Arrival rate (λ)	Average rate at which new tasks arrive in the system. However, the pattern of arrival – bursty or evenly spread out evenly – is also important.
Service rate (μ)	Tasks that can be completed per unit of time. Note that $\mu = 1/S$.
Utilization (U)	The fraction of time that the system is busy ($0 \leq U \leq 1$). $$U = \begin{cases} \frac{\lambda}{\mu}, & \lambda < \mu \\ 1, & \lambda \geq \mu \end{cases}$$
Throughput (X)	The number of tasks processed per unit time. $$X = U\mu$$ Alternately, $$X = \begin{cases} \lambda, & U < 1 \\ \mu, & U = 1 \end{cases}$$
Task count (N)	The average number of tasks in the system. The number of queued tasks plus the number of tasks currently being serviced[6].

8.1 Little's Law

Little's Law applies to any *stable system*. It denotes a general relationship between average throughput, response time, and the number of tasks in the system. Note that this equation applies to any system *provided it is stable.*

$$N = XR$$

This result is simple and intuitive. That it has been proven for *any stable system* is powerful.

[6]The equation in the OSPP text is incorrect. We do not know the number of processing units.

8.2 Response Time vs. Utilization

In an ideal world, our system would have very fast response time at 100% utilization. However, in general, the higher the utilization of a system, the lower the response time. The system designer must find the right balance between *enough resources* and *acceptable response time.* This is a classic *time vs. money* trade-off. The key here is the arrival rate, λ.

Best case: Evenly spaced arrivals. If we have a set of tasks that are all the same size that all arrive at equally spaced intervals, it is possible that no task would need to be queued and therefore response time becomes the service time.

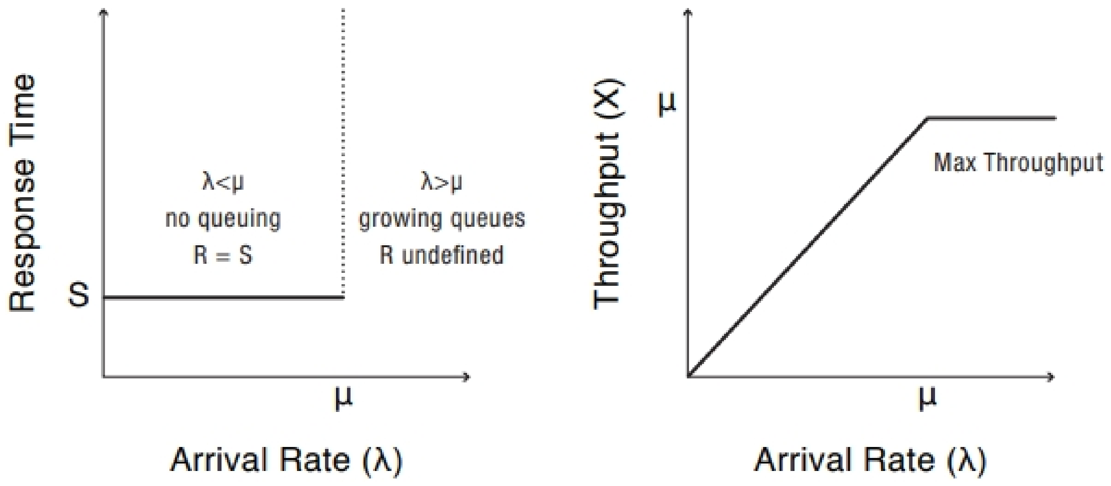

Figure 7: Response Time with Evenly Spaced Arrivals

Figure 7 shows the relationship between task arrival rate and response time for evenly spaced arrivals.

1. $\lambda < \mu$. When the arrival rate is below the service rate, there is no queuing, so $R = S$.
2. $\lambda = \mu$. When the arrival rate equals the service rate, tasks leave the system as the same rate at which they arrive. This equilibrium can be misleading. This does not mean that W (queuing delay) is zero, but rather that the queuing delay is not increasing and is not decreasing.
3. $\lambda > \mu$. When the arrival rate exceeds the service rate, the system is no longer in equilibrium and the wait queue will grow without bound. In such a state, the response time cannot be characterized, except as *bad.*

Worst case: Bursty arrivals. When tasks arrive at unpredictable rates, or in clusters, analysis becomes more difficult. Let's consider the simpler case of a set of tasks arriving at the same time.

When a group of tasks all arrive at time T_0, one task can be serviced immediately, but the remaining tasks must wait their turn.

