## Contents

1 Learning Objectives 4

2 Limited Direct Execution 5
   2.1 Processor Virtualization 5
   2.2 System Calls 5
   2.3 Switching Processes 7

3 Processor Caches 8
   3.1 Examples of Hardware Caches 10
   3.2 Cache Write Policies 10
   3.3 Coherence 13
      3.3.1 Bus Snooping 15
      3.3.2 MESI 16
   3.4 Memory Barriers 17

4 Scheduling Overview 18
   4.1 Processes and Threads 19
      4.1.1 User-Level Threads 19
      4.1.2 Kernel-Level Threads 21
   4.2 Starvation 23
   4.3 Metrics 24
   4.4 Running Environments 24
   4.5 Scheduling Classifications 25
   4.6 Short-Term Scheduling 29
   4.7 Policy Characteristics 30
   4.8 Example 31

5 No Preemption 33
   5.1 First In, First Out (FIFO) 33
   5.2 Shortest Job First (SJF) 33
   5.3 Shortest Job Next (SJN / SPN) 34

6 Preemption 35
   6.1 The Time Slice 35
   6.2 Shortest Time to Completion First (STCF) 36
   6.3 Allowing I/O 36
   6.4 Round Robin (RR) 37
   6.5 Priority Scheduling 39
      6.5.1 Priority Inversion – A Cautionary Tale 40
      6.5.2 Fixed Priority 40
      6.5.3 Dynamic Priority 41
List of Figures

1. Steps of a Context Switch ........................................... 8
2. A write-through cache with no-write allocation ................. 12
3. A write-back cache with write allocation ......................... 13
4. MESI Permitted States .............................................. 17
5. User-level Threads ................................................... 20
6. One to One ............................................................ 21
7. Many to One .......................................................... 22
8. Many to Many ........................................................ 23
9. Types of Scheduling .................................................. 25
10. Queuing Diagram for Scheduling ................................... 28
11. Levels of Scheduling .................................................. 28
12. A Comparison of Scheduling Policies ............................... 32
13. Shortest Job First ..................................................... 33
14. Shortest Job First with Late Arrivals ............................. 34
15. Time quantum exceeds process lifetime ......................... 35
16. Process lifetime exceeds time quantum ......................... 35
17. Shortest Time to Completion ...................................... 36
18. Process State Model .................................................. 37
19. CPU Utilization with I/O Enabled ............................... 37
20. Round Robin .......................................................... 38
21. Round Robin with I/O ................................................. 38
22. Priority Queuing ....................................................... 39
23. MLFQ Scheduler, Budget Equals Time Slice .................... 43

List of Tables

1. Hardware Cache Examples ........................................... 10
3. Snooping Protocol Categories ...................................... 15
4. MESI Protocol .......................................................... 16
5. Differences between Process and Thread ....................... 19
6. Classification Descriptions ......................................... 27
7. Comparison of Classes .............................................. 27
8. Short-Term Scheduling Algorithm Examples ................... 29
9. Scheduling Criteria .................................................... 30
10. Policy Characteristics .................................................. 31
11. Process Information .................................................. 31
1 Learning Objectives

- Explain how uniprocessor, multiprocessor, and multicore scheduling decisions are impacted by the memory hierarchy of the computer system.
- Explain the high-level goals for a scheduling policy in a multiprocessing system.
- Explain the differences between “long running” and “interactive” processes.
- Explain the different environments for which certain scheduling policies are optimal.
- Explain why limited direct execution is necessary.
- Discuss, at a high level, the differences between traditional and real-time scheduling.
- Discuss how the presence of kernel- and user-level threads changes the approach to scheduling.
- Be able to explain, in some depth, these scheduling algorithms
  \- FIFO
  \- SJF
  \- SJN / SPN
  \- STFC / SRT
  \- RR
  \- Priority
  \- MLFQ
- Cache types
- Cache write policies
- Be able to define, distinguish, and use the terms
  \- Affinity
  \- Scheduling
  \- Multicore
  \- Multiprocessor
  \- Fair-share
  \- Processor utilization
  \- Base priority
  \- Feedback
  \- Priority inversion
  \- Job arrival
  \- Response time
  \- Turnaround time
  \- Preemption
  \- Starvation
  \- Scheduling overhead
  \- Throughput
  \- Memory barrier
  \- MESI
  \- Bus snooping
  \- Cache coherence
  \- Latency
- Explain, at a high level, these scheduling algorithms
  \- Basic UNIX
  \- Lottery
  \- Lottery
  \- \(O(1)\)
  \- Completely Fair
- Discuss basic issues with differences in scheduling algorithms with respect to constraints upon the computing environment.
2 Limited Direct Execution

The main goals for our operating system are:

- **Performance.** how can we implement processor virtualization (section 2.1) without adding excessive overhead to the system?

- **Control.** how can we run processes efficiently while retaining control over the CPU? Control is particularly important to the OS, as it is in charge of resources; without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access.

Obtaining high performance while maintaining control is a key challenge in designing an operating system. This is called *limited direct execution*.

This gives rise to two concerns:

1. When we run a process, how can the OS make sure it doesn’t do anything that we don’t want it to do, while still running it efficiently? That is, we need some degree of protection from rogue processes. It is necessary to isolate the operating system and all processes.

2. When we are running a process, how does the operating system stop it from running and switch to another process; thus implementing the time sharing we require to virtualize the CPU? And, of course, later return to the same process at the same point in its execution. That is, how can we repeatedly allocate a processor to a process and then take it away without the process knowing about it?

2.1 Processor Virtualization

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU 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 CPU in this manner, virtualization is achieved.

Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect. But running on the CPU 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 CPU or memory?

Thus, the approach we take is to introduce a new processor *mode*, known as *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; doing so would result in the processor raising an exception; the OS would then likely kill the process.

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.

2.2 System Calls

We are still left with a challenge, however: what should a user process do when it wishes to perform some kind of privileged operation, such as reading from disk? To enable this, virtually all modern
Scheduling

hardware provides the ability for user programs to perform a system call\(^1\).

System calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory. Most operating systems provide a few hundred calls; early Unix systems exposed a more concise subset of around twenty calls.

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 for the calling process. When finished, the OS calls a special return-from-trap instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode. The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller’s registers in order to be able to return correctly when the OS issues the return-from-trap instruction. On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process kernel stack; the return-from trap will pop these values off the stack and resume execution of the user mode program. Other hardware systems use different conventions, but the basic concepts are similar across platforms.

There is one important detail left out of this discussion: how does the trap know which code to run inside the OS? Clearly, the calling process can’t specify an address to jump to (as you would when making a procedure call); doing so would allow programs to jump anywhere into the kernel which clearly is a Very Bad Idea. Thus the kernel must carefully control what code executes upon a trap.

The kernel does so by setting up an interrupt vector table\(^2\). An interrupt vector table (IVT) is a data structure that associates a list of interrupt handlers with a list of interrupt requests in a table of interrupt vectors. Each entry of the interrupt vector table, called an interrupt vector, is the address of an interrupt handler. While the concept is common across processor architectures, IVTs may be implemented in architecture-specific fashions. For example, a dispatch table is one method of implementing an interrupt vector table. The IVT is configured at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. For example, what code should run when a hard disk interrupt takes place, when a keyboard interrupt occurs, or when a program makes a system call? The OS informs the hardware of the locations of these trap handlers, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., where to jump to in the code) when system calls and other exceptional events take place.

To specify the exact system call, a system call number is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack; the OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of protection; user code cannot specify an exact address to jump to, but rather must request a particular service via number.

One last aside: being able to execute the instruction to tell the hardware where the trap

---

1This section taken from OSTEP.

2In spite of the name, we typically say that the system call causes a “trap into the OS”. The Intel architecture uses the \texttt{int} instruction, with a specific interrupt number, to trap into the kernel with a system call number in register EAX. This interrupt number is set by the operating system at boot.
tables are is a very powerful capability. Thus, as you might have guessed, it is also a privileged operation. If you try to execute this instruction in user mode, the hardware won’t let you, and you can probably guess what will happen (hint: adios, offending program). Point to ponder: what horrible things could you do to a system if you could install your own trap table? Could you take over the machine?

There are two phases in the limited direct execution protocol. In the first (at boot time), the kernel initializes the IVT, and the CPU remembers its location for subsequent use. The kernel does so via a privileged instruction. In the second, the kernel sets up a few things when a new process is created (e.g., allocating a node on the process list, allocating memory) before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process.

When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process. The process then completes its work, and returns from main(); The C library has an exit() library call that will invoke the correct underlying system call for process termination (exit() on Linux), which traps into the OS). At this point, the OS cleans up and we are done.

