CS 4400
Computer Systems

LECTURE 22

Concurrent programming, threads, and shared variables

(Chapter 12.3-12.7)

Application-Level Concurrency

- Computing in parallel on multicores
  - logical vs. physical concurrency
- Accessing slow I/O devices
  - already done by kernel, can be done at app-level
- Interacting with humans
  - create a separate concurrent flow to handle each user action
- Reducing latency by deferring work
  - defer work to a concurrent flow that runs at low priority
- Service multiple network clients
  - create a separate concurrent flow for each client (more later)

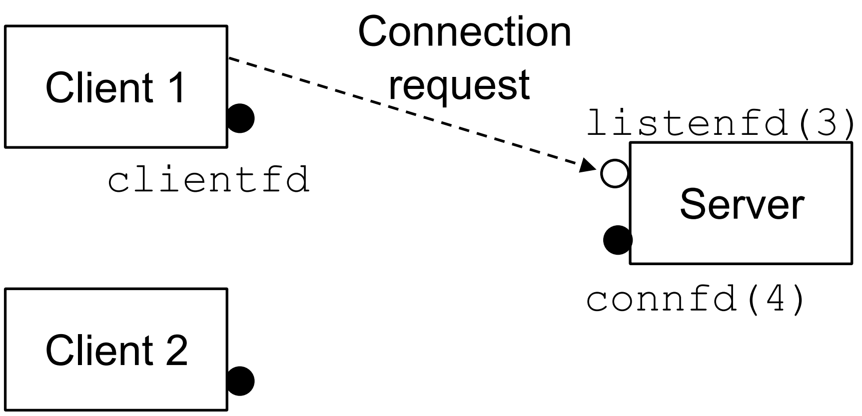
Building Concurrent Programs

- Processes
  - Each logical control flow is a processes (kernel-scheduled)
  - Separate virtual address spaces—use IPC primitives to communicate
- Threads
  - Logical flows that run in the context of a single process
  - Kernel schedules each thread
  - Hybrid approach—kernel-scheduled, shared address space
- Event-driven
  - Single non-blocking process / thread

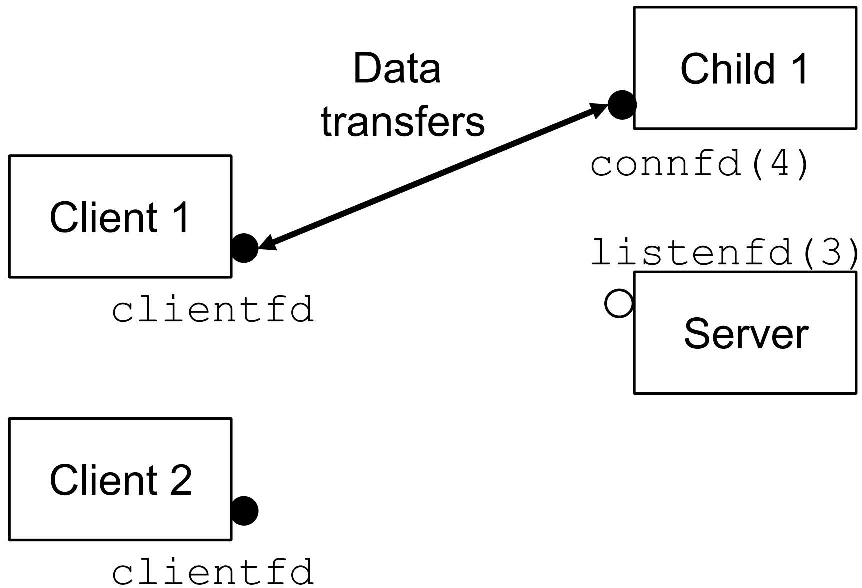
Concurrency w/ Processes

- Accept client connection requests in parent, and then create a new child process to service each new client.