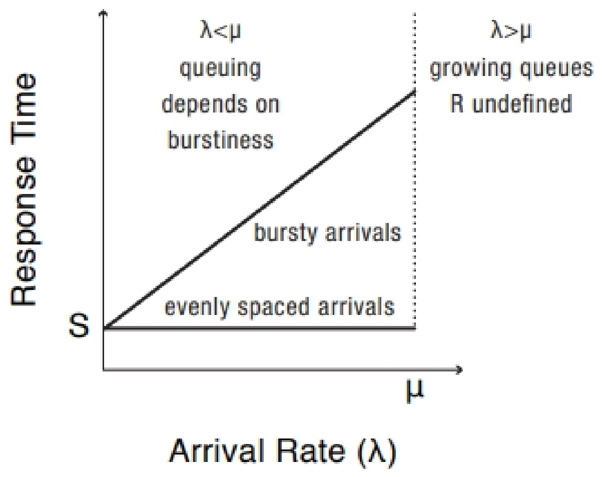

Figure 8: The Effect of Bursty Arrivals on Response Time

As can be seen from Figure 8, the larger the set of tasks arriving as a group, the worst the average response time.

If we extend this example to evenly spaced sets of bursty arrivals, then we have to consider arrival rate, λ.

$$X = \begin{cases} \lambda < \mu, & \text{Response time grows linearly with the size of the set.} \\ \lambda > \mu, & \text{Response time is undefined due to unlimite queue growth.} \end{cases}$$

Between the extremes. Most systems fall somewhere between the best case and worst case scenarios. As such, the arrival times for individual tasks appear random. Such systems are complex to model.

Modeling a more real-world system requires many simplifying assumptions and perhaps a lot of model tweaking. As a result, we will not delve into this topic here. If you are interested in probability and distributions, read the corresponding section in the text, beginning on p. 351, and we will be happy to assist you off-line. The books discusses an approach based on an exponential distribution model. There are other approaches, e.g., stochastic and Poisson models, that could be equally valid.

9 Other Schedulers

9.1 Traditional UNIX Scheduler

The scheduler used in traditional (e.g., 4.3 BSD) Unix is primarily targeted at a time-sharing environment. The target is to provide good response time for interactive jobs while ensuring that lower priority, background jobs are not starved.

This scheduler uses multi-level feedback queuing with round robin within each priority queue. Back when this scheduler was in use, a one-second preemption time interval was used (time-slice)[7]. Priority is based on process type and execution history where the following formulas apply:

[7]Processors were much slower back then.

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$
$$P_j(j) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

Where

$CPU_j(i)$	=	measure of processor utilization by process j through interval i.
$P_j(i)$	=	priority of process j at the start of interval i; lower values equate to higher priorities.
$Base_j$	=	base priority of process j, and
$nice_j$	=	user-controllable adjustment factor.

The priority of each process is recomputed once per second. Once the priorities are updated, a new scheduling decision is made. The purpose of the base priority is to divide all processes into fixed bands of priority levels. The CPU and $nice$ components are restricted to prevent a process from migrating out of its assigned priority band. The bands are used to optimize access to block devices (e.g., disk) and to allow the OS to respond quickly to system calls. The priority bands, in decreasing order of priority are:

- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

Note that the priorities above *user process* have very little work to do for any given time frame, only preempting the user process band relatively rarely. Within the user process band, the use of execution history tends to penalize processor-bound processes at the expense of I/O bound processes. For this approach, the scheduler is run 60 times per second.

9.2 Linux CFS Scheduler

Beginning with the Linux 2.6.23 kernel, a new approach, called the *Completely Fair Scheduler* was introduced. The CFS is complex and models an idealized multitasking processor on real hardware that provides fair access to all tasks. For each task, CFS maintains a *virtual runtime* value. The virtual runtime isthe amount of time spent eecuting so far normalized by the number of runnable processes. A process with a very small virtual runtime value has a high need for the processor. The CFS has other features that help, for example, ensure tasks not currently *runnable*, e.g., *sleeping* receive a fair share of the processor when they eventually become runnable.

The CFS is based on a Red Black tree, as opposed to run queues as for other schedulers.

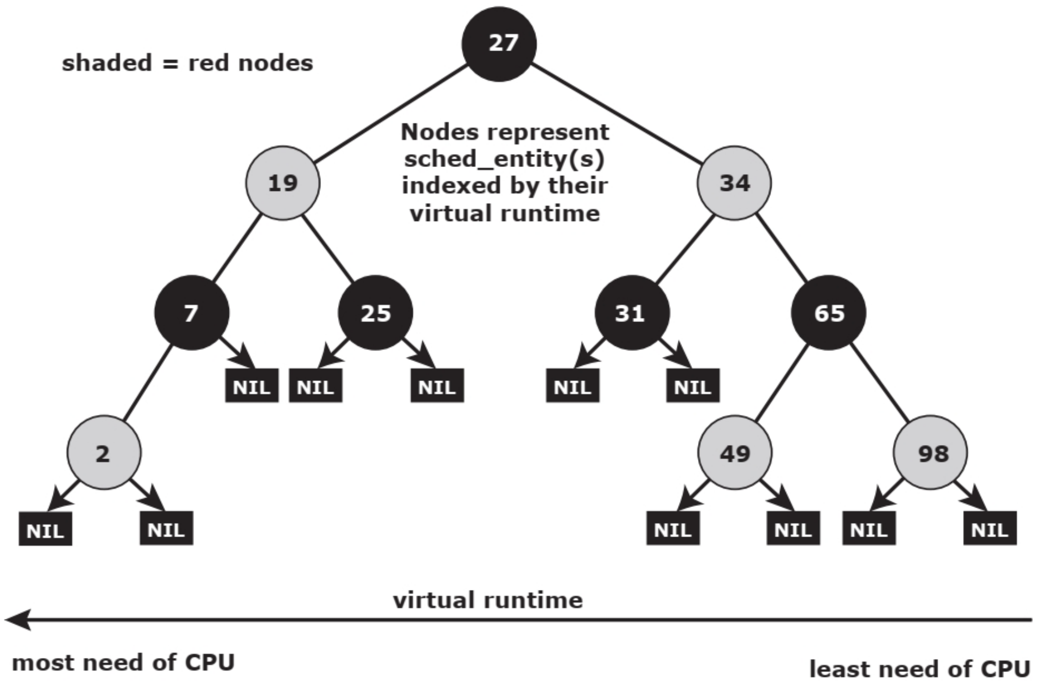

Figure 9: Example of Red Black Tree for CFS

A Red Black tree is a type of self-balancing binary search tree that obeys these rules:

1. A node it either red or black.
2. The root is black.
3. All leaves (NIL) are black.
4. If a node is red, then both of its children are black.
5. Every path from a given node up to any of its descendant NIL node contain the same number of black nodes.

This approach is highly efficient for inserting, deleting, and searching tasks due to its $\mathcal{O}(log_2 N)$ complexity.

The scheduler selects the process with the lowest virtual runtime by doing a depth-first search from the left. After running, the process will update its virtual runtime and be reinserted into the tree. Since the process has now used more processor time, it is likely to be inserted into a different spot in the tree, which is then rebalanced, and a new scheduling decision is made.

Caching and other optimizations help this approach to be even more efficient. Real-time scheduling has been left out to simplify the discussion.

9.3 Lottery Scheduler

A lottery scheduler is of a type known as *proportional-share* or *fair-share* scheduler. It operates in a markedly different manner from CFS. The basic idea is quite simple: every so often, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery. That is, the processes holding the most tickets should have the best, but not guaranteed, chance of running next. Each time a process is assigned to a processor, a ticket is consumed.

A key feature of the lottery scheduler is the simplicity of implementation. The most basic approach would be a simple linked list where each node holds the process ID and the number of tickets help by that PID. Since the drawing of a ticket is random, the placement of any one process in the list does not matter. The key here is to have a very good random number generator.

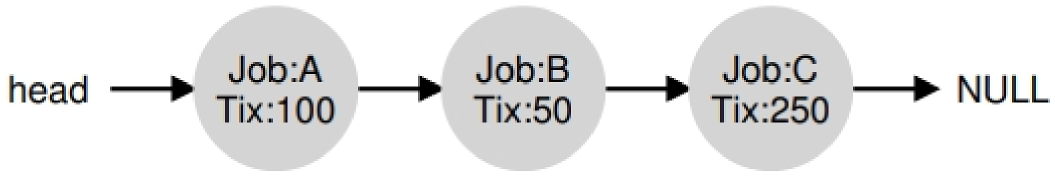

Figure 10: Lottery Scheduler Lookup

As an example, let us assume a setup as in Figure 10. At this time, the system has 400 tickets outstanding. Let us assume that the random ticket selector chooses “300”. By beginning at the head of the list, we can see that Job A does not contain the 300th ticket. Similarly with Job B. However, Job C’s 150th ticket will be the 300th ticket overall. Job C is the winner and is scheduled. Note that Job C’s ticket count is reduced to 299 and the overall number of tickets is similarly reduced to 399.