2.3 Switching Processes

Switching the processor from one process to another is called a context switch. The context switch process now becomes the storing the CPU context of the process leaving the CPU, determining the next context to be loaded, and then loading the new context. When a process is not running in a CPU, its context is stored in the process control block (PCB) for that process. In other words, a context switch (Figure 1) is when the operating system removes one virtual processor from the real processor and loads another.

- **Cooperative. Synchronous.** Quite polite. Jobs know when they yield. They also know that there are other processes using the computing environment. **Voluntary yield.**

- **Interrupts. Asynchronous.** Processes do not know when they yield. Processes do not need to know about any other process in the computing environment. **Involuntary yield.**

The combination of virtual CPUs 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.
This diagram shows both an *involuntary yield* (i.e., timer interrupt) and an *voluntary yield* (i.e., system call).

### 3 Processor Caches

There is an inherent trade-off between size and speed (given that 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)\(^4\).

The buffering provided by a cache benefits both latency and throughput (bandwidth):

- **Latency.** A larger resource incurs a significant latency for access – e.g. it can take hundreds of clock cycles for a modern 4 GHz processor to reach DRAM. This is mitigated by reading in large chunks, in the hope that subsequent reads will be from nearby locations (*locality*). Prediction or explicit prefetching might also guess where future reads will come from and make requests ahead of time (*prefetching*); if done correctly the latency is bypassed altogether.

- **Throughput.** The use of a cache also allows for higher throughput from the underlying resource, by assembling multiple fine grain transfers into larger, more efficient requests. In the case of DRAM circuits, this might be served by having a wider data bus. For example, consider a program accessing bytes in a 32-bit address space, but being served by a 128-bit off-chip data bus; individual uncached byte accesses would allow only 1/16th of the total bandwidth to be used, and 80% of the data movement would be memory addresses instead of data itself. Reading larger chunks reduces the fraction of bandwidth required for transmitting address information.

- **Operation.** Hardware implements cache as a block of memory for temporary storage of data likely to be used again. Central processing units (CPUs) and hard disk drives (HDDs) frequently use a cache, as do web browsers and web servers.

---

\(^3\)Nothing is instantaneous, not light or gravity. This is an important fact to know.

\(^4\)See [Wikipedia](https://en.wikipedia.org/wiki/Cache_(computer_science)).
A cache is made up of a pool of entries. Each entry has associated data, which is a copy of the same data in some backing store. Each entry also has a tag, which specifies the identity of the data in the backing store of which the entry is a copy. Tagging allows simultaneous cache-oriented algorithms to function in multi-layered fashion without (differential relay) interference.

When the cache client (a CPU, web browser, operating system) needs to access data presumed to exist in the backing store, it first checks the cache. If an entry can be found with a tag matching that of the desired data, the data in the entry is used instead. This situation is known as a cache hit. For example, a web browser program might check its local cache on disk to see if it has a local copy of the contents of a web page at a particular URL. In this example, the URL is the tag, and the content of the web page is the data. The percentage of accesses that result in cache hits is known as the hit rate or hit ratio of the cache.

The alternative situation, when the cache is checked and found not to contain any entry with the desired tag, is known as a cache miss. This requires a more expensive access of data from the backing store. Once the requested data is retrieved, it is typically copied into the cache, ready for the next access.

During a cache miss, some other previously existing cache entry is removed in order to make room for the newly retrieved data. The heuristic used to select the entry to replace is known as the replacement policy. One popular replacement policy, “least recently used” (LRU), replaces the oldest entry, the entry that was accessed less recently than any other entry. More efficient caching algorithms compute the use-hit frequency against the size of the stored contents, as well as the latencies and throughputs for both the cache and the backing store. This works well for larger amounts of data, longer latencies, and slower throughputs, such as that experienced with hard drives and networks, but is not efficient for use within a CPU cache.

While CPU caches are generally managed entirely by hardware, a variety of software manages other caches. The page cache in main memory, which is an example of disk cache, is managed by the operating system kernel.

While the disk buffer, which is an integrated part of the hard disk drive, is sometimes misleadingly referred to as “disk cache”, its main functions are write sequencing and read prefetching. Repeated cache hits are relatively rare, due to the small size of the buffer in comparison to the capacity of the disk. However, high-end disk controllers often have their own on-board cache of the hard disk data blocks.

Finally, a fast local hard disk drive can also cache information held on even slower data storage devices, such as remote servers (NFS or web cache) or local tape drives or optical jukeboxes; such a scheme is the main concept of hierarchical storage management. Also, fast flash-based solid-state drives (SSDs) can be used as caches for slower rotational-media hard disk drives, working together as hybrid drives or solid-state hybrid drives (SSHDs).
### 3.1 Examples of Hardware Caches

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cache</td>
<td>Small memories on or close to the CPU can operate faster than the much larger main memory. Most CPUs since the 1980s have used one or more caches, sometimes in cascaded levels; modern high-end embedded, desktop, and server processors may have as many as six types of cache (between levels and functions). Examples of caches with a specific function are the D-cache and I-cache and the translation lookaside buffer for the MMU.</td>
</tr>
<tr>
<td>GPU cache</td>
<td>Earlier graphics processing units (GPUs) often had limited read-only texture caches, and introduced morton order swizzled textures to improve 2D cache coherency. Cache misses would drastically affect performance. As GPUs advanced, they have developed progressively larger and increasingly general caches, including instruction caches. These caches have grown to handle synchronization primitives between threads and atomic operations, and interface with a CPU-style MMU.</td>
</tr>
<tr>
<td>DSPs</td>
<td>Digital signal processors have similarly generalized over the years. Earlier designs used scratchpad memory fed by DMA, but modern DSPs often include a very similar set of caches to a CPU.</td>
</tr>
<tr>
<td>Translation lookaside buffer</td>
<td>A memory management unit (MMU) that fetches page table entries from main memory has a specialized cache, used for recording the results of virtual address to physical address translations. This specialized cache is called a translation lookaside buffer (TLB)(^5).</td>
</tr>
</tbody>
</table>

| Table 1: Hardware Cache Examples |

### 3.2 Cache Write Policies

When a system writes data to cache\(^6\), it must at some point write that data to lower level caches, main memory, and (possibly) the backing store. The timing of this write is controlled by what is known as the write policy. There are two basic approaches:

1. **Write-through**: Write is done synchronously both to the cache and to the backing store.

2. **Write-back** (also called write-behind): Initially, writing is done only to the cache. The write to the backing store is postponed until the modified content is about to be replaced by another cache block.

A write-back cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as dirty for later writing to the backing store. The data in these locations are written back to the backing store only when they are evicted from the cache, an effect referred to as a lazy write. For this reason, a read miss in a write-back cache (which

---

\(^5\)We will study TLBs in depth when we get to virtual memory.

\(^6\)See [Wikipedia](https://en.wikipedia.org/wiki/Cache_write_policy)
requires a block to be replaced by another) will often require two memory accesses to service: one to write the replaced data from the cache back to the store, and then one to retrieve the needed data.

Other policies may also trigger data write-back. The client may make many changes to data in the cache, and then explicitly notify the cache to write back the data.

Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:

1. **Write allocate** (also called fetch on write): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.

2. **No-write allocate** (also called write-no-allocate or write around): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in this way:

1. A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.
2. A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
Entities other than the cache may change the data in the backing store, in which case the copy in the cache may become out-of-date or stale. Alternatively, when the client updates the data in the cache, copies of those data in other caches will become stale. Communication protocols between the cache managers which keep the data consistent are known as coherency protocols.

### 3.3 Coherence

In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it\(^7\). When one of the copies of data is changed, the other

---

\(^7\)See [Wikipedia](https://en.wikipedia.org).
copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands (data) are propagated throughout the system in a timely fashion.

The following are the requirements for cache coherence:

<table>
<thead>
<tr>
<th>Write Propagation</th>
<th>Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Serialization</td>
<td>Reads/Writes to a single memory location must be seen by all processors in the same order.</td>
</tr>
</tbody>
</table>

Theoretically, coherence can be performed at the load/store granularity. However, in practice it is generally performed at the granularity of cache blocks.

Coherence defines the behavior of reads and writes to a single address location. One type of data occurring simultaneously in different cache memories is called cache coherence, or in some systems, global memory.

In a multiprocessor system, consider that more than one processor has cached a copy of the memory location X. The following conditions are necessary to achieve cache coherence:

1. In a read made by a processor P to a location X that follows a write by the same processor P to X, with no writes to X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P.

