CSE 156/L
threading pitfalls

Threads

- Main purpose is to enable **concurrency**, thereby improving the performance of the system
  - Reduce latency, i.e., using parallel algorithms
  - Overlap latency, doing other work while waiting for a long-running task to finish
  - Improve throughput by executing multiple copies of the same work/service
- Also, good to take advantage of the available hardware resources, e.g., multiple cores

Thread Correctness

- Race conditions
  - Execution order of access to shared data
  - Non-deterministic program outcome
- Deadlock
  - Take care acquiring **multiple** locks: order locks
  - Use timed non-blocking calls

Thread Safety

- Functions called from a thread must be *thread-safe*
- Four overlapping groups of thread-unsafe functions:
  1. Failing to protect shared variables
  2. Relying on persistent state across invocations
  3. Returning a pointer to a static variable
  4. Calling thread-unsafe functions

Thread-Unsafe Functions

1. Failing to protect shared variables
   - Fix: Use lock or semaphore operations
   - Issue: Synchronization operations will slow down code, due to overhead and contention

Thread-Unsafe Functions

2. Relying on persistent state across multiple function invocations
   - E.g., random number generator relies on static state
   - E.g., strtok(), breaking a string into smaller tokens
   - Fix: Rewrite function so that caller passes in all necessary state

```
static unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-Unsafe Functions

3. Returning a ptr to a `static` variable
   - Fixes:
   - a. Rewrite code so caller passes pointer to `struct`
     - Issue: Requires changes in caller and callee
   - b. Lock-and-copy
     - Issue: Requires only simple changes in caller (and none in callee)
     - However, caller must free memory

```
struct hostent *
gethostbyname(char *name)
{
  static struct hostent h;
  //contact DNS and fill in h
  return &h;
}
```

```
hostp = malloc(...);
gethostbyname_r(name, hostp);
```

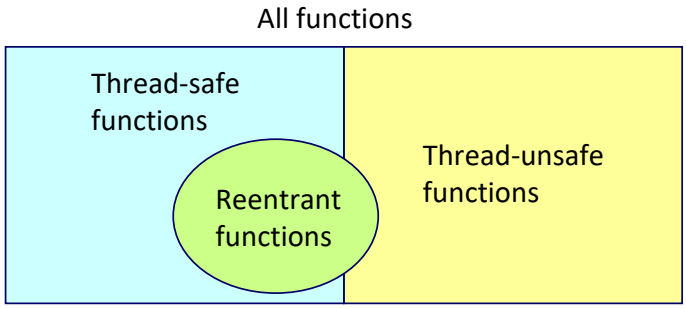
```
struct hostent *
gethostbyname_ts(char *p)
{
  struct hostent *q = malloc(...);
  lock(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;      /* copy */
  unlock(&mutex);
  return q;
}
```

Thread-Unsafe Functions

4. Calling thread-unsafe functions (libraries)
   - Calling one thread-unsafe function makes an entire function thread-unsafe
   - Fix: Modify the function so it calls only thread-safe functions

Re-entrant Functions

- A function is reentrant if it accesses NO shared variables when called from multiple threads
- Reentrant functions are not a proper subset of the set of thread-safe functions



Thread-Safe Library Functions

- Many functions in the Standard C Library are thread-safe (using synchronization internally)
  - E.g.: `malloc`, `free`, `printf`, `scanf`
- But there are a few exceptions:
  - E.g.: `strtok`, `system`, `strerror`

Thread-unsafe function	Reentrant version
`asctime`	`asctime_r`
`ctime`	`ctime_r`
`gethostbyaddr`	`gethostbyaddr_r`
`gethostbyname`	`gethostbyname_r`
`inet_ntoa`	`(none)`
`localtime`	`localtime_r`
`rand`	`rand_r`

Thread Bugs

- Deadlock (e.g., dining philosopher's problem)
  - If a thread needs more than one mutex, follow a strict ordering
  - May use mutex back-off loop to acquire all locks
- Priority inversion
  - Need three threads with different priorities
  - Lower priority thread prevents high priority thread from running

Threads Summary

- Provide another mechanism for writing concurrent programs
  - Cheaper overhead than processes
- Easy to share data between threads
- Risk of race conditions and deadlock
- Cost of easy sharing:
  - Error-prone synchronization code within a process
  - E.g., access to any global variable