CSE 351 Lecture 19 – Memory & Caches IV, System Control Flow

Cache Writes and Data Consistency

Writes are trickier to handle because we are *changing* data but multiple copies of the data might exist in our memory hierarchy. We have different policies that govern our cache behavior when the data is already in the cache (a ***write-hit***) and when the data is not in the cache (a ***write-miss***).

On a <u>write-hit</u>:

- A ***write-through*** cache writes the change into the block in the cache AND in the level below.
- A ***write-back*** cache writes the change only into the block in the cache, but makes a note that this block has been updated (*i.e.* it holds different, more up-to-date data than the level below). This requires an additional management bit called the ***dirty bit*** to be stored for each line in the cache. The updated data is only written to the level below when a dirty line is evicted from the cache.

On a <u>write-miss</u>:

- A ***write allocate*** cache will load the requested block into the cache before executing a write-hit.
- A ***no-write allocate*** cache will skip the cache and send the write directly to the level below.

Notice that a cache must have a write-hit policy AND a write-miss policy. The most common pairing is a *write-back, write allocate* cache, which tries to avoid going to the lower level as much as possible. Next most commonly (but much less so), you will find *write-through, no-write allocate* caches, which are primarily concerned with making sure that multiple copies of data in different levels remain consistent with each other.

Cache-Friendly Code

Now that you have a better idea of the mechanics behind caches and how cache parameters and policies affect memory access patterns, our main goal is to help you build intuition as to how you might optimize your code based on your newfound understanding of the memory hierarchy!

This starts by considering how well your code uses both spatial and temporal locality. Our goal for *spatial locality* is to use the smallest memory strides possible. Our goal with *temporal locality* is to maximize our use of whatever data has been loaded into the cache before it gets evicted. This can be done by tackling parts of a larger problem/algorithm at a time to reduce a single large working set into many smaller working sets. The seminal example of this is using a technique known as ***cache blocking***, where you “block” (break into smaller chunks) the operations on a large data structure such that the chunks fit in the cache, on matrix multiplication, which we will discuss in lecture.

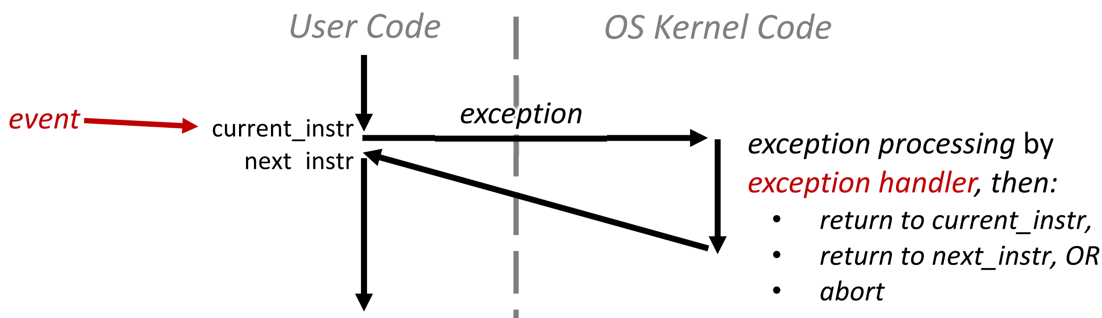
Although getting the absolute optimum performance is machine-specific and requires knowledge of your system’s parameters and how caches operate, following the guidelines above and maximizing locality can still produce generally cache-friendly code for a variety of systems.

System Control Flow

Typically, the term *control flow* has been used to describe the order of execution of individual statements within a single program, which react to changes in *program* state. But the reality is that your computer is executing multiple programs (including user programs and the operating system) concurrently. In order to achieve this, your CPU needs a mechanism by which we can react to changes in *system* state, which we call ***exceptional control flow***. This will allow us to transfer control between processes and the operating system as well as allow us to react to external signals like input and output devices.

An ***exception*** is transfer of execution/control to a part of the operating system called the ***kernel*** in response to some event. The exception will invoke some kernel code called an ***event handler*** that will try to deal with the event. There are three possible outcomes upon the completion of the event handler:

1. *Re-execute the instruction that cause the initial event* – the event has been handled, but the original instruction did not complete as intended.
2. *Execute the next instruction* – the event has been handled and the original instruction completed as intended.
3. *Abort the process* – the event could not be handled so the process needed to exit.



We will examine different categories of exceptions and examples of each of these scenarios next lecture.