2. In a read made by a processor P1 to location X that follows a write by another processor P2 to X, with no other writes to X made by any processor occurring between the two accesses and with the read and write being sufficiently separated, X must always return the value written by P2. This condition defines the concept of coherent view of memory. Propagating the writes to the shared memory location ensures that all the caches have a coherent view of the memory. If processor P1 reads the old value of X, even after the write by P2, we can say that the memory is incoherent.

The above conditions satisfy the write propagation criteria required for cache coherence. However, they are not sufficient as they do not satisfy the transaction serialization condition. To illustrate this better, consider the following example:

A multi-processor system consists of four processors – P1, P2, P3 and P4 – all containing cached copies of a shared variable S whose initial value is 0. Processor P1 changes the value of S (in its cached copy) to 10 following which processor P2 changes the value of S in its own cached copy to 20. If we ensure only write propagation, then P3 and P4 will certainly see the changes made to S by P1 and P2. However, P3 may see the change made by P1 after seeing the change made by P2 and hence return 10 on a read to S. P4 on the other hand may see changes made by P1 and P2 in the order in which they are made and hence return 20 on a read to S. The processors P3 and P4 now have an incoherent view of the memory.

Therefore, in order to satisfy transaction serialization, and hence achieve cache coherence, the following condition along with the previous two mentioned in this section must be met:

- Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, from any two processors, the processors can never
read location X as B and then read it as A. The location X must be seen with values A and B in that order.

The alternative definition of a coherent system is via the definition of sequential consistency memory model: “[a] cache coherent system must appear to execute all thread loads and stores to a single memory location in a total order that respects the program order of each thread”. Thus, the only difference between the cache coherent system and sequentially consistent system is in the number of address locations the definition talks about (single memory location for a cache coherent system, and all memory locations for a sequentially consistent system).

Another definition is: “a multiprocessor is cache consistent if all writes to the same memory location are performed in some sequential order”.

Rarely, but especially in algorithms, coherence can instead refer to the locality of reference. Multiple copies of same data can exist in different cache simultaneously and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

### 3.3.1 Bus Snooping

A protocol for maintaining cache coherency in symmetric multiprocessing environments. In a snooping system, all caches on the bus monitor (or snoop) the bus to determine if they have a copy of the block of data that is requested on the bus. Every cache has a copy of the sharing status of every block of physical memory it has. Multiple copies of a document in a multiprocessing environment typically can be read without any coherence problems; however, a processor must have exclusive access to the bus in order to write.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write-invalidate</strong></td>
<td>The processor that is writing data causes copies in the caches of all other processors in the system to be rendered invalid before it changes its local copy. The local machine does this by sending an invalidation signal over the bus, which causes all of the other caches to check for a copy of the invalidated file. Once the cache copies have been invalidated, the data on the local machine can be updated until another processor requests it.</td>
</tr>
<tr>
<td><strong>Write-update</strong></td>
<td>The processor that is writing the data broadcasts the new data over the bus (without issuing the invalidation signal). All caches that contain copies of the data are then updated. This scheme differs from write-invalidate in that it does not create only one local copy for writes.</td>
</tr>
</tbody>
</table>

Table 3: Snooping Protocol Categories

---

8See Wikipedia for more on consistency models.
Basic ideas:

- Multiple copies are not a problem when reading.
- Processor must have exclusive access to write a word.

▷ What happens if two processors try to write to the same shared data word in the same clock cycle? The bus arbiter decides which processor gets the bus first (and this will be the processor with the first exclusive access). Then the second processor will get exclusive access. Thus, bus arbitration forces sequential behavior.

▷ This sequential consistency is the most conservative of the memory consistency models. With it, the result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved.

- All other processors sharing that data must be informed of writes.

A good example can be found at this Wikipedia site. Another site, Tutorialspoint also has an excellent tutorial with a very good example.

### 3.3.2 MESI

The MESI protocol\(^9\) is an invalidate-based cache coherence protocol, and is one of the most common protocols which support write-back caches. It is also known as the Illinois protocol (due to its development at the University of Illinois at Urbana-Champaign). Write back caches can save a lot on bandwidth that is generally wasted on a write through cache. There is always a dirty state present in write back caches which indicates that the data in the cache is different from that in main memory. Illinois Protocol requires cache to cache transfer on a miss if the block resides in another cache. This protocol reduces the number of main memory transactions with respect to the MSI protocol. This marks a significant improvement in the performance.

MESI is based on a well-defined finite state machine, or automaton\(^{10}\).

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>The cache line is present only in the current cache, and is dirty - it has been modified (M state) from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Shared state(S).</td>
</tr>
<tr>
<td>Exclusive</td>
<td>The cache line is present only in the current cache, but is clean - it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.</td>
</tr>
<tr>
<td>Shared</td>
<td>Indicates that this cache line may be stored in other caches of the machine and is clean - it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.</td>
</tr>
<tr>
<td>Invalid</td>
<td>Indicates that this cache line is invalid (unused).</td>
</tr>
</tbody>
</table>

Table 4: MESI Protocol

\(^9\)See Wikipedia.

\(^{10}\)Deterministic and non-deterministic automata are covered in CS 311, Computational Structures.
For any given pair of caches, the permitted states of a given cache line are as follows:

![Figure 4: MESI Permitted States](image)

When the block is marked M (modified) or E (exclusive), the copies of the block in other Caches are marked as I(Invalid).

The MESI protocol has a well-known downside. In case continuous read and write operations are performed by various caches on a particular block, the data has to be flushed to the bus every time. Thus the main memory will pull this on every flush and remain in a clean state. The MOESI protocol overcomes this disadvantage.

### 3.4 Memory Barriers

A memory barrier\(^\text{11}\), also known as a membar, memory fence or fence instruction, is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Memory barriers are necessary because most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but can cause unpredictable behaviour in concurrent programs and device drivers unless carefully controlled. The exact nature of an ordering constraint is hardware dependent and defined by the architecture’s memory ordering model. Some architectures provide multiple barriers for enforcing different ordering constraints.

Memory barriers are typically used when implementing low-level machine code that operates on memory shared by multiple devices. Such code includes synchronization primitives and lock-free data structures on multiprocessor systems, and device drivers that communicate with computer hardware.

When a program runs on a single-CPU machine, the hardware performs the necessary bookkeeping to ensure that the program executes as if all memory operations were performed in the order specified by the programmer (program order), so memory barriers are not necessary. However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor

\(^{11}\)See [Wikipedia](https://en.wikipedia.org)
system, or memory-mapped peripherals, out-of-order access may affect program behavior. For example, a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.

Multi-threaded programs usually use synchronization primitives provided by a high-level programming environment, such as Java and .NET Framework, or an application programming interface (API) such as POSIX Threads or Windows API. Synchronization primitives such as mutexes and semaphores are provided to synchronize access to resources from parallel threads of execution. These primitives are usually implemented with the memory barriers required to provide the expected memory visibility semantics. In such environments explicit use of memory barriers is not generally necessary.

Each API or programming environment in principle has its own high-level memory model that defines its memory visibility semantics. Although programmers do not usually need to use memory barriers in such high level environments, it is important to understand their memory visibility semantics, to the extent possible. Such understanding is not necessarily easy to achieve because memory visibility semantics are not always consistently specified or documented.

Just as programming language semantics are defined at a different level of abstraction than machine language opcodes, a programming environment’s memory model is defined at a different level of abstraction than that of a hardware memory model. It is important to understand this distinction and realize that there is not always a simple mapping between low-level hardware memory barrier semantics and the high-level memory visibility semantics of a particular programming environment. As a result, a particular platform’s implementation of POSIX Threads may employ stronger barriers than required by the specification. Programs which take advantage of memory visibility as implemented rather than as specified may not be portable.

4 Scheduling Overview

The evolution of schedulers parallel important advances in computer hardware. We will discuss only a few of the schedulers that have existed and even then only discuss the algorithms in the most basic form. This will allow us to clearly see the strengths and weaknesses of the different approaches. All the algorithms have application in today’s diverse computing environments, despite being developed in specific, historical, contexts. After all, an Internet-aware toaster hardly requires a multicore processor and the Linux operating system, complete with GUI.

Scheduling answers a very basic question:

**Which process gets the CPU next?**

This question can be a surprisingly difficult to answer.