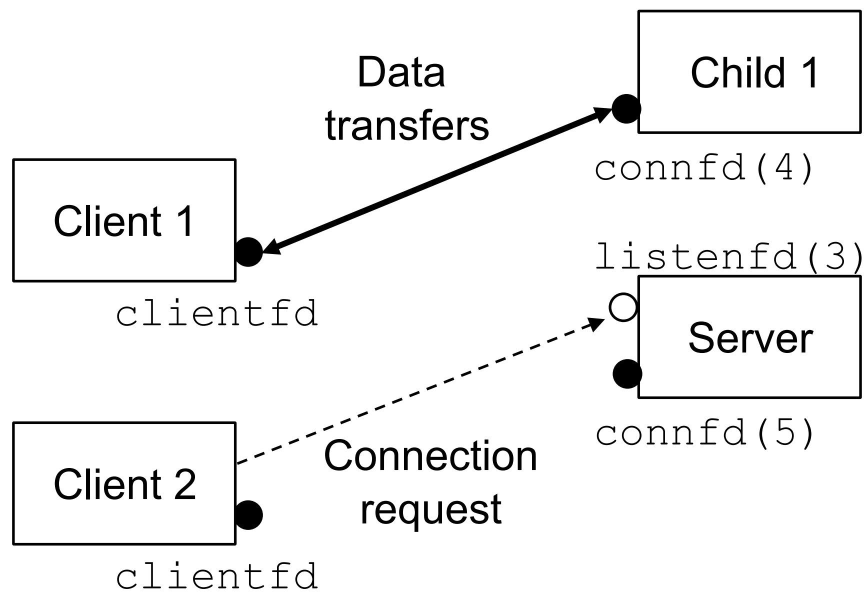
- Example: two clients, one server listening



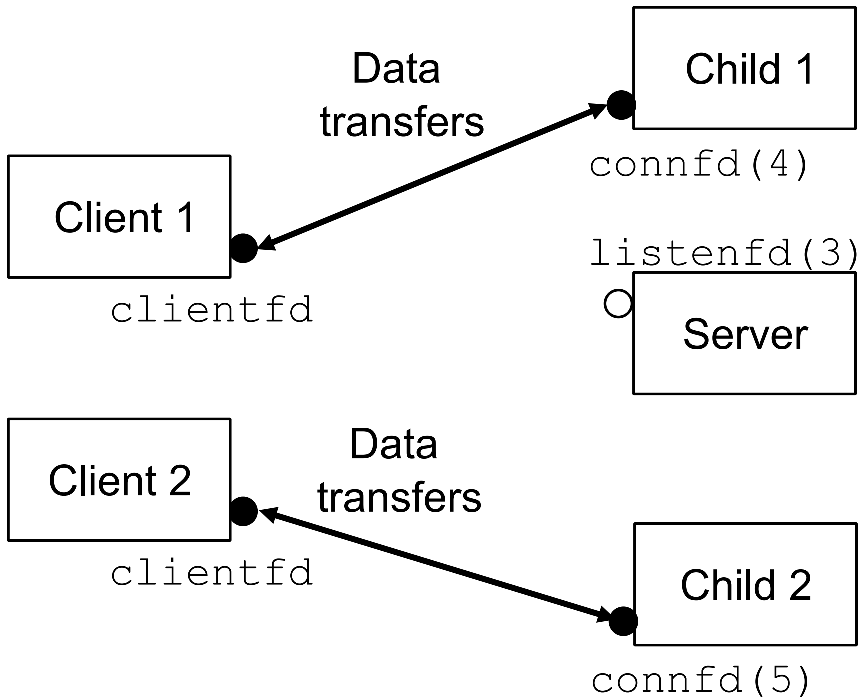
Step 1: Server accepts connection request from client



Step 2: Server forks a child process to service the client.



Step 3: Server accepts another connection request.



Step 4: Server forks another child to service new client.

(Parent is waiting for next connection request and two children are servicing their respective clients concurrently.)

Pros and Cons of Processes

- **Pro**: Clean model for sharing state information between parents and children
  - file tables are shared (child gets copy of socket descriptors)
  - user address spaces are not shared (cannot overwrite virtual memory of another process)
- **Con**: Separate address spaces make it more difficult for processes to share state information.
  - must use explicit ***interprocess communications (IPC)*** mechanisms
- **Con**: Performance as good

Threads

- A ***thread*** is a logical flow that runs in the context of a process.
  - So far, our programs have consisted of a single thread
  - The kernel automatically schedules threads

- Each thread has its own ***thread context***
  - thread ID (TID)—a unique integer
  - stack and stack pointer
  - program counter, gen-purpose registers, and condition codes

