LINUX Threads and Concurrent Servers

Conventional Concurrent Servers
The parent process does an `accept()` and forks a process. The child process serves the client process.

Problems
- The `fork()` system call is expensive even if copy-on-write semantics are used (delayed copy of parent’s data area).
- IPC is required to pass information between the parent and child processes.

Threads Help Solve Both Problems
- Threads, also known as “light-weight processes,” are created 10-100 times faster than a process
- Threads in a process share the address space of the process
  - synchronization problem, i.e., access to the shared data by multiple threads needs to be synchronized

Threads Share
- Code (text) and data areas
- Open files (through the PPFDT)
- Current working directory
- User and group IDs
- Signal handlers and signal setups

Threads Have Their Own
- Thread ID
- CPU context (CPU registers: PC, SP, Flags, etc.)
- Stack (for activation records)
- Priority
- `errno`
LINUX Threads and Concurrent Servers

Characteristics of LINUX Threads

- Dynamic creation with `pthread_create()` function
- Concurrent execution
- Scheduling of threads like processes. A thread can give up the CPU voluntarily by calling `sched_yield()` function.
- Private stack
- Shared global variables
- Shared PPFDT
- Coordination and synchronization functions

Advantages of Threads

- Shared Memory
  - Easier to build concurrent servers that need to communicate
  - Easier to construct control and monitor systems
- Increased Efficiency
  - Lower creation cost
  - Smaller context switch time

Disadvantages of Threads

- Shared Memory
  - Race problem
  - Mutual exclusion needs to be implemented on shared memory to allow multiple threads to access data for write/update
- Many Library Functions are not Thread Safe
  - Library functions that return pointers to static data are not thread safe. For example, if multiple threads call `gethostbyname()`, the answer to one lookup may be overwritten by another coordination between threads is required for using such library functions.
LINUX Threads and Concurrent Servers

- Lack of Robustness
  - A severe error caused by one thread (e.g., segmentation fault) terminates the whole process

Thread Coordination and Synchronization

- Mutex
  - For mutually exclusive access to shared data
  - A mutex is initialized by calling `pthread_mutex_init()` function
  - A separate mutex is needed for each data item to be protected
  - A thread should call `pthread_mutex_lock()` before accessing shared data and `pthread_mutex_unlock()` after using the data.
    After the first call to these functions, the subsequent calls block.

- Semaphore
  - A counting semaphore allows \( N \) copies of a resource to be available
  - Use `sem_init()` to initialize a semaphore to \( N \) (passed as an argument)
  - Use `sem_wait()` before using a copy of the resource and `sem_post()` after returning the copy of the resource... \( N \) `sem_wait()` calls are successful and \( N+1^{st} \) blocks

Pthread Functions

- `pthread_create()`
- `pthread_join()` (equivalent of `waitpid()`)
- `pthread_self()` (equivalent of `getpid()`)
- `pthread_detach()`

A thread is either joinable or detached. When a joinable thread terminates, its TID and exit status are retained until another thread calls `pthread_join()`. A detached thread is like a daemon: when it terminates all its resources are released and a thread cannot wait for it to terminate.

- `pthread_exit(void *status)`
  - Must not be a pointer to an object which is local to the calling thread.
Ways for a Thread to Terminate

- The thread function returns (It must be declared to return a void pointer; the return value is the exit status of the thread)
- The main function of the process (the main thread) returns
- Any sibling thread calls `exit()`
- The child calls `pthread_exit()`