One thing to keep in mind: running the scheduler takes time. Any time spent running the scheduler is time that the CPU cannot devote to running a user process. Scheduler latency is a large concern. If the scheduler is going to become more complex, and thus have more latency, there needs to be an offsetting benefit. Otherwise, the scheduler isn’t of very practical use. Computer systems exist to run user programs, not operating systems. Even the overall operating system has to have some benefit that offsets the amount of overhead (CPU, memory, storage, etc.) it imposes on the computing environment.

\[^{12}\text{See Talkie Toaster from Red Dwarf.}\]
4.1 Processes and Threads

Historically, schedulers focused on managing processes that only have one thread of control. However, modern computing is dominated by multi-threaded approaches. Since a thread is an *execution path*, or path that the instruction pointer (program counter) takes through the code on behalf of the thread, it makes sense to consider scheduling at the level of a thread\textsuperscript{13}. The scheduling of threads within the same process is fast and cheap; there are other benefits as well. The problem resides in the question, *does our kernel understand the concept of a thread?*

<table>
<thead>
<tr>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process is heavy weight or resource intensive.</td>
<td>Thread is light weight, taking lesser resources than a process.</td>
</tr>
<tr>
<td>Process switching needs interaction with operating system.</td>
<td>Thread switching does not need to interact with operating system.</td>
</tr>
<tr>
<td>In multiple processing environments, each process executes the same code but has its own memory and file resources.</td>
<td>All threads can share same set of open files, child processes.</td>
</tr>
<tr>
<td>If one process is blocked, then no other process can execute until the first process is unblocked.</td>
<td>While one thread is blocked and waiting, a second thread in the same task can run.</td>
</tr>
<tr>
<td>Multiple processes without using threads use more resources.</td>
<td>Multiple threaded processes use fewer resources.</td>
</tr>
<tr>
<td>In multiple processes each process operates independently of the others.</td>
<td>One thread can read, write or change another thread’s data.</td>
</tr>
</tbody>
</table>

Table 5: Differences between Process and Thread

4.1.1 User-Level Threads

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter (PC), stack, registers and a small thread control block. Also, there is no kernel involvement in synchronization for user-level threads.

\textsuperscript{13}Remember that all processes have at least one thread of control.
Some of the advantages of user-level threads are:

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.
- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch user-level threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Some of the disadvantages of user-level threads are:

- Multi-threaded applications in user-level threads cannot use multiprocessing to their advantage.
- The entire process is blocked if one user-level thread performs blocking operation.
4.1.2 Kernel-Level Threads

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, *kernel-level threads are slower than user-level threads.*

**One-to-One Model**

There is one-to-one relationship of user-level thread to the kernel-level thread – see Figure 6. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding kernel thread.

![Figure 6: One to One](image)

**Many-to-One Model**

Many-to-one model of Figure 7 maps many user level threads to one kernel-level thread. Thread management is done in user space by the thread library. When a thread makes a blocking system call, the entire process will be blocked. Only one thread can access the kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the kernel threads use the many-to-one relationship modes.
Figure 7: Many to One

Many-to-Many Model
The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The Figure 8 shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.
Some of the advantages of kernel-level threads are:

- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel routines can also be multi-threaded.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Some of the disadvantages of kernel-level threads are:

- A mode switch to kernel mode is required to transfer control from one thread to another in a process.
- Kernel-level threads are slower to create as well as manage as compared to user-level threads.

### 4.2 Starvation

Starvation is usually caused by an overly simplistic scheduling algorithm. For example, if a (poorly designed) multi-tasking system always switches between the first two tasks while a third never gets to run, then the third task is being starved of CPU time. The scheduling algorithm, which is part of the kernel, is supposed to allocate resources equitably; that is, the algorithm should allocate resources such that no process perpetually (permanently) lacks necessary resources.

Many operating system schedulers employ the concept of process priority. A high priority process A will run before a low priority process B. If the high priority process (process A) blocks and never yields, the low priority process (B) will, in some systems, never be scheduled – it will experience starvation. If there is an even higher priority process X, which is dependent on a result from process B, then process X might never finish, even though it is the most important process in
Scheduling

the system. This condition is called a priority inversion. Modern scheduling algorithms normally contain code to guarantee that all processes will receive a minimum amount of each important resource (most often CPU time) in order to prevent any process from being subjected to starvation.

Starvation is normally caused by deadlock in that it causes a process to freeze. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set. On the other hand, a process is in starvation when it is waiting for a resource that is continuously given to other processes. Starvation-freedom is a stronger guarantee than the absence of deadlock: a mutual exclusion algorithm that must choose to allow one of two processes into a critical section and picks one arbitrarily is deadlock-free, but not starvation-free.

A possible solution to starvation is to use a scheduling algorithm with priority queue that also uses the aging technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. MLFQ is a well-know technique that uses aging.

4.3 Metrics

We will use two simple metrics to help us decide the suitability of the different approaches.

- **Turnaround time**
  \[ T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \]
  Long running jobs with no user interaction (CPU bound)
  Simple, logical
  Can be used with deadline scheduling
  Simple and easy to calculate

- **Response time**
  \[ T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}} \]
  Interactive jobs
  Not really intuitive, but useful, as we will see
  Simple and easy to calculate

There are two type of processes:

- CPU bound. Long running with little or no I/O
- Interactive. Frequent use of I/O, such as a process that prints to the display and requests user input.

4.4 Running Environments

- Uniprocessor
  1. Batch
  2. Multiprogramming processor bound
  3. Multiprogramming interactive
  4. Mixed processor bound and interactive

- Multiprocessor and Multicore
4.5 Scheduling Classifications

The new suspend states, ready and blocked, are not part of the basic process lifetime model. However, most modern systems make use of them as having these states allows certain optimizations, especially with regards to memory utilization.
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-term</td>
<td>The long-term scheduler, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in main memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time – whether many or few processes are to be executed concurrently, and how the split between I/O-intensive and CPU-intensive processes is to be handled. The long-term scheduler is responsible for controlling the degree of multiprogramming. In general, most processes can be described as either I/O-bound or CPU-bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that a long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. On the other hand, if all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes. In modern operating systems, this is used to make sure that real-time processes get enough CPU time to finish their tasks.</td>
</tr>
<tr>
<td>Medium-term</td>
<td>The medium-term scheduler temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa, which is commonly referred to as “swapping out” or “swapping in” (also incorrectly as “paging out” or “paging in”). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.</td>
</tr>
</tbody>
</table>
The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers – a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as “voluntary” or “co-operative”), in which case the scheduler is unable to “force” processes off the CPU.

A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-term</td>
<td>The short-term scheduler decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers – a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as “voluntary” or “co-operative”), in which case the scheduler is unable to “force” processes off the CPU. A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.</td>
</tr>
</tbody>
</table>

Table 6: Classification Descriptions

<table>
<thead>
<tr>
<th>Long-Term Scheduler</th>
<th>Short-Term Scheduler</th>
<th>Medium-Term Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is a job scheduler</td>
<td>It is a CPU scheduler</td>
<td>It is a process swapping scheduler.</td>
</tr>
<tr>
<td>Speed is lesser than short term scheduler</td>
<td>Best speed</td>
<td>Speed is in between both short and long term scheduler.</td>
</tr>
<tr>
<td>It controls the degree of multiprogramming</td>
<td>It provides lesser control over degree of multiprogramming</td>
<td>It reduces the degree of multiprogramming.</td>
</tr>
<tr>
<td>It is almost absent or minimal in time sharing system</td>
<td>It is also minimal in time sharing system</td>
<td>It is a part of time sharing systems.</td>
</tr>
<tr>
<td>It selects processes from pool and loads them into memory for execution</td>
<td>It selects those processes which are ready to execute</td>
<td>It can re-introduce the process into memory and execution can be continued.</td>
</tr>
</tbody>
</table>

Table 7: Comparison of Classes

In this class, we will focus on the short-term scheduler. The short-term scheduler determines which jobs from the ready list are placed (scheduled) on a processor. The scheduler algorithms that we will study in this class are all short-term scheduler algorithms.
Figure 10: Queuing Diagram for Scheduling

Figure 11: Levels of Scheduling
4.6 Short-Term Scheduling

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-come-first-served</td>
<td>Select the process that has been waiting the longest for a processor.</td>
</tr>
<tr>
<td>Round robin</td>
<td>Use time slicing to limit any running process to a short burst of process time (time slice) and rotate among all ready processes. Uses a FIFO queue.</td>
</tr>
<tr>
<td>Shortest process next</td>
<td>Select the process with the shortest expected processing time. Do not preempt the process (run to completion).</td>
</tr>
<tr>
<td>Shortest remaining time</td>
<td>Select the process with the shortest expected remaining processing time. Process preemption is allowed.</td>
</tr>
<tr>
<td>Highest response ratio</td>
<td>Based the scheduling decision on an estimate of the normalized turnaround time.</td>
</tr>
<tr>
<td>next</td>
<td>Establish a set of scheduling queues and allocate processes to queues based on some metric (feedback). A time budget is commonly used.</td>
</tr>
</tbody>
</table>

