OS 3/26

Reagan Shirk

March 26, 2020

Thread Scheduling

- Recall from last time: we have user and kernel level threads and you can have various combinations of both
  - i.e. one user thread to one kernel thread, multiple user threads to multiple user threads, multiple user threads to one kernel thread, one user thread to multiple kernel threads
- Convention:
  - process contention scope (user level)
    - If your process has a lot of threads in it, it's going to have to fight between those threads
    - The user can prioritize threads
  - system conention scope (kernel level)
    - Actually scheduling on the CPU
    - All of the kernel level threads have to be aware of all of the other threads going on, they need to work nicely with other processes in the system to schedule the execution of threads
    - You *might* be able to prioritize a bit but it's really up to the scheduler

Multiprocessor Scheduling

- You have to know your architecture and what it supports
- Multiprocessor appears in a lot of different ways
  - Could be a multicore CPU
  - Multithreading on the CPU
  - Non-uniform memory access
  - Other heterogeneous multiprocessing environments

SMP Systems: Symmetric multiprocessing Systems

- Things like we have in our machines, all of the processors on our machines are all the same and they can do their own things independently by themselves and we have to send tasks to each one
- Different ways of... doing this? I'm not sure what this list exactly is but I'll figure it out
  - You have a queue where each index has a task, and each task has a ... CPU? You can have your processors look for specific types of tasks
    - One single point of failure, the queue can grow very large with stress (issue)
    - There can be data structures that are shared in the queue, and accessing these data structures between each CPU can be contentious, create Race Conditions (issue), we'll talk about race conditions more in the next chapter
  - You can also have each CPU have its own queue of tasks
- tl;dr: you can have a queue of tasks with assigned CPUs or each CPU assigned a queue of tasks

Multicore Processors

- Multicores may spend a lot of time on I/O
  - You have all of these cores on one machine, and when you have I/O there is a wait/lag that needs to happen. This is called a memory stall
- Each core contains its own hardware threads, can kind of act like little CPUs
  - Each core has some memory (registers), some instructions, an ALU, etc. They all have their own architecture and can do their own thing- these are hardware threads
  - Processes can choose which core to run on, this is where it gets tricky
  - The cores will need to have context switches - need to know when it's time to give up their hold and start doing something else

Context Swtiches

- Coarse-Graind and Fine-Graind Switches
- Coarse-Graind:
  - Don't switch too often, fewer context switches
  - Remember: context switches are expensive
  - Shared resources = a lot of changes that don't get finished
- Fine-Graind:
  - Switch often!
  - Shared resources = operations can happen
  - Want to make sure that context-switches are lightweight
  - Don't want to do this if the switches take a long time

Load Balancing

- You want everything to be balanced, you want all processors carrying their weight
- In Linux, you can request for a particular process to be set to a particular core
- Push migration: the task checks the processor load and moves if the processor has too much to do
  - A type of social distancing lmao
- Pull migration: unused processors can request tasks

Processor Affinity

- You can add an affinity to processors by saying that "I want all of these tasks to use these processors"
- Advantage: you have a warm cache which means that the memory is already loaded which means there aren't any fetches that needs to be done; works well if all tasks are using the same set of memory. Take advantage of the Principle of Locality
- Linux has "soft affinity" where you can say that these tasks *should* go to these processes, but I guess they don't actually *have* to? He didn't really elaborate on the difference there
- Important for NUMA arcitectures (non-uniform memory access)
  - You want to use cores that are suited for various tasks when you have unequal processes
- Healthy tug of war between load balancing and process affinity

Queing Theory

- Little's Formula: says that the average queue length is the arrival rate times the average wait time:

$$\text{Average Queue Length} = \text{Arrival Rate} \times \text{Average Wait Time}$$

Chapter 6: Synchronization

- Only matters when there are shared resources
  - If it's not shared, we don't care
- Let's say we have a task that knows how to produce things and a task that knows how to consume things
  - There is some shared resource between the produce and consume tasks, let's call it a buffer
  - The producer produces an item and puts it inside of the buffer
  - The consumer is going to go consume the item
  - What happens when the producer tries to produce and something is already there? What happens when the consumer tries to consume and nothing is there? An error!

```
Producer:

while(true)
{
    while (buffer.full); //do nothing

    insert thing to buffer;
    buffer.full = true;
}
```

```
Consumer:

while (true)
{
    while (!buffer.full); //do nothing

    eat thing from buffer;
    buffer.full = false;
}
```

- What happens when we have more than one buffer? We'll answer that question next time.