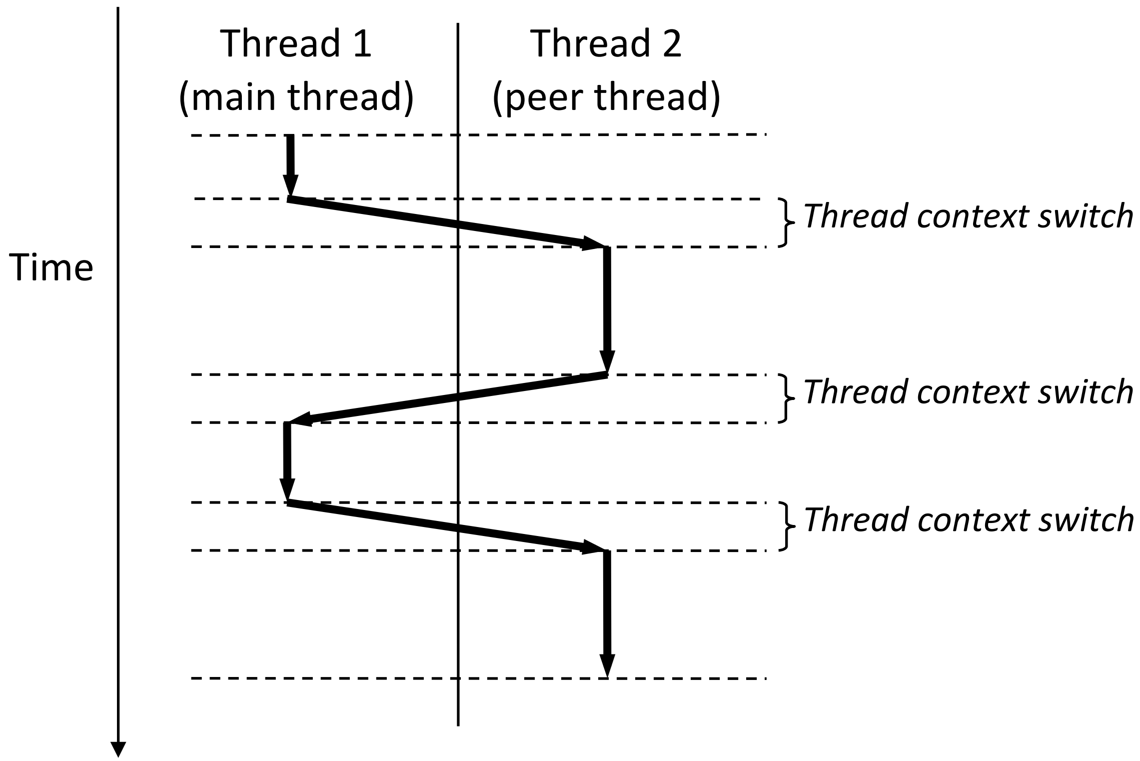
Threads

- All threads running in a process share the entire virtual address space sharing
  - code, read/write data, the heap, any shared library code/data, and the set of open files
  - if a shared memory location is modified by one thread, the other threads see the change (if they read the memory loc)

Threads vs. Processes

- A thread context switch can be faster than a process context switch
  - a thread context is much smaller than a process context
- Threads are not organized in a parent-child hierarchy
  - threads associated with a process form a pool of peers, independent of which threads were created by which other threads
  - a thread can kill any of its peers, or wait for any of its peers to terminate
  - each peer can read or write the same shared data

Execution Model



Control passes to the peer thread because the main thread executes a slow system call or is interrupted by the system's interval timer.

- Each process begins life as a single, main thread
- The main thread creates a peer thread, and from that point the two threads run concurrently

Creating Threads

```
typedef void* (func)(void*);

int pthread_create(pthread_t* tid,
    pthread_attr_t* attr, func* f, void* arg);
```

- Creates a new thread and runs the thread routine `f` in the context of the new thread and with input argument `arg`.
- `attr` can be used to change the default thread attributes.
  - we'll always use `NULL`
- Upon return, `tid` is set to the (opaque) ID of the thread
- A thread can determine its own thread ID using:

```
pthread_t pthread_self(void);
```