Table 8: Short-Term Scheduling Algorithm Examples

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User Oriented, Performance Related</strong></td>
<td></td>
</tr>
<tr>
<td>Turnaround time</td>
<td>This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.</td>
</tr>
<tr>
<td>Response time</td>
<td>For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user’s point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.</td>
</tr>
<tr>
<td>Deadlines</td>
<td>When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.</td>
</tr>
<tr>
<td><strong>User Oriented, Other</strong></td>
<td></td>
</tr>
<tr>
<td>Predictability</td>
<td>A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.</td>
</tr>
</tbody>
</table>

**Systems Oriented, Performance Related**
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Throughput</strong></td>
<td>The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.</td>
</tr>
<tr>
<td><strong>Processor utilization</strong></td>
<td>This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.</td>
</tr>
</tbody>
</table>

**System Oriented, Other**

| Fairness | In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation. |
| Priority enforcement | When processes are assigned priorities, the scheduling policy should favor higher-priority processes. |
| Resource Balancing | The scheduling policy should keep the resources of the system busy. Processes that will under utilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling. |

Table 9: Scheduling Criteria

### 4.7 Policy Characteristics

<table>
<thead>
<tr>
<th></th>
<th>FCFC</th>
<th>Round Robin</th>
<th>SPN</th>
<th>SRT</th>
<th>HRRN</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheduling Function</strong></td>
<td>(\text{max}[x])</td>
<td>constant</td>
<td>(\text{min}[s])</td>
<td>(\text{min}[s - e])</td>
<td>(\frac{w+e}{s})</td>
<td>complex</td>
</tr>
<tr>
<td><strong>Decision Mode</strong></td>
<td>Non-preemptive</td>
<td>Preemptive (at time quantum)</td>
<td>Non-preemptive</td>
<td>Preemptive (at arrival)</td>
<td>Non-preemptive</td>
<td>Preemptive (at time quantum)</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>Not emphasized</td>
<td>May be low if quantum is too small</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Not emphasized</td>
</tr>
<tr>
<td><strong>Response Time</strong></td>
<td>May be high, especially if there is a large variance in process execution times</td>
<td>Provides good response time for short processes</td>
<td>Provides good response time for short processes</td>
<td>Provides good response time</td>
<td>Provides good response time</td>
<td>Not emphasized</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>Minimum</td>
<td>Minimum</td>
<td>Can be high</td>
<td>Can be high</td>
<td>Can be high</td>
<td>Can be high</td>
</tr>
<tr>
<td>Effect on Processes</td>
<td>FCFC</td>
<td>Round Robin</td>
<td>SPN</td>
<td>SRT</td>
<td>HRRN</td>
<td>Feedback</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>-------------</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>Penalizes short processes; penalizes I/O bound processes</td>
<td>Fair treatment</td>
<td>Penalizes long processes</td>
<td>Penalizes long processes</td>
<td>Good balance</td>
<td>May favor I/O bound processes</td>
<td></td>
</tr>
<tr>
<td>Starvation</td>
<td>No</td>
<td>No</td>
<td>Possible</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
</tr>
</tbody>
</table>

Table 10: Policy Characteristics

4.8 Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 11: Process Information
Figure 12: A Comparison of Scheduling Policies

Note that “q” stands for “time quantum”, e.g., the length of a time slice.
5 No Preemption

Strange at it may seem, CPUs did not always have interrupts. Systems in this category are primarily batch and real-time systems.

5.1 First In, First Out (FIFO)

Also called first come, first serve (FCFS). This algorithm is optimal for average turnaround time in a batch environment where all jobs arrive before the first job is run and all jobs have the same run time.

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 computer required for scheduling.

What happens if we allow for jobs that do not all take the same amount of time?

5.2 Shortest Job First (SJF)

We still require that all jobs arrive a priori, but now the jobs can have different run times.

Example:
Job A will take 100 time units. Job B and C will each take 10 time units. Let’s look at two possible schedules.

![Figure 13: Shortest Job First](image)

The average turnaround time differs dramatically. The order in which we set the schedule now matters. Sorting the jobs so that the shortest job is always selected next results in another optimal schedule. However, now we have the overhead of sorting. This is low, however, because all jobs arrive before any jobs are scheduled, so it is only done once.
5.3 Shortest Job Next (SJN / SPN)

What happens if we allow jobs to arrive at any time? If we allow jobs to arrive at any time, scheduling becomes much most difficult. With late arrivals, we cannot establish an optimal total ordering a priori. Instead, we potentially have to sort the list of jobs whenever we want to schedule the next process. After all, a new job, if shorter, should move to the front of the line, right? This approach is called shortest job next\(^{14}\). SJN without preemption is known as shortest process next (SPN), although SPN often uses a technique called exponential averaging\(^{15}\).

**Example:**
Job A will take 100 time units. Job B and C will each take 10 time units. We now allow late jobs and have interrupts.

![Shortest Job First with Late Arrivals](Image)

Figure 14: Shortest Job First with Late Arrivals

Now, we have the overhead (latency) of running the scheduler after each job finishes. This is a lot of work. The benefits had better outweigh the costs! While we get optimal turnaround time for the shortest jobs, how about jobs that will take a long time? Sure, they will be delayed in the presence of shorter jobs, but could they also suffer from starvation? Unfortunately, the answer is an emphatic yes!

We need a better approach.

---

\(^{14}\)See Shortest job next at Wikipedia.

\(^{15}\)We will not cover averaging techniques, including exponential averaging, in this class. We just don’t have time. As a result, SJN and SPN will be considered equivalent terms for this class.
6 Preemption

Whoever invented interrupts ought to be awarded a medal. The *timer interrupt* is designed to allow the operating system to gain control of the processor periodically — *guaranteed!*

6.1 The Time Slice

![Figure 15: Time quantum exceeds process lifetime](image1)

![Figure 16: Process lifetime exceeds time quantum](image2)

Virtually all modern, non real-time, schedulers are preemptive, and quite willing to stop one process from running in order to run another. The scheduler employs a mechanism we learned about previously; the scheduler can perform a context switch, stopping one running process temporarily and resuming (or starting) another.

We will look at interrupts primarily from the standpoint of the time slice, which is the maximum amount of time that a process can stay in the CPU before it is forced to perform an involuntary yield. A time slice is a fixed period set for each operating system, usually by the operating systems...
designer. A process can choose to perform a voluntary yield of the CPU in less than a time slice but it cannot hold the CPU for longer than a time slice. When a time slice expires, the scheduler is run – no arguments!

With the timer interrupt, we now schedule jobs for one time slice at a time. We can even view any jobs that arrives at the back of the queue to be a “new” job in the sense that it wants to be scheduled for, at least, one more time slice. This is how we get to the definition of response time. $T_{response} = T_{scheduled} - T_{arrival}$. Now we interpret this to mean that the response time is the time it takes a process to move to the front of the queue and be scheduled. This is more meaningful than just the first time an interactive job reaches the CPU.

6.2 Shortest Time to Completion First (STCF)

Also called shortest remaining time\(^\text{16}\) and shortest time remaining. A very similar technique is called shortest remaining time (SRT). As with SJN/SPN, we will consider STCF and SRT – a preemptive version of SPN – to be equivalent, with the terms being able to be used interchangeably.

For this approach, each time that a process is removed from the CPU, we will set

$$T_{remaining} = T_{remaining} - T_{CPU}$$

where $T_{CPU}$ is the amount of time the process used during its current time slice.

![Figure 17: Shortest Time to Completion with Interrupts](image)

What about long running jobs? You guessed it, starvation.

6.3 Allowing I/O

When I/O is allowed, more issues need to be handled

- CPU utilization becomes important, requiring a more elaborate concept of process state.
- Blocked state in now necessary.

\(^{16}\text{See Shortest remaining time at Wikipedia}\)
• Blocked jobs are not scheduled.

We are able to get schedule compression.

6.4 Round Robin (RR)

Round robin uses a simple FIFO queue. This looks suspiciously like FIFO scheduling, but interrupts give this approach better applicability, especially for interactive jobs.
When a job moves to the blocked state because of I/O, the system will remember the I/O device on which the job is waiting. This shortens the time looking for the correct blocked process. This approach can also be used for any process in the blocked state. For example, jobs executing the `wait()` system call could be placed on a queue for the timer interrupt.

