COMSW4118 - 8-1 9/28/2017

Last time: Started Threads

- **Threads** - diff than processes, share the same address space (1 per process)

Can be implemented as:

- Purely as a user library - kernel knows nothing of user threads: very fast, but OS does not know of threads and treats them as a single process (if the user library maps them to a single process) - a system call that blocks will block all user threads & concurrency is bad since OS does not know of user threads it can't schedule them on multiple CPUs.
- Each thread mapped to a process - too heavyweight
- OS itself has threads & user library uses those - downside is you need an OS that knows how to deal with threads.

Ways to implement threads in the OS:

- You can introduce a new structure for threads on top or within of the existing structure for processes & treat them differently
- Reuse the same process structure (task-struct in Linux) for threads - that's the Linux solution:
  - a task_struct has an mm-struct for the address space - processes have their own mm but threads share this.
  - a thread group shares the same address space, a thread group leader is the first thread created for the group - the initial process that got created (the one that created the mm-struct)
  - task_struct has a member group-leader that points to another task-struct to itself (for a process)
  this gets set from **fork()** that creates the process ~~another~~ in do_fork()

? When a thread that is not the group leader forks:

- a new process gets created - do we copy all threads (in addition to memory space)? threads are independent in Linux, if we copy them over, do we include sleeping threads? and the stuff they are waiting on?
- Generally, most OSs ~~##~~ implement fork() as copying only the thing that called it, only the calling thread/process task-struct (in addition to the address space).
- Thus, fork() creates a new task-struct w/ new address space but does not copy threads.

? To distinguish process task-structs from thread ones:

The thread group leader is itself for a process, but sth. else for a thread

- **clone()** is called to create threads - it calls do_fork() (like fork()) to copy the process but with parameters what to clone/copy in copy_process().

do_fork() calls copy_process(), which sets the group_leader to self BUT then changes it to the ~~[illegible]~~ calling thread, if the clone_flags indicate it is a thread (CLONE_THREAD)

- There are a variety of clone_flags that control what is shared w/ new thread/process: address_space, signals, files

Syscall example: getpid() to return pid of caller: uses namespaces, so can't just return current.pid ← → so, there is a context for the pid (to support virtual machines)

to get the process identifier (not the calling thread one). → but, ignore for this class and just use ~~[illegible]~~ task_struct.tgid
so, this returns the thread group leader's PID!

tgid → thread group leader!

- Processes and threads are related in groups; for example:
  - ▶ killing a parent shell process terminates children
  - ▶ killing thread group leader terminates rest of group threads

▶ Purpose of using Threads (among many) is to efficiently share data
How do we make sure threads access data in a safe way?

Locking / Synchronization

▶ Race condition - value depends on order of access (race)

▶ Atomic Operations - one way to prevent garbage values

▶ Critical Section properties:

- Mutual exclusion - only one thread in it at a time
- Forward progress - if you're in critical section and no one else is, then you should be able to make progress (i.e. modify and access values)
- Bounded waiting - if you want to enter a critical section, you will not be blocked forever from entering it

⊛ Locks are commonly used to guard critical sections.

Lock L → modify data → Unlock L

Locks only work if you use them correctly! If not everyone sticks to the locking convention, the lock will not guard all access/modifications. This is both agreeing ~~on~~ and using the locks.