 Threads

What are threads?
- A **process** is a program in execution
  - It has state: PC, variable values, etc
- A traditional process has a single thread of control
  - One program counter
- A threaded process has multiple threads of control
  - Each has a program counter and it’s own path through the executable
Threads

Why do threads exist?
- To better use system resources
  - When one thread in a process blocks, another may be able to run
  - Multiple threads within a process may run concurrently
  - Threads are “lighter weight” than separate processes

Speaking of multiple processes...
- Sometimes multiple processes might be the way to go
  - Depends on needs of the system being developed and the programming language
- However, multiple processes incur costs that threads don’t
  - Threads share the memory of the process
  - This means they don’t have to use messages or other techniques to communicate
  - Switching between threads may be more efficient since less context switching is required
Example

An example from my research:
- System to plan a path through obstacles for autonomous vehicle
  - Main algorithm is a single thread
  - Graphics is a thread
  - There processor intensive part of the algorithm that occurs at regular intervals
    - This part of the problem is decomposed and allocated to a number of threads to run concurrently

Example

Multi-threaded web server:
- Concurrency needed – for obvious reasons
- Each client request handled in a separate thread
- Faster than handling each request in a separate process
  - Any downside?
Python Threads

Python supports a thread library called threading that includes:

• Thread class
  • start() — invokes the thread’s run() method
  • run() — often overridden; what the thread does
  • join() — caller blocks until the thread terminates
  • name — Thread-1 by default; can be assigned
  • is_alive() — returns True if the thread is alive
  • daemon — program terminates when no non-daemon threads remain

Producer/Consumer Problem

Problem in which multiple threads cooperate. Some produce data and others consume it.

• shared buffer
• producer places information in buffer
• consumer uses information in the buffer
• What happens if buffer is full?
• What happens if buffer is empty?
• Are other shared objects required?
Lock

A mechanism that ensures mutual exclusion
• What is mutual exclusion?
  • No two processes/threads can have simultaneous access to some resource or code section
  • For example, we may want to protect a file from simultaneous access
• How does a lock work?
  • A process/thread requests the lock. If acquired, the process/thread can proceed. Otherwise, it must wait until it acquires the lock.
  • When done, the process/thread releases the lock, making it available to others.

Semaphore

Another mechanism that ensures mutual exclusion
• How do semaphores differ from locks?
  • A semaphore is very similar to a lock except that it includes an associated value.
• How does a semaphore work?
  • Binary semaphore:
    • Begins with value 1. When acquired the value is decremented. If the value is 0, the semaphore can’t be acquired. When released the value is incremented.
  • Counting semaphore:
    • Begins with some value. Acquisition and release work the same. This allows multiple (but limited) processes/threads concurrent access.