**Figure 20: Round Robin**

**Figure 21: Round Robin with I/O**

- New perspective on jobs
Each job is comprised of many sub-jobs based on arrival

- Time slice in length

- Ordered queue service in FIFO order

- At fixed intervals
  - Remove job from CPU
  - Put at back of queue
  - Take new job from front of queue

- Removes need for time to completion

- Good for interactive jobs

**What does it means for a job to arrive?**

With time slices, we can redefine job arrival time such that the definition for $T_{response}$ now makes more sense. Now, a job arrives when it is placed on a scheduling queue and completes when it is removed from the CPU.

What about jobs that take many time slices to complete?

### 6.5 Priority Scheduling

Includes fixed priority, dynamic priority, and multi-level feedback.

![Priority Queuing](image)

Figure 22: Priority Queuing
6.5.1 Priority Inversion – A Cautionary Tale

Priority inversion is a scenario in scheduling where a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.

This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks.

Example

Consider two tasks H and L, of high and low priority respectively, either of which can acquire exclusive use of a shared resource R. If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource. Sharing an exclusive resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time. Despite good design, however, it is possible that a third task M of medium priority (p(L) < p(M) < p(H), where p(x) represents the priority for task (x)) becomes runnable during L’s use of R. At this point, M being higher in priority than L, preempts L (since M does not depend on R), causing L to not be able to relinquish R promptly, in turn causing the highest priority process to be unable to run (that is, H suffers unexpected blockage indirectly caused by lower priority tasks like M).

In some cases, priority inversion can occur without causing immediate harm – the delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high priority task is left starved of the resources, it might lead to a system malfunction or the triggering of predefined corrective measures, such as a watchdog timer resetting the entire system. The trouble experienced by the Mars Pathfinder lander in 1997 is a classic example of problems caused by priority inversion in real-time systems17.

Priority inversion can also reduce the perceived performance of the system. Low priority tasks usually have a low priority because it is not important for them to finish promptly (for example, they might be a batch job or another non-interactive activity). Similarly, a high priority task has a high priority because it is more likely to be subject to strict time constraints – it may be providing data to an interactive user, or acting subject to real-time response guarantees. Because priority inversion results in the execution of a lower priority task blocking the high priority task, it can lead to reduced system responsiveness, or even the violation of response time guarantees.

6.5.2 Fixed Priority

In fixed priority preemptive scheduling (FPPS)18, the operating system assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower-priority processes get interrupted by incoming higher-priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.

17See Christian Poellabauer’s lecture slides.
18See Wikipedia.
Scheduling

- If the number of rankings is limited, it can be characterized as a collection of FIFO queues, one for each priority ranking. Processes in lower-priority queues are selected only when all of the higher-priority queues are empty.

- Waiting time and response time depend on the priority of the process. Higher-priority processes have smaller waiting and response times.

- Deadlines can be met by giving processes with deadlines a higher priority.

- Starvation of lower-priority processes is possible with large numbers of high-priority processes queuing for CPU time. ← caution!

6.5.3 Dynamic Priority

What if we add priority to RR. How about a simple hi/low 2 queue scheme. We’ll call the high priority queue “interactive” and the low priority queue “CPU bound”. The low priority queue will only be checked when the high priority queue is empty.

All jobs will arrive on the interactive queue. If a job uses its entire time slice, we will assume that it is a CPU bound job and place it on that queue in round robin order when the time slice expires. How well does this approach work? Is there any chance of starvation?

Sure would be nice to be able to favor short running jobs without causing problems for long running jobs.

6.6 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 we 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 CPU 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 the CPU
  - 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?
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\(^ {19} \).

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 \( O(1) \)
- MLFQ better tracks jobs that change back and forth between interactive and long running
  - Better represents the real world
  - Why?

\(^ {19} \)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.*
• Promotion strategies – reset v. one-level
  ➤ Would there be a perceivable difference between the two strategies in the real world?
  ➤ 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 CPU), its priority is reduced (i.e., it moves down one queue).
5. After some time period S, promote all jobs
   • In project four, you will use one-level promotion

Linux kernel before 2.6.0 used MLFQ.

7 Traditional Unix Scheduling

The traditional UNIX scheduler uses multi-level feedback with round robin being used within each priority queue. The system makes use of a one-second preemption. If a process does not block or complete within one second, it is preempted. Priority is based on process type and execution history. It uses these formulas:

\[
CPU_k(i) = \frac{CPU_j(i-1)}{2}
\]

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

where

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(_j(i))</td>
<td>measure of processor utilization by process (j) through interval (i)</td>
</tr>
<tr>
<td>(P_j(i))</td>
<td>priority of process (j) at the beginning of interval (i); lower values equate to higher priorities</td>
</tr>
<tr>
<td>(Base_j)</td>
<td>base priority of process (j)</td>
</tr>
<tr>
<td>(nice_j)</td>
<td>user-controllable adjustment factor(^{21})</td>
</tr>
</tbody>
</table>

The priority of each process is recomputed once per second, at which time 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 moving


\(^{21}\)See “man nice” on any UNIX / Linux system.

44
out of its assigned band. The bands are used to optimize access to block devices (e.g., disk) and allow the OS to respond quickly to system calls.

The bands, in decreasing order of priority are:

- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control (e.g., the keyboard)
- User processes

Within the user process band, the use of execution history will tend to penalize processor-bound processes at the expense of I/O-bound processes, which should improve efficiency and perceived performance.

Coupled with round robin, which uses preemption, this strategy is well equipped to satisfy the requirement for a general-purpose time share system.

8 Linux O(1) Scheduler

An O(1) scheduler\(^{22}\) is a kernel scheduling design that can schedule processes within a constant amount of time, regardless of how many processes are running on the operating system. This is an improvement over previously used O(n) schedulers, which schedule processes in an amount of time that scales linearly based with the number of processes.

The Linux 2.6.8.1 scheduler did not contain any algorithms that run in worse than O(1) time. That is, every part of the scheduler is guaranteed to execute within a certain constant amount of time regardless of how many tasks are on the system. This allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows.

There are two key data structures in the Linux 2.6.8.1 scheduler that allow for it to perform its duties in O(1) time, and its design revolves around them – runqueues and priority arrays.

The main issue with this algorithm is the complex heuristics used to mark a task as interactive or non-interactive. The algorithm tries to identify interactive processes by analyzing average sleep time (the amount of time the process spends waiting for input). Processes that sleep for long periods of time probably are most often waiting for user input, so the scheduler assumes that they are interactive. The scheduler gives a priority bonus to interactive tasks (for better throughput) while penalizing non-interactive tasks by lowering their priorities. All the calculations to determine the interactivity of tasks are complex and subject to potential miscalculations (or even gaming), causing non-interactive behavior from an interactive process.

9 Fair Share Schedulers

A proportional-share scheduler, also sometimes referred to as a fair-share scheduler, is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

\(^{22}\)See Wikipedia.
9.1 Lottery

Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.

Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways. One way is with the concept of ticket currency. Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.

Another useful mechanism is ticket transfer. With transfers, a process can temporarily hand off its tickets to another process. This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client’s behalf. To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client’s request. When finished, the server then transfers the tickets back to the client and all is as before. Finally, ticket inflation can sometimes be a useful technique. With inflation, a process can temporarily raise or lower the number of tickets it owns. Of course, in a competitive scenario with processes that do not trust one another, this makes little sense; one greedy process could give itself a vast number of tickets and take over the machine. Rather, inflation can be applied in an environment where a group of processes trust one another; in such a case, if any one process knows it needs more CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other processes.

One of the most beautiful aspects of lottery scheduling is its use of randomness. When you have to make a decision, using such a randomized approach is often a robust and simple way of doing so. Random approaches have at least three advantages over more traditional decisions. First, random often avoids strange corner-case behaviors that a more traditional algorithm may have trouble handling. For example, consider the LRU replacement policy (studied in more detail in a future chapter on virtual memory); while often a good replacement algorithm, LRU attains worst-case performance for some cyclic-sequential workloads. Random, on the other hand, has no such worst case. Second, random also is lightweight, requiring little state to track alternatives. In a traditional fair-share scheduling algorithm, tracking how much CPU each process has received requires per-process accounting, which must be updated after running each process. Doing so randomly necessitates only the most minimal of per-process state (e.g., the number of tickets each has). Finally, random can be quite fast. As long as generating a random number is quick, making the decision is also, and thus random can be used in a number of places where speed is required. Of course, the faster the need, the more random tends towards pseudo-random.

9.2 Linux Completely Fair Scheduler

The Completely Fair Scheduler (CFS) is a process scheduler which was merged into the 2.6.23 (October 2007) release of the Linux kernel and is the default scheduler. It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance.

