**CMPSCI 377 Operating Systems** **Fall 2005**

# Lecture 5: September 27

*Lecturer: Emery Berger* *Scribe: Gabor Asztalos, Kat Reagan*

Today:

- Intro to threads and their distinction/relation to a process

## 5.1 Threads

### 5.1.1 Threads and Processes

Threads are a subclass of processes.
Process(Recall):
-Is a unit of execution.
-Control, address space, resources.
-Call fork() to create.
Thread:
-Control only aka pc, stack, registers.
-Created with pthreadcreate().
-A process can contain many threads becuase threads are finer grain(see below).
-Threads communicate through memory yielding fastest possible communication.

Definition: Granularity - description of 'size'. Fine grain = small, coarse grain = large.

### 5.1.2 Thread System History/Evolution

MS-Dos:
-No threads.
-One process.
-Result:Nothing could be done simultaneously. Forced to wait for termination of process even for simple I/O.
Embedded Systems:
-One process.
-Many threads.
-Result:Good structure for these systems for there are usually many events to recognize but very simple responses.
Unix, Ultrix, MacOS(preX):
-Many processes.

-At most one thread per process.
-Result:Finally your 8 mhz cpu could run more than one process, but these processes could not do much at once.

Today:
-Many processes.
-Each with many threads.
-Result:Your disk can't keep up with the processor.

### 5.1.3 Types of Threads

Kernel Threads:
-Light weight processes.
-Scheduled by OS.
-Context switch required.
-Involves the OS meaning time sharing(quantas).
-Result:
-Can be scheduled on multiple processors.
-Threads can mask latency
Because don't block on I/O.
aka waiting to read from device.
-Can be slow, but still faster than processes.

User-Level Threads:
-Zero OS invlolvement.
-Process containing thread is all that is known.
ex: java green threads. Threads have own scheduler.
-No context switch.
-Result:
-Flexible, easy scheduling.
-No system calls = fast.
-Requires cooperative threads.
One uncoooperative thread can take over.
-OS only sees processes not threads.
Threads block on I/O.
Restricted to one processor.

Hybird Method:
-Uses mapping to separate and abstract.
-Kernel Thread ===- LWP's ===- user lever threads.
-Result:
-If there is a fault in user level thread like a block on I/O it creates another LWP to handle the thread. This, however, requires load balancing(see below).
-Best of both worlds, but never made it due to confussions.

Load Balancing - there are numerous processes on many processors each with many threads on todays machines. How can we best distribute the workload in the form of these threads between the processes?

1.) Collect all threads and distribute. This doesn't work because the processes are on more than one core.
2.) Work Sharing - if a process has too many threads it looks for a thread to share the work with. This causes problems because processes spend time looking instead of running and little to no work is done.
3.) Work Stealing - if a process reaches a idle state it looks for a peer to take extra work from. This is the obvious optimal solution for work is always being done.