CMPSCI 377 Operating Systems **Fall 2005**

Lecture 5: September 29

Lecturer: Emery Berger *Scribe: Pavel Sapozhnikov, Ryan Flaherty*

5.1 THREADS

What are threads?
To answer this question we'll take a look at the term **granularity**.

- Coarse-grained = gravel
- Fine-grained = sand

Thus, threads are sand while processes are gravel. Which means threads are always less than processes and one process may contain many threads
We can also look at the differences between Processes and Threads.
PROCESSES consist of Control, address space, resources, while THREADS consists of Control only.

C++ API:

- Process =

  `fork()`

- Thread =

  `pthread_create()`

So threads are easier to maintain than processes because there is less book keeping to do.

We also want to look at how OS implement threads.
OS implements threads at:

- **User level**
  User level threads require no context switching and they use thread library to manage threads. They are also more flexible than Kernel threads. User level threads can also use multiple scheduling algorithms. So why would we not want to use User level threads:
  - Cooperative threads(Things just ran)
    No Quanta
  - OS knows about processes not threads
    User threads calling I/O halts the process. Multiple threads on one process, dont get more time.

- **Kernel**
  Kernel threads are being scheduled by OS and are lightweight threads. Kernel threads require **context switch** which is very expensive. By context switch we mean, that we need to save PC, registers and stack pointers. However, there is no work that needs to be done with TLB (Virtual Memory). Kernel threads can be extended to multiple processors, which earlier systems didn't support (Sun).

So at some point Sun came up with this really fast **Hybrid model** which mapped User level threads onto lightweight processes. However, this model was too hard for the developers to understand so they didn't develop it.
Disadvantage of Hybrid model:

- 1 big disadvantage 99 percent of programmers are stupid
- Load Balancing - Spread out threads
  So there are two classic ways of implementing Load Balancing
  - **Work Sharing**
    Waste of time. Giving excess work away. This is a waste of time because you have to go around asking people if they can receive the work.
  - **Work Stealing**
    Work Stealing means get work from someone else only if I am out of work. This is the Optimal approach. This is the real deal you can't get any better than that.

5.2 SCHEDULING

There are two types of scheduling:

- **long-term**
  - how many jobs do you run at once?
- **short-term**
  - how does OS select program from ready queue to execute?
  - Kernel runs scheduler on:
    * interupt
    * when process is created

In the short-term there is **Non-preemptive** and **Preemptive** systems.

- **Non-preemptive**
  - Don't use anymore
  - Scheduler can't do anything while a process is running
- **Preemptive**
  - May interrupt a running process

Important Metrics:

- **Utilization** - percent of time that the CPU is busy
  - might be jobs that do not use CPU and you will 'starve' them
- **Throughput** - processes completing divided by time
  - not many jobs that complete on their own
  - serializing will maximize throughput
  - time slicing hurts throughput
- **Response Time** - time between ready and next I/O
  - serializing makes for a horrible response time
  - time slicing makes for good response time but has alot of context switching
- **Waiting Time** - time processes spend in ready state
  - time slicing drops waiting time

Schedulers these days must be able to handle multiple cores and multiple CPUs
Each CPU gets its own queue. If there is only one queue there will be contention.
There is also **CPU Affinity**, where a process can ask for a specific CPU each time it runs.

OS principle:

- use past behavior as a predicter of future behavior
  - works well for spinning down/saving power
- recent past = recent future
- long-term past = long-term future

Scheduling issues:

- Conflicting goals - cannot optimize all criteria simultaneously
- Want to:
  - Max - CPU utilization and throughput
  - Min - waiting time and response time

You must choose according to the system type. **Interactive** or **Services**.

Scheduling Interactive Systems:

- Want to minimize response time
  - 500 ms is the threshold
- Want to minimize rariance of response time
  - if a process takes unusually long the use gets upset

Scheduling Servers:

- Want to maximize throughput.
- Want to minimize OS overhead and context switching.

Scheduling Algorithm

FCFS (first come first serve) aka. FIFO
In the early versions the jobs did not give up the CPU even for I/O. Now we can assume the jobs will give up the CPU.
1st example:

- B wait time = 0
- C wait time = 1
- A wait time = 3

2nd example:

- A wait time = 0
- B wait time = 4
- C wait time = 5

3rd example:

- A wait time = 0
- B wait time = 1
- C wait time = 5

The advantage of FCFS is that it is simple.
The disadvatages of FCFS is that the average wait time is highly variable. It may also lead to poor overlap of I/O and CPU.