Terminating Threads

- A thread terminates in one of the following ways.
  - Its top-level thread routine returns
  - Through (if called by the main thread, it waits for all peer threads):

```
int pthread_exit(void* thread_return);
```

  - Calls `exit`, which terminates the process and all associated threads.
  - Another peer thread calls `pthread_cancel` with the ID of the current thread.

```
int pthread_cancel(pthread_t tid);
```

Reaping Terminated Threads

```
int pthread_join(pthread_t tid,
                 void** thread_return);
```

- Blocks until thread `tid` terminates.
- Assigns the `void*` returned by the thread routine to the location pointed to by `thread_return`.
- Reaps any memory resources held by the terminated thread.
- Unlike `wait_pid`, this function can only wait for a specific thread to terminate.

Example: Pthreads

```c
/* Pthreads is a standard interface for manipulating
 * threads from C programs. */

#include "csapp.h"
void* thread(void* vargp);

/* main thread */
int main() {
  pthread_t tid;   /* thread ID of peer thread */
  /* create peer thread */
  Pthread_create(&tid, NULL, thread, NULL);
  /*--Now, main thread and peer thread are running concurrently.--*/
  /* wait for peer thread to terminate */
  Pthread_join(tid, NULL);
  /* terminate all threads */
  exit(0);
}

/* The code and local data for a thread are encapsulated in
 * a thread routine.  Each thread routine takes as input a
 * single generic pointer and returns a generic pointer. */
void* thread(void* vargp) {
  printf("Hello, world!\n");
  return NULL;   /* terminate peer thread */
}
```

Detaching Threads

- At any time, a thread is joinable or detached
  - ***joinable***—the thread can be reaped and killed by other threads, at which time its memory resources are freed
  - ***detached***—the thread cannot be reaped or killed by other threads, and its memory resources are free automatically by the system when it terminates

- By default, all threads are created joinable

- To avoid memory leaks, each joinable thread should either be reaped by another thread or detached

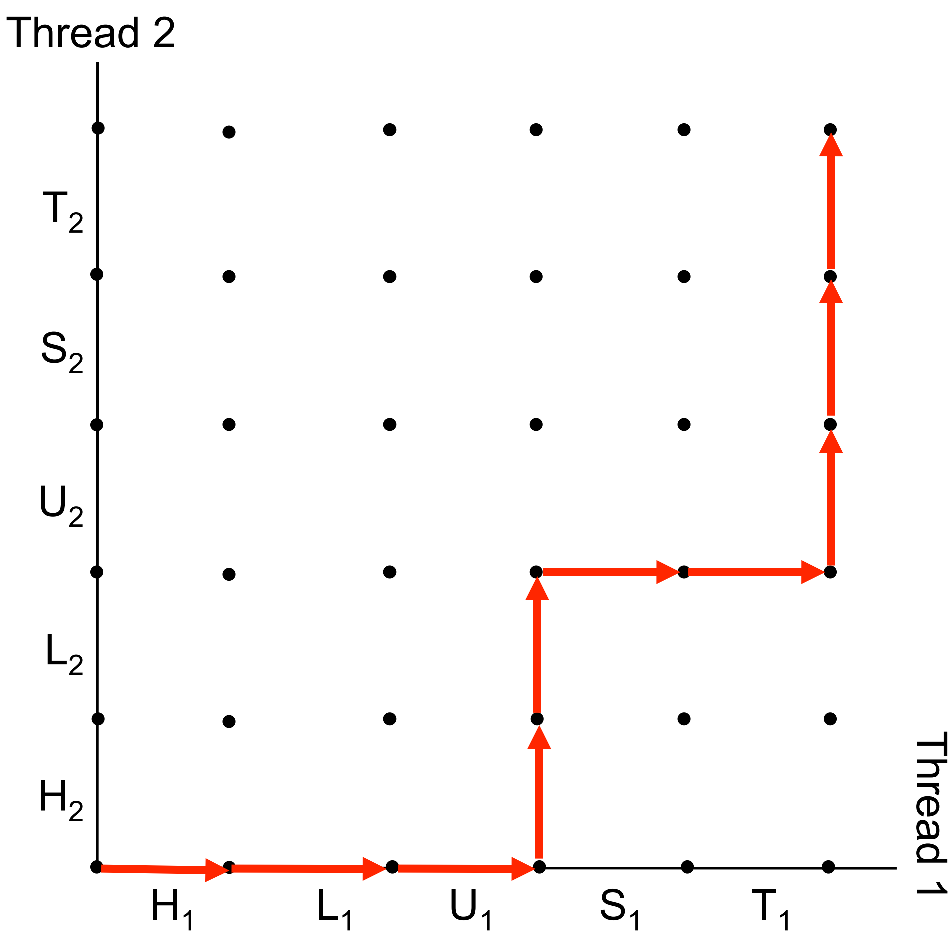
```
int pthread_detach(pthread_t tid);
```

Mapping Variables to Memory

- ***Global variable***—any variable declared outside of a function.
  - one instance that can be referenced by any thread
- ***Local automatic variables***—any variable declared inside a function without the `static` attribute.
  - each thread's stack contains its own instance (even if multiple threads execute the same thread routine)
- ***Local static variables***—any variable declared inside a function with the `static` attribute.
  - like global variables, one instance for all threads

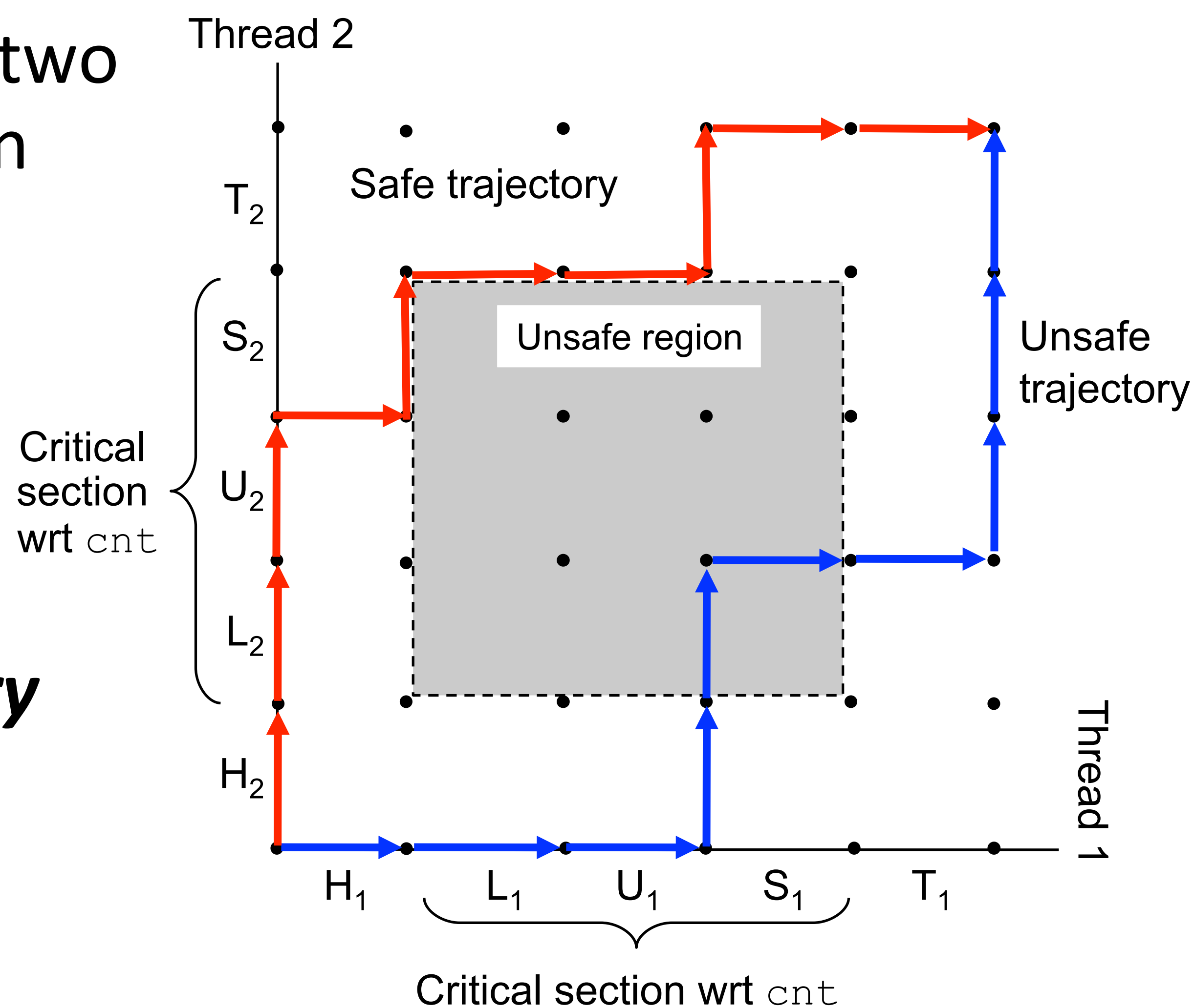
Process Graph

- Models the execution of n threads as a trajectory through an n-dimensional Cartesian space.
- each axis k shows the progress of thread k
- each point (I_1, I_2, ..., I_n) represents the state where thread k has completed instruction I_k
- the trajectory corresponds to the ordering of instructions



Critical Section

- Instructions L_i, U_i, and S_i constitute a ***critical section*** for thread *i*.
- The intersection of two critical sections is an ***unsafe region***.
- A ***safe trajectory*** skirts the unsafe region.
- An ***unsafe trajectory*** touches any part of the unsafe region.



Semaphore

- A global variable s ≥ 0 that can only be manipulated using one of two operations: P and V.

- P(s)
  - if s!= 0, s-- and return (occurs indivisibly)
  - if s = = 0, suspend the process until s becomes nonzero (process is restarted by a V operation), after restarting s-- and return

- V(s)
  - s++ and check to see if any processes are blocked in a P operation waiting for s to become nonzero (restarts exactly one of such processes)
  - increment occurs indivisibly

Posix Semaphores

- Functions for manipulating semaphores.

```
int sem_init(sem_t* sem, 0, unsigned int value);
int sem_wait(sem_t* sem);            /* P(s) */
int sem_post(sem_t* sem);    /* V(s) */
```

- Example:

```
sem_t mutex; /* semaphore to synch cnt access */
sem_init(&mutex, 0, 1);           /* init mutex */
...
for(i = 0; i < NITERS; i++) {
  sem_wait(&mutex);            /* protect shared */
  cnt++;                        /* variable cnt */
  sem_post(&mutex);
}
```

Producer-Consumer Model

- Producer and consumer threads share a bounded buffer, with *n* slots.
  - producer thread adds items to the buffer
  - consumer thread retrieves items from the buffer
- Must guarantee mutually-exclusive access to the buffer, and that the producer/consumer cannot access the buffer if it is full/empty.

```
typedef struct {
  int* buf;     /* Buffer array */
  int n;        /* Max # of slots */
  int front;    /* buf[(front+1)%n] is 1st item */
  int rear;     /* buf[rear%n] is last item */
  sem_t mutex; /* Protects accesses to buf */
  sem_t slots; /* Counts available slots */
  sem_t items; /* Counts available items */
} sbuf_t;
```

Other Concurrency Issues

- We've looked at techniques for mutual exclusion and producer-consumer synchronization, a small part of concurrent programming.
- Synchronization is a fundamentally difficult problem that raises issues that do not arise in sequential programs.
- What follows is a sample of the issues programmers must be aware of when writing concurrent programs.
- Presented in the context of threads, the issues exist whenever concurrent flows manipulate shared resources.

Thread Safety

- A function is ***thread-safe*** iff it always produces correct results when called repeatedly from multiple concurrent threads.
  - a function that is not thread-safe is called thread-unsafe
- Four (non-disjoint) classes of thread-unsafe functions:
  - Class 1: functions that do not protect shared variables
  - Class 2: functions that keep state across multiple invocations
  - Class 3: functions that return a pointer to a static variable
  - Class 4: functions that call thread-unsafe functions

Class 1: Shared Variables

```
/* thread-unsafe routine */
void* count(void* arg) {
   int i;
   for(i = 0; i < NITERS; i++)
      cnt++;
   return NULL;
}
```

- To make thread-safe, protect the shared variable with synchronization operations.
- ***Pro***: No changes in the calling program required.
- ***Con***: Synchronization operations will slow down the function even when called from single-threaded program

Class 2: Keeps State Across Calls

```
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;
}
```

- Calling `rand` repeatedly from a single thread is correct.
  - What can happen if it is called from multiple threads?
- To make thread-safe, we must rely on the caller to pass state information via arguments.
  - forces a change in the code of the calling routine
  - potentially 100s of call sites, a difficult and error-prone change

Class 3: Returns Pointer to Static

- Some functions compute a result in a local `static` variable and return a pointer to that variable.
  - results being used by one thread may be silently overwritten by another thread
- To make thread-safe, require the caller to pass the address of the variable in which to store the result.
  - removes shared variable, requires change in calling code
- Another option is the ***lock-and-copy*** technique.
  - associate a mutex with the thread-unsafe function
  - especially useful when the thread-unsafe function is impossible to modify (e.g., it is linked from a library)

Lock-and-Copy

- At each call site:
  - dynamically allocate memory for the result
  - lock the mutex
  - call the thread-unsafe function
  - copy the result returned by the function to this memory
  - unlock the mutex

```
struct hostent* gethostbyname_ts(char* hostname) {
      struct hostent *sharedp, *unsharedp;

      unsharedp = Malloc(sizeof(struct hostent)); /* dyn mem */
      P(&mutex);                                   /* lock mutex */
      sharedp = gethostbyname(hostname); /* thread-unsafe fn */
      *unsharedp = *sharedp;       /* copy to private struct */
      V(&mutex);                                 /* unlock mutex */
      return unsharedp;
}
```

Class 4: Calls Thread-Unsafe

- If function ***f*** calls thread-unsafe function ***g***, ***f*** may or may not also be thread-unsafe.

- If ***g*** keeps state across multiple invocations, then ***f*** is also thread-unsafe.
  - only solution is to rewrite ***g***

- If ***g*** does not protect shared variables or returns a pointer to a static variable, ***f*** may still be thread-safe.
  - solution is to protect call to ***g*** with a mutex (like previous example)

Reentrancy

- Reentrant functions do not reference any shared data when they are called by multiple threads.
- The set of reentrant functions is a proper subset of the thread-safe functions.
  - due to the lack of synchronization ops, reentrant functions are typically more efficient that non-reentrant thread-safe functions
- The only way to convert a Class 2 thread-unsafe function into a thread-safe one is to rewrite it to be reentrant.

```
/* rand_r - a reentrant pseudo-random integer generator */
int rand_r(unsigned int* nextp) {
   *nextp = *nextp * 1103515245 + 12345;
   return (unsigned int)(*nextp / 65536) % 32768;
}
```

Determining Reentrancy

- ***Explicitly reentrant***—all function arguments are passed by value and all data references are to local automatic stack variables.

- ***Implicitly reentrant***—allows some parameters in an otherwise explicitly-reentrant function to be pointers.
  - thus, it is a reentrant function only if the calling threads are careful to pass pointers to non-shared data
  - example: function `rand_r`

- Why is function `gethotstbyname_ts` thread-safe, but not reentrant?

Races

- A ***race*** occurs when the correctness of a program depends on one thread reaching point ***x*** in its control flow before another thread reaches point ***y***.

- Threaded programs must work correctly for any feasible trajectory.
  - Often programmers assume that threads will take a particular trajectory through the execution state space.

Deadlock

- A run-time error where a collection of threads are blocked, waiting for a condition that will never be true.
- The programmer has incorrectly ordered the semaphore ops.
- In state d, each program is waiting for the other to do a V op that won't occur.

Avoiding Deadlock

- Deadlock is difficult to predict in a program.
  - some trajectories will skirt the deadlock region
  - others will be trapped by it
- When semaphores are used for mutual exclusion, a simple rule can be applied.
  - A program is deadlock-free if, for each pair of mutexes (s, t) in the program, each thread that holds both s and t simultaneously locks them in the same order.
- In our example, lock s first then t, in each thread.

Summary

- A concurrent program consists of a collection of logical flows that overlap in time.
  - via processes—scheduled by the kernel, separate address space
  - via threads—scheduled by the kernel, shared address space
- P and V operations on semaphores help to synchronize concurrent accesses to shared data.
  - provides mutually exclusive access to shared data
  - schedules access to shared buffers in producer-consumer programs
- Difficult concurrency issues:
  - thread safety, reentrant functions, races, deadlocks