In contrast to the previous $O(1)$ scheduler used in older Linux 2.6 kernels, the CFS scheduler implementation is not based on run queues. Instead, a red-black tree implements a “time line” of future task execution. Additionally, the scheduler uses nanosecond granularity accounting, the atomic units by which an individual process’ share of the CPU was allocated (thus making redundant the previous notion of time-slices). This precise knowledge also means that no specific
heuristics are required to determine the interactivity of a process, for example.

Like the old $O(1)$ scheduler, CFS uses a concept called “sleeper fairness”, which considers sleeping or waiting tasks equivalent to those on the runqueue. This means that interactive tasks which spend most of their time waiting for user input or other events get a comparable share of CPU time when they need it.

Con Kolivas’s work with scheduling, most significantly his implementation of “fair scheduling” named Rotating Staircase Deadline, inspired Ingo Molnár to develop his CFS, as a replacement for the earlier $O(1)$ scheduler, crediting Kolivas in his announcement. CFS is an implementation of a well-studied, classic scheduling algorithm called weighted fair queuing. Originally invented for packet networks, fair queuing had been previously applied to CPU scheduling under the name stride scheduling. CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.

The Linux kernel received a patch for CFS in November 2010 for the 2.6.38 kernel that has made the scheduler "fairer" for use on desktops and workstations. Developed by Mike Galbraith using ideas suggested by Linus Torvalds, the patch implements a feature called autogrouping that significantly boosts interactive desktop performance. The algorithm puts parent processes in the same task group as child processes. (Task groups are tied to sessions created via the `setsid()` system call) This solved the problem of slow interactive response times on multi-core and multi-CPU (SMP) systems when they were performing other tasks that use many CPU-intensive threads in those tasks. A simple explanation is that, with this patch applied, one will be able to still watch a video, read email and perform other typical desktop activities without glitches or choppiness while, say, compiling the Linux kernel or encoding video.

In 2016, the Linux scheduler was patched for better multicore performance, based on the suggestions outlined in the paper, The Linux Scheduler: A Decade of Wasted Cores.

Scheduling, as in dividing CPU cycles among threads was thought to be a solved problem. We show that this is not the case. Catering to complexities of modern hardware, a simple scheduling policy resulted in a very complex bug-prone implementation. We discovered that the Linux scheduler violates a basic work-conserving invariant: scheduling waiting threads onto idle cores. As a result, runnable threads may be stuck in runqueues for seconds while there are idle cores in the system; application performance may degrade many fold. The nature of these bugs makes it difficult to detect them with conventional tools. We fix these bugs, understand their root causes and present tools, which make catching and fixing these bugs substantially easier. Our fixes and tools will be available at git.io/vaGOW.

10 Multiprocessor / Multicore

Note: multiprocessor scheduling is an NP-hard optimization problem.

High performance on multicore processors requires that schedulers be reinvented. Traditional schedulers focus on keeping execution units busy by assigning each core a thread to run. Schedulers ought to focus, however, on high utilization of on-chip memory, rather than of execution cores, to reduce the impact of expensive DRAM and remote cache accesses. A challenge in achieving good use of on-chip memory is that the memory is split up among the cores in the form of many small
caches. Multiprocessor schedulers have to schedule tasks which may or may not be dependent upon one another. For example, take the case of reading user credentials from console, then use it to authenticate, then if authentication is successful display some data on the console. Clearly one task is dependent upon another. This is a clear case of where some kind of ordering exists between the tasks. In fact it is clear that it can be modeled with partial ordering. Then, by definition, the set of tasks constitute a lattice structure.

10.1 Affinity

An interesting and common class of workloads for shared-memory multiprocessors is multiprogrammed workloads. Because these workloads generally contain more processes than there are processors in the machine, there are two factors that increase the number of cache misses. First, several processes are forced to time-share the same cache, resulting in one process displacing the cache state previously built up by a second one. Consequently, when the second process runs again, it generates a stream of misses as it rebuilds its cache state. Second since an idle processor simply selects the highest priority runnable process, a given process often moves from one CPU to another. This frequent migration results in the process having to continuously reload its state into new caches, producing streams of cache misses. To reduce the number of misses in these workloads, processes should reuse their cached state more. One way to encourage this is to schedule each process based on its affinity to individual caches, that is, based on the amount of state that the process has accumulated in an individual cache. This technique is called cache affinity scheduling.

Processor affinity – also known as “pinning” or “cache affinity” – enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. This can be viewed as a modification of the native central queue scheduling algorithm in a symmetric multiprocessing operating system. Each item in the queue has a tag indicating its kin processor. At the time of resource allocation, each task is allocated to its kin processor in preference to others.

Processor affinity takes advantage of the fact that remnants of a process that was run on a given processor may remain in that processor’s state (for example, data in the cache memory) after another process was run on that processor. Scheduling that process to execute on the same processor improves its performance by reducing performance-degrading events such as cache misses. A practical example of processor affinity is executing multiple instances of a non-threaded application, such as some graphics-rendering software.

Scheduling-algorithm implementations vary in adherence to processor affinity. Under certain circumstances, some implementations will allow a task to change to another processor if it results in higher efficiency. For example, when two processor-intensive tasks (A and B) have affinity to one processor while another processor remains unused, many schedulers will shift task B to the second processor in order to maximize processor use. Task B will then acquire affinity with the second processor, while task A will continue to have affinity with the original processor.

---

23 Reinventing Scheduling for Multicore Systems, Silas Boyd-Wickizer, Robert Morris, M. Frans Kaashoek (MIT). Professors Morris and Kaashoek are the originators of the xv6 operating system.


25 See Wikipedia.
Processor affinity can effectively reduce cache problems, but it does not reduce the persistent load-balancing problem. Also note that processor affinity becomes more complicated in systems with non-uniform architectures. For example, a system with two dual-core hyper-threaded CPUs presents a challenge to a scheduling algorithm.

There is complete affinity between two virtual CPUs implemented on the same core via hyper-threading, partial affinity between two cores on the same physical processor (as the cores share some, but not all, cache), and no affinity between separate physical processors. As other resources are also shared, processor affinity alone cannot be used as the basis for CPU dispatching. If a process has recently run on one virtual hyper-threaded CPU in a given core, and that virtual CPU is currently busy but its partner CPU is not, cache affinity would suggest that the process should be dispatched to the idle partner CPU. However, the two virtual CPUs compete for essentially all computing, cache, and memory resources. In this situation, it would typically be more efficient to dispatch the process to a different core or CPU, if one is available. This could incur a penalty when process repopulates the cache, but overall performance could be higher as the process would not have to compete for resources within the CPU.

There are two types of affinity:

- **Soft Affinity.** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

- **Hard Affinity.** Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like `sched.setaffinity()` that supports hard affinity.

There are two ways to multi-thread a processor:

- **Coarse-Grained Multi-threading.** In coarse grained multi-threading, a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.

- **Fine-Grained Multi-threading.** This approach switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

### 10.2 Simplifying Our Approach

Having a single scheduler perform scheduling for all processors / cores is one approach that we might take. However, there are simpler ideas that can make our scheduler much simpler.

One approach is for each processor / core to have its own instance of the scheduler. Each instance often shares the same ready queue, which can be problematic as it reduces concurrency. Alternately, each processor could have a separate ready queue. This would necessitate the existence of some other part of the overall scheduling system to allocate processes to ready queues.

Another approach would be for one processor / core make all scheduling decisions, allocating process to individual ready queues based on a higher-level policy. Such an approach could even move processes from one ready queue to another as a form of feedback.
11 Real-Time

Here we concentrate on the Linux approach\(^\text{26}\).

A real-time system is one that provides guaranteed system response times for events and transactions – that is, every operation is expected to be completed within a certain rigid time period. A system is classified as hard real-time if missed deadlines cause system failure and soft real-time if the system can tolerate some missed time constraints.

Real-time systems require that tasks be executed in a strict priority order. This necessitates that only the N highest-priority tasks be running at any given point in time, where N is the number of CPUs. A variation to this requirement could be strict priority-ordered scheduling in a given subset of CPUs or scheduling domains (explained later in this article). In both cases, when a task is runnable, the scheduler must ensure that it be put on a runqueue on which it can be run immediately – that is, the real-time scheduler has to ensure system-wide strict real-time priority scheduling (SWSRPS). Unlike non-real-time systems where the scheduler needs to look only at its runqueue of tasks to make scheduling decisions, a real-time scheduler makes global scheduling decisions, taking into account all the tasks in the system at any given point. Real-time task balancing also has to be performed frequently. Task balancing can introduce cache thrashing and contention for global data (such as runqueue locks) and can degrade throughput and scalability. A real-time task scheduler would trade off throughput in favor of correctness, but at the same time, it must ensure minimal task ping-ponging.