The matter of how to allocate tickets is not covered here as it is more a matter of policy than implementation.

10 Future Directions

Scheduling algorithms for multiprocessor and multi-node systems is an important area of research. Extending this type of scheduling to *distributed systems* is another research area.

We have avoided a detailed discussion of how real cache hierarchies are implemented and managed. That topic is more appropriate for a hardware architecture class or a more advanced class on operating systems.

11 Study Questions

1. What is usually the critical performance requirement in an interactive operating system?
2. What is "limited direct execution" and why is it desirable?
3. What are the two main goals operating systems being studied in this class?
4. What is the difference between turnaround time and response time?
5. What is meant by the term schedule compression?
6. What is the difference between preemptive and non-preemptive scheduling?
7. Briefly define each of the following
   (a) FCFS scheduling
   (b) Round robin scheduling
   (c) Shortest process next scheduling
   (d) Shortest remaining time scheduling
   (e) Multi-level feedback scheduling
8. Define the term "transaction"?
9. Most round-robin schedulers use a fixed size quantum. Give an argument in favor of a small quantum. Now give an argument in favor of a large quantum. Compare and contrast the types of systems and jobs to which the arguments apply. Are there any for which both are reasonable?
10. What is meant by the term "processor virtualization"?
11. What is the purpose of a system call?
12. Why is there a mode change on a system call?
13. What is the primary purpose of a system call?
14. What is an interrupt descriptor table in the Intel architecture?
15. What are the two categories of ways to change processes executing on a processor?
16. What does the term "voluntary interrupt" mean? Give an example.
17. What does the term "involuntary interrupt" mean? Give an example.
18. List and explain the steps of a context switch.
19. What are the primary benefits of a processor cache?
20. What is meant by the term "hit ratio" 'when applied to caches?
21. Why is "least recently used" (LRU) a goal for cache line replacement?
22. What is the purpose of the translation lookaside buffer?

23. Define the principle cache write policies.
24. Why is a write-back cache more complex to implement?
25. What do the following terms mean? Use your own words, no cut-and-paste.
    (a) Write allocate
    (b) No-write allocate
26. What is meant by the term “cache coherence”? Be specific.
27. What is meant by the term “transaction serialization”?
28. Explain “bus snooping”.
29. Explain the MESI protocol.
30. What is a memory barrier (membar)?
31. Why are membars useful?
32. What is the primary purpose of the scheduler?
33. What are the principle differences between a process and a thread?
34. Explain the strengths and weaknesses of user-level threads.
35. Explain the strengths and weaknesses of kernel-level threads.
36. Explain the strengths and weaknesses of combined user- and kernel-level threads.
37. Why might a mapping of one-to-one between kernel- and user-level threads be desirable? (this is an an advanced question)
38. Define, mathematically, “turnaround time”.
39. Define, mathematically, “response time”.
40. Explain what type of processes for which turnaround time is an appropriate measure. Why not response time?
41. Explain what type of processes for which response time is an appropriate measure. Why not turnaround time?
42. Define “starvation”.
43. Explain the difference between “multi-processor” and “multi-core”?
44. Define “long-term scheduler”. When is it used?
45. Define “medium-term scheduler”. When is it used?
46. Define “short-term scheduler”. When is it used?
47. Why does this class focus on short-term scheduling?

48. What are the distinguishing characteristics of the following schedulers?
    - FCFS.
    - RR
    - SPN
    - STR
    - HRRN
    - Priority
    - Feedback
49. What is meant by the term "predictability"?
50. Define "throughput".
51. What is meant by the term "fairness"?
52. Which scheduling algorithms, discussed in class, are non-preemptive?
53. Which scheduling algorithms, discussed in class, are preemptive?
54. What does the term "preemption" mean?
55. What is a time slice?
56. How is a time slice implemented?
57. Why are I/O system calls usually "voluntary" context switches?
58. What is the purpose of the ZOMBIE state in xv6?
59. RR redefines what it means for a job to arrive. Explain why this is useful.
60. What is the maximum amount of time that a process can stay in the blocked state?
61. Define "priority inversion" and give an example.
62. What is a primary weakness of fixed priorities?
63. Dynamic priority suffers from what weakness that MLFQ addresses?
64. Why doesn't MLFQ need to know which jobs are are interactive and which are long-running?
65. Explain, in your own words, rule 4 of the final MLFQ rule set.
66. What scheduler does tradition UNIX use?
67. In traditional UNIX scheduling, what is the purpose of the nice value?
68. Why is the $\mathcal{O}(1)$ no longer used?
69. Briefly explain "lottery scheduling".
70. What is the principle improvement of the "completely fair scheduler" over the "$\mathcal{O}(1)$" scheduler?

71. What was the problem with the completely fair scheduler with regards to threads?

72. Define "processor affinity" and explain why it is useful to track.

73. Define "cache affinity" ad explain why it is useful to track.

74. What is the primary difference between hard- and soft-affinity?

75. Why is it useful for each processor to run its own instance of the scheduler? Note that this is the xv6 approach.

76. Define the term "real-time scheduling".

77. What are the primary constraints for real-time schedulers?

78. Show that, among non-preemptive scheduling algorithms, SPN provides the minimum average wait time for a batch of jobs that arrive at the same time. Assume the scheduler will always execute a task if one is available.

79. Jobs A through E (5 jobs) arrive at the same time. they have estimated run times of 15, 9, 3, 6, and 12 minutes, respectively. Their priorities are 6, 3, 7, 9, and 4, respectively. A lower value represents a higher priority. For each listed scheduling algorithm, determine the turnaround time for each process and the average turnaround time for all jobs. Ignore context witch overhead (overhead == 0). Only round robin uses a time-slice. The rest run one job at a time to completion. All jobs are processor bound.

    - Round robins with a time quantum of 1 minute
    - Priority scheduling
    - FCFS – use arrival order of 15, 9, 3, 6, and 12.
    - Shortest job first.

    For this question, specify the optimal arrival order for FCFS that gives the best average turnaround time.

True/False

80. In a multiprogramming system multiple processes exist concurrently in main memory.

81. The key to multiprogramming is scheduling.

82. Scheduling affects the performance of the system because it determines which processes will wait and which will progress.

83. The main objective of long-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior.

84. In most interactive operating systems adequate response time is the critical requirement.

85. One problem with a priority scheduling scheme, that does not use feedback, is that lower-priority processes may suffer starvation.

86. FCFS performs much better for short processes than long ones.

87. Round robin is particularly effective in a general purpose time sharing system or transaction processing system

88. The objective of a fair-share scheduler is to monitor usage to give fewer resources to users who have had more than their fair share, and more to those who have had less than their fair share

89. The traditional UNIX scheduler employs multi-level feedback using round robin within each of the priority queues.

90. First-come-first-served (FCFS) is a simple scheduling policy that tends to favor I/O bound processes over processor bound processes.

91. In fair share scheduling each user is assigned a weighting of some sort that defines that user's share of system resources as a fraction of the total usage of those resources.

---

92. Which of the following scheduling policies allow the O/S to interrupt the currently running process and move it to the Ready state?

    (a) FIFO
    (b) FCFS
    (c) Non-preemptive
    (d) Preemptive

93. A risk with ____________ is the possibility of starvation for longer processes, as long as there is a steady supply of shorter processes.

    (a) SRT
    (b) SPN
    (c) FIFO
    (d) FCFS
    (e) RR

94. In the case of the ____________ policy, the scheduler always chooses the process that has the shortest expected remaining processing time.

    (a) SRT
    (b) FCFS
    (c) SPN
    (d) Priority

95. The aim of ______________________________ is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency

96. ________________________ is the elapsed time between the submission of a request until the response begins to appear as output.

97. The simplest scheduling policy is ________________________________.

98. Giving each process a slice of time before being preempted is a technique known as ________________________________.

99. A common technique for predicting a future value on the basis of a time series of past values is ________________________________.

100. The ________________________________ approach means that the operating system allocates the processor to a process and when the process blocks or is preempted, feeds it back into one of several priority queues.

101. The need to know or estimate required processing times for each process, the starvation of longer processes, and the lack of preemption are all difficulties with the ___________________ scheduling technique.

102. ________________________________ is a scheduling policy in which the process with the shortest expected processing time is selected next, and if a shorter process becomes ready in the system, the currently running process is preempted.

---

103. Match each scheduling algorithm to its attributes. Matches and choices are one-to-one.

Scheduling algorithms:

FIFO (First In First Out)

SJF (Shortest Job First)

SJN (Shortest Job Next)

STCF (Shortest Time to Completion First)

RR (Round Robin)

MLFQ (Multi-Level Feedback Queue)

Sets of attributes:

A
  i. All jobs available at the beginning, decide job order at the beginning.
  ii. Any job run order is optimal for average turnaround time.
  iii. Each job runs until it completes.
  iv. All jobs have the same run time.
  v. Starvation cannot occur.

B
  i. All jobs available at the beginning, decide job order at the beginning.
  ii. Jobs can have different run times.
  iii. Orders jobs by run time. This order is optimal for average turnaround time.
  iv. Each job runs until it completes.
  v. Starvation cannot occur.

C
  i. New jobs arrive at any time, choose which job to run next after each job completes.
  ii. Jobs can have different run times.
  iii. Orders jobs by run time.

iv. Starvation can occur.

D
i. New jobs arrive at any time, choose which job to run after each time slice.
ii. Jobs can have different run times.
iii. Orders jobs by remaining run time.
iv. Starvation can occur.

E
i. New jobs arrive at any time, choose which job to run next at the end of each time slice.
ii. Jobs can have different run times.
iii. Orders jobs using a single queue, always run the job at the head of the queue.
iv. Add new jobs to the tail of the queue.
v. When a job finishes a time slice but has not completed, add it to the tail of the queue.
vi. Starvation cannot occur.

F
i. New jobs arrive at any time, choose which job to run next at the end of each time slice.
ii. Jobs can have different run times.
iii. Orders jobs using multiple queues.
iv. Each job has a priority, each priority has a queue.
v. Add each job to the queue for its priority.
vi. Choose next job from head of highest-priority non-empty queue.
vii. Periodically promote all jobs.
viii. Demote a job each time it uses up a budget for processor time.
ix. Starvation cannot occur.

---

104. Consider the following jobs, written as (job name, run time): (P1, 60), (P2, 10), (P3, 30), (P4, 50), (P5, 20), (P6, 40). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:
Answer: P2, P5, P3, P6, P4, P1
Explanation: SJF sorts by run time from smallest to largest.

105. Consider the following situation, where we have a SJN (Shortest Job Next) scheduling algorithm and jobs are written as (job name, run time):

- Jobs (P1, 10), (P2, 50), (P3, 30) arrive.
- Scheduler chooses and runs a job.
- Current job completes.
- Scheduler chooses and runs a job.
- Jobs (P4, 20), (P5, 60), (P6, 40) arrive.
- Current job completes.
- Scheduler chooses and runs the remaining jobs to completion.

Place all jobs in the order they run:
Answer: P1, P3, P4, P6, P2, P5
Explanation:

- Jobs (P1, 10), (P2, 50), and (P3, 30) arrive.
- P1 is shortest, so P1 runs.
- Job ends.
- We have (P2, 50) and (P3, 30) remaining.
- P3 is shortest, so P3 runs.
- Jobs (P4, 20), (P5, 60), and (P6, 40) arrive.
- Time slice ends.
- We have (P2, 50), (P4, 20), (P5, 60), and (P6, 40) remaining.
- P4 is shortest, so P4 runs.
- Time slice ends.
- We have (P2, 50), (P5, 60), and (P6, 40) remaining.
- P6 is shortest, so P6 runs.
- Time slice ends.
- We have (P2, 50) and (P5, 60) remaining.
- P2 is shortest, so P2 runs.
- Time slice ends.
- We have (P5, 60) remaining.
- P5 is shortest, so P5 runs.
- Time slice ends.
- All jobs are complete.

106. Consider the following situation, where we have a STCF (Shortest Time-to-Completion First) scheduling algorithm, and jobs are written as (job name, run time). A timer interrupt runs the scheduler every 10 units of time. So, jobs are broken up into time slices of length 10. For example, (P1, 30) runs in 3 time slices: [P1: 1..10], [P1: 11..20], [P1: 21..30]. At the beginning of each time slice, the scheduler chooses a job to run from the jobs that have arrived but not yet completed.

    - Job (P1, 30) arrives.
    - Scheduler chooses and runs a job.
    - Job (P2, 30) arrives.
    - Time slice ends.
    - Scheduler chooses and runs a job.
    - Time slice ends.
    - Scheduler chooses and runs a job
    - Time slice ends.
    - Scheduler chooses and runs a job.

- Job (P3, 10) arrives.
- Time slice ends.
- Scheduler chooses and runs the remaining jobs until all are complete.

Place the time slices for all jobs in the order they run:
Answer: [P1: 1..10], [P1: 11..20], [P1: 21..30], [P2: 1..10], [P3: 1..10], [P2: 11..20], [P2: 21..30]
Explanation:

- (P1, 30) arrives.
- P1 has shortest remaining time, so [P1: 1..10] runs.
- (P2, 30) arrives.
- Time slice ends.
- We have (P1, 20), (P2, 30).
- P1 has shortest remaining time, so [P1: 11..20] runs.
- Time slice ends.
- We have (P1, 10), (P2, 30).
- P1 has shortest remaining time, so [P1: 21..30] runs.
- Time slice ends.
- We have (P2, 30).
- Only P2, so [P2: 1..10] runs.
- (P3, 10) arrives.
- Time slice ends.
- We have (P2, 20), (P3, 10).
- P3 has shortest remaining time, so [P3: 1..10] runs.
- Time slice ends
- We have (P2, 20).
- Only P2, so [P2: 11..20] runs.
- Time slice ends.
- We have (P2, 10).
- Only P2, so [P2: 21..30] runs.
- Time slice ends.
- All jobs are complete.

107. Consider the following situation, where we have a RR (Round Robin) scheduling algorithm, and jobs are written as (job name, run time). A timer interrupt runs the scheduler every 10 units of time. So, jobs are run in time slices of length 10. For example, (P1, 30) runs in 3 time slices: [P1: 1..10], [P1: 11..20], [P1: 21..30]. At the beginning of each time slice, the scheduler chooses a job to run from the jobs that have arrived but not yet completed.

    - Job (P1, 30) arrives.

- Scheduler chooses and runs a job.
- Job (P2, 20) arrives.
- Time slice ends.
- Scheduler chooses and runs a job.
- Time slice ends.
- Scheduler chooses and runs a job.
- Job (P3, 10) arrives.
- Time slice ends.
- Scheduler chooses and runs the remaining jobs until all are complete.

Place the time slices for all jobs in the order they run:
Answer: [P1: 1..10], [P2: 1..10], [P1: 11..20], [P2: 11..20], [P3: 1..10], [P1: 21..30]
Explanation:

- Job (P1, 30) arrives. Queue: (P1, 30).
- P1 is at the head of the queue, so [P1: 1..10] runs.
- Job (P2, 20) arrives. Queue: (P2, 20).
- Time slice ends. P1 is not complete, so P1 is reenqueued. Queue: (P2, 20), (P1, 20).
- P2 is at the head of the queue, so [P2: 1..10] runs.
- Time slice ends. P2 is not complete, so P2 is reenqueued. Queue: (P1, 20), (P2, 10).
- P1 is at the head of the queue, so [P1: 11..20] runs.
- Job (P3, 10) arrives. Queue: (P2, 10), (P3, 10).
- Time slice ends. P1 is not complete, so P1 is reenqueued. Queue: (P2, 10), (P3, 10), (P1, 10).
- P2 is at the head of the queue, so [P2: 11..20] runs.
- Time slice ends. P2 has completed. Queue: (P3, 10), (P1, 10).
- P3 is at the head of the queue, so [P3: 1..10] runs.
- Time slice ends. P3 has completed. Queue: (P1, 10).
- P1 is at the head of the queue, so [P1: 21..30] runs.
- Time slice ends. P1 has completed.
- All jobs are complete.

Scheduling

Question 1 (1 point)

The SJN (Shortest Job Next) scheduling algorithm prevents starvation by always scheduling the job which will take the shortest amount of time to complete.

- ○ True
- ○ False

Question 2 (1 point)

What is the principal disadvantage of STCF (Shortest Time to Completion First) scheduling?

- ○ The budget for the process may be too small.
- ○ It uses first-in first-out queuing.
- ○ It uses first-in last-out queuing.
- ○ Shortest job may never run.
- ○ Some jobs may never run.

Question 3 (1 point)

A process performing I/O cannot make progress until the I/O completes. How can the OS prevent this process from wasting CPU cycles? Assume that the system uses interrupts.

- ◯ By avoiding spin locks.
- ◯ By moving the process performing the I/O from the RUNNING to RUNNABLE state.
- ◯ By using a condition variable.
- ◯ By calling yield() after performing I/O.
- ◯ By utilizing a blocked/sleeping state.

Question 4 (1 point)

RR (Round Robin) scheduling runs a job for a specific time slice before switching to the next job in the runnable queue. This approach changes both turnaround time and response time, relative to scheduling approaches that don't use interrupts, in what way?

- ◯ Increases turnaround time, decreases response time.
- ◯ Decreases turnaround time, increases response time.
- ◯ Decreases both turnaround time and response time.
- ◯ Increases both turnaround time and response time

Question 5 (1 point)

Does transitioning processes waiting on I/O into a blocked state improve CPU utilization? Why? Choose one:

- ◯ No. Blocked or otherwise, every process still takes the same total time, so we don't use the CPU more efficiently.
- ◯ Yes. Blocked processes don't run on the CPU, so they don't waste CPU time running and waiting for I/O to complete, and runnable processes can use that CPU time instead.
- ◯ Yes. Blocking allows each running process to transition to a runnable state after its time slice expires, which improves CPU utilization.
- ◯ No. Blocking processes saves CPU time, but the overhead of managing blocked processes is greater than the time saved.

Question 6 (1 point)

What is the role of the scheduler in an operating system?

- ◯ Selects which blocked process will enter the CPU next.
- ◯ Wakes up processes blocked on I/O.
- ◯ Removes an unused process from the CPU.
- ◯ Responds to interrupts.
- ◯ Selects the next runnable process to enter the CPU.
- ◯ Creates a new process.
- ◯ "Wakes up processes blocked on I/O" and "responds to interrupts."

Question 7 (1 point)

How does the MLFQ (Multi-Level Feedback Queue) algorithm avoid starvation?

- ◯ Periodic priority adjustment upward and round robin.
- ◯ Short jobs have priority over long running jobs.
- ◯ Periodic priority adjustment downward.
- ◯ Priority queues.

MLFQ Walkthrough

Question 8 (2 points)

Consider the following single CPU system with an MLFQ scheduling algorithm in the following state:

Time budget: 100 usecs by default, and the time left in a process' budget will be denoted in parentheses after the process' name, i.e. PID(budget).

The head of a list is the first process in the list: i.e. for PA->PB->PC, PA is the head and PC is the tail.

The larger the number, the higher the priority, i.e. priority list 2 has a higher priority than priority list 1.

Priority List 2: P8(50)->P5(50)->P6(25)->P7(100)

Priority List 1: P4(100)

Priority List 0: P1(75)->P2(100)->P3(50)

If P8 were scheduled, started running on the CPU and ran for 50 usecs, and then the timer interrupt fired, which of the following would describe the resulting system state? Assume no new processes enter the system. Select all that apply:

- [ ] P8 will be added to the tail of Priority list 2 with a budget of 100 usecs.
- [ ] P8 will be added to the tail of Priority list 1 with a budget of 100 usecs.
- [ ] P4 will be the next process to run on the CPU.
- [ ] P8 will be added to the tail of Priority list 1 with a budget of 50 usecs.
- [ ] P5 will be the next process to run on the CPU.
- [ ] P8 will be added to the tail of Priority list 0 with a budget of 100 usecs.
- [ ] P1 will be the next process to run on the CPU.

Question 9 (2 points)

Match the metric best suited for each of the types of processes listed:

☐	batch accounting software	
☐	a simulation running in the background with little to no I/O	**1.** turnaround time
☐	a video game	**2.** response time
☐	the shell	

Question 10 (2 points)

Consider two processes, P_1 and P_2. Select all situations which apply to an MLFQ scheduling algorithm:

- [ ] If P_1 and P_2 have equal run times they will always be handled in RR.
- [ ] If P_1 has a shorter run time than P_2, P_1 will always be scheduled first.
- [ ] As long as P_1 is on the same priority list as P_2, P_1 and P_2 will be run in RR.
- [ ] If P_1 is on a higher priority list than P_2, P_1 will always be scheduled first.

Question 11 (2 points)

Match each property to the mechanism that enforces it in the MLFQ (Multi-Level Feedback Queue) scheduling algorithm. Matches and choices are one-to-one:

[ ]	Budget and demotion	1. Processes with the same priority do not starve each other
[ ]	Periodic promotion	2. Interactive processes do not starve long-running processes
[ ]	Always runs a process from the highest-priority non-empty queue	3. Improved response time for interactive processes
[ ]	Each priority queue is RR (Round Robin)	4. Higher-priority processes are preferred to lower-priority processes

Question 12 (2 points)

MLFQ (Multi-Level Feedback Queue) algorithms prevent starvation by which of the following? Select all that apply.

- ☐ Periodically reducing all processes' priorities to the lowest priority.
- ☐ Periodically increasing the priority for all active processes
- ☐ Randomly choosing the next process to run from the highest priority list.
- ☐ Using an RR (Round Robin) queue
- ☐ Always running the jobs which will complete the quickest.

Submit Quiz *0 of 12 questions saved*