Office analogy for parallel programming scheduling concepts

For more information, visit http://parallel.auckland.ac.nz/education/parallelar

This document presents an analogy that can be used to help turn parallel programming scheduling concepts into concrete mental models. The analogy is programming language agnostic, so that it applies to the basic concepts rather than implementation details of any particular programming language.

The execution of a parallelized program is represented by an office with employees (or contractors) working towards a common goal. Within this overarching analogy, alignments are made between specific technical concepts and the office environment:

- The **office space** represents the **system hardware**.
- Each **desk** represents a **processor core**.
- The existence of a **company** (employees with tasks to perform) represents an **instance of a program** running.
- An **employee/contractor** represents a **thread** in the program.
- **Hiring** an employee represents the **creation** of a thread.
- **Releasing** an employee represents the **killing off** of a thread.
- A **piece of paper** represents a **computational task**.
- A **filing cabinet** represents the **central location** of ready-to-execute tasks.

A company exists when there is work to be performed, and some office workers (employees or contractors) to perform that work. This is analogous to a program being composed of computational tasks and threads that execute those tasks. When we create a new instance of the program, we have new threads to perform those tasks.

Since a processor core can only handle one thread at a time, there is only one seat per desk. This limits the number of office workers at each desk to one.

Assume there are four desks in an office (i.e. a quad-core system). If the program is running sequentially, this means only one employee is hired to sit down at one of the four desks to complete the work. Once the work is completed, the worker will be released. This represents the thread being killed off.

However, if the same program is parallelized, the first employee hired (the main thread) will proceed to hire other employees/contractors. This is analogous to creating and starting new worker threads. The number of threads created will depend on the scheduling policy. Regardless, the main thread must wait for these spawned threads to finish their work. In the analogy, the main employee will fall asleep on the office couch while the other employees proceed to work.

Once any thread is created and assigned a task, it is ready to begin working. However, it cannot progress on a task until a processor core is available. This means that a queue
might form in the office; once a seat at a desk opens up, the next employee in the queue will sit down and continue their allocated task(s).

When there are more threads than there are processor cores, the program is executed by interleaving the work of each thread via time-sharing slices. Only one thread runs at a time on each core. If it does not complete its work during a time slice, it is paused to allow another thread to progress. So in the office, an employee may only accomplish part of their task before being placed in the back of the queue to relinquish their desk for another worker. This is especially evident in a fully-parallel scheduling policy, where a new thread is allocated for every task (wasting lots of time due to context switching).

A dynamic scheduling policy relies on a central location for storing and assigning tasks to threads during runtime. The analogy uses a filing cabinet to store these tasks, which are represented by pieces of paper (with instructions for the workers on them). When a worker is freed up, they are allocated a task from the filing cabinet (without needing to get off the desk).