The standard Linux kernel provides two real-time scheduling policies, SCHED_FIFO and SCHED_RR. The main real-time policy is SCHED_FIFO. It implements a first-in, first-out scheduling algorithm. When a SCHED_FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. It has no time-slices. All other tasks of lower priority will not be scheduled until it relinquishes the CPU. Two equal-priority SCHED_FIFO tasks do not preempt each other. SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted time-slices based on their priority and run until they exhaust their time-slice. Non-real-time tasks use the SCHED_NORMAL scheduling policy (older kernels had a policy named SCHED_OTHER).

From the GNU C Library manual, Realtime Scheduling.

Whenever two processes with the same absolute priority are ready to run, the kernel has a decision to make, because only one can run at a time. If the processes have absolute priority 0, the kernel makes this decision as described in Traditional Scheduling. Otherwise, the decision is as described in this section.

If two processes are ready to run but have different absolute priorities, the decision is much simpler, and is described in Absolute Priority.

Each process has a scheduling policy. For processes with absolute priority other than zero, there are two available:

- First Come First Served
- Round Robin

The most sensible case is where all the processes with a certain absolute priority have the same scheduling policy.

\(^{26}\)See Linux Journal.
In Round Robin, processes share the CPU, each one running for a small quantum of time ("time slice") and then yielding to another in a circular fashion. Of course, only processes that are ready to run and have the same absolute priority are in this circle.

In First Come First Served, the process that has been waiting the longest to run gets the CPU, and it keeps it until it voluntarily relinquishes the CPU, runs out of things to do (blocks), or gets preempted by a higher priority process.

First Come First Served, along with maximal absolute priority and careful control of interrupts and page faults, is the one to use when a process absolutely, positively has to run at full CPU speed or not at all.

Judicious use of sched_yield function invocations by processes with First Come First Served scheduling policy forms a good compromise between Round Robin and First Come First Served.

To understand how scheduling works when processes of different scheduling policies occupy the same absolute priority, you have to know the nitty gritty details of how processes enter and exit the ready to run list.

In both cases, the ready to run list is organized as a true queue, where a process gets pushed onto the tail when it becomes ready to run and is popped off the head when the scheduler decides to run it. Note that ready to run and running are two mutually exclusive states. When the scheduler runs a process, that process is no longer ready to run and no longer in the ready to run list. When the process stops running, it may go back to being ready to run again.

The only difference between a process that is assigned the Round Robin scheduling policy and a process that is assigned First Come First Serve is that in the former case, the process is automatically booted off the CPU after a certain amount of time. When that happens, the process goes back to being ready to run, which means it enters the queue at the tail. The time quantum we are talking about is small. Really small. This is not your father’s timesharing. For example, with the Linux kernel, the round robin time slice is a thousand times shorter than its typical time slice for traditional scheduling.
12 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 different 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 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. Define “priority inversion” and give an example.

61. What is a primary weakness of fixed priorities?

62. Dynamic priority suffers from what weakness that MLFQ addresses?

63. How will promotion be implemented in project four?

64. Why doesn’t MLFQ need to know which jobs 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 traditional UNIX use?

67. In traditional UNIX scheduling, what is the purpose of the nice value?

68. Why is the $O(1)$ no longer used?

69. Briefly explain “lottery scheduling”.

70. What is the principle improvement of the “completely fair scheduler” over the “$O(1)$” 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” and 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 contextwitch 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 robbins 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.

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.

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

103. FIFO (First In First Out)
   (a) All jobs available at the beginning, decide job order at the beginning.
   (b) Any job run order is optimal for average turnaround time.
   (c) Each job runs until it completes.
   (d) All jobs have the same run time.
   (e) Starvation cannot occur.

104. SJF (Shortest Job First)
   (a) All jobs available at the beginning, decide job order at the beginning.
   (b) Jobs can have different run times.
   (c) Orders jobs by run time. This order is optimal for average turnaround time.
   (d) Each job runs until it completes.
   (e) Starvation cannot occur.

105. SJN (Shortest Job Next)
   (a) New jobs arrive at any time, choose which job to run next after each job completes.
   (b) Jobs can have different run times.
   (c) Orders jobs by run time.
   (d) Starvation can occur.

106. STCF (Shortest Time to Completion First)
   (a) New jobs arrive at any time, choose which job to run after each time slice.
(b) Jobs can have different run times.
(c) Orders jobs by remaining run time.
(d) Starvation can occur.

107. RR (Round Robin)

(a) New jobs arrive at any time, choose which job to run next at the end of each time slice.
(b) Jobs can have different run times.
(c) Orders jobs using a single queue, always run the job at the head of the queue.
(d) Add new jobs to the tail of the queue.
(e) When a job finishes a time slice but has not completed, add it to the tail of the queue.
(f) Starvation cannot occur.

108. MLFQ (Multi-Level Feedback Queue)

(a) New jobs arrive at any time, choose which job to run next at the end of each time slice.
(b) Jobs can have different run times.
(c) Orders jobs using multiple queues.
(d) Each job has a priority, each priority has a queue.
(e) Add each job to the queue for its priority.
(f) Choose next job from head of highest-priority non-empty queue.
(g) Periodically promote all jobs.
(h) Demote a job each time it uses up a budget for CPU time.
(i) Starvation cannot occur.

109. 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:

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

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

(a) 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.
(b) No. Blocked or otherwise, every process still takes the same total time, so we don’t use the CPU more efficiently.
(c) No. Blocking processes saves CPU time, but the overhead of managing blocked processes is greater than the time saved.

(d) Yes. Blocking allows each running process to transition to a runnable state after its time slice expires, which improves CPU utilization. This seems confusing to me, MM

111. Consider the following jobs, written as (job name, run time): (P1, 10), (P2, 30), (P3, 60), (P4, 20), (P5, 40), (P6, 50). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

112. Consider the following jobs, written as (job name, run time): (P1, 40), (P2, 30), (P3, 20), (P4, 10), (P5, 50), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

113. Consider the following jobs, written as (job name, run time): (P1, 10), (P2, 40), (P3, 20), (P4, 30), (P5, 50), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

114. Consider the following jobs, written as (job name, run time): (P1, 10), (P2, 30), (P3, 40), (P4, 20), (P5, 50), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

115. Consider the following jobs, written as (job name, run time): (P1, 40), (P2, 30), (P3, 20), (P4, 50), (P5, 10), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

116. Consider the following jobs, written as (job name, run time): (P1, 20), (P2, 30), (P3, 50), (P4, 40), (P5, 10), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

117. Consider the following jobs, written as (job name, run time): (P1, 20), (P2, 50), (P3, 40), (P4, 30), (P5, 10), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

118. Consider the following jobs, written as (job name, run time): (P1, 50), (P2, 10), (P3, 20), (P4, 30), (P5, 40), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

119. Consider the following jobs, written as (job name, run time): (P1, 20), (P2, 10), (P3, 30), (P4, 50), (P5, 40), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:
120. Consider the following jobs, written as (job name, run time): (P1, 10), (P2, 30), (P3, 50), (P4, 20), (P5, 40), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

121. Consider the following jobs, written as (job name, run time): (P1, 30), (P2, 50), (P3, 20), (P4, 10), (P5, 40), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

122. Consider the following jobs, written as (job name, run time): (P1, 50), (P2, 40), (P3, 10), (P4, 30), (P5, 20), (P6, 60). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

123. Consider the following jobs, written as (job name, run time): (P1, 50), (P2, 40), (P3, 10), (P4, 30), (P5, 60), (P6, 20). Place them in the order the SJF (Shortest Job First) scheduling algorithm would run them:

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

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

Place all jobs in the order they run:

124. 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.

(a) Job (P1, 30) arrives.
(b) Scheduler chooses and runs a job.
(c) Jobs (P2, 10), (P3, 30) arrive.
(d) Time slice ends.
(e) Scheduler chooses and runs the remaining jobs until all are complete.
Place the time slices for all jobs in the order they run:

125. 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.


Place the time slices for all jobs in the order they run:

126. 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:

Place the time slices for all jobs in the order they run:

127. 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.

(a) Job (P1, 30) arrives.
(b) Scheduler chooses and runs a job.
(c) Job (P2, 30) arrives.
(d) Time slice ends.
(e) Scheduler chooses and runs a job.
(f) Time slice ends.
(g) Scheduler chooses and runs a job
(h) Time slice ends.
(i) Scheduler chooses and runs a job.
(j) Job (P3, 10) arrives.
(k) Time slice ends.
(l) Scheduler chooses and runs the remaining jobs until all are complete.

Place the time slices for all jobs in the order they run: