Review

- Multithreaded Programming
  - Race Conditions
  - Semaphores
  - Thread Safety, Deadlock, and Starvation

- Sockets and Asynchronous I/O
  - Sockets
  - Asynchronous I/O
Review: Multithreaded programming

- Thread: abstraction of parallel processing with shared memory
- Program organized to execute multiple threads in parallel
- Threads *spawned* by main thread, communicate via shared resources and *joining*
- *pthread* library implements multithreading
  - `int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void *(start_routine)(void *), void * arg);`
  - `void pthread_exit(void *value_ptr);`
  - `int pthread_join(pthread_t thread, void **value_ptr);`
  - `pthread_t pthread_self(void);`
Review: Resource sharing

- Access to shared resources need to be controlled to ensure deterministic operation
- Synchronization objects: mutexes, semaphores, read/write locks, barriers
- Mutex: simple single lock/unlock mechanism
  - `int` `pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
  - `int` `pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - `int` `pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int` `pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int` `pthread_mutex_unlock(pthread_mutex_t *mutex);`
Review: Condition variables

- Lock/unlock (with mutex) based on run-time condition variable
- Allows thread to wait for condition to be true
- Other thread signals waiting thread(s), unblocking them
  - `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
  - `int pthread_cond_destroy(pthread_cond_t *cond);`
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - `int pthread_cond_signal(pthread_cond_t *cond);`
Review

Multithreaded Programming
  - Race Conditions
  - Semaphores
  - Thread Safety, Deadlock, and Starvation

Sockets and Asynchronous I/O
  - Sockets
  - Asynchronous I/O
Multithreaded programming

- OS implements scheduler – determines which threads execute when
- Scheduling may execute threads in arbitrary order
- Without proper synchronization, code can execute non-deterministically
- Suppose we have two threads: 1 reads a variable, 2 modifies that variable
- Scheduler may execute 1, then 2, or 2 then 1
- Non-determinism creates a race condition – where the behavior/result depends on the order of execution
Race conditions

- Race conditions occur when multiple threads share a variable, without proper synchronization
- Synchronization uses special variables, like a mutex, to ensure order of execution is correct
- Example: thread $T_1$ needs to do something before thread $T_2$
  - condition variable forces thread $T_2$ to wait for thread $T_1$
  - producer-consumer model program
- Example: two threads both need to access a variable and modify it based on its value
  - surround access and modification with a mutex
  - mutex groups operations together to make them atomic – treated as one unit
Race conditions in assembly

Consider the following program `race.c`:

```c
unsigned int cnt = 0;

void *count(void *arg) { /* thread body */
    int i;
    for (i = 0; i < 100000000; i++)
        cnt++;
    return NULL;
}

int main(void) {
    pthread_t tids[4];
    int i;
    for (i = 0; i < 4; i++)
        pthread_create(&tids[i], NULL, count, NULL);
    for (i = 0; i < 4; i++)
        pthread_join(tids[i], NULL);
    printf("cnt=%u\n", cnt);
    return 0;
}
```

What is the value of `cnt`?

Race conditions in assembly

Ideally, should increment $\text{cnt} \times 4 \times 100000000$ times, so $\text{cnt} = 400000000$. However, running our code gives:

```
athena% ./race.o
cnt=137131900
athena% ./race.o
cnt=163688698
athena% ./race.o
cnt=163409296
athena% ./race.o
cnt=170865738
athena% ./race.o
cnt=169695163
```

So, what happened?

---

1 Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.
Race conditions in assembly

- C not designed for multithreading
- No notion of atomic operations in C
- Increment `cnt++;` maps to three assembly operations:
  1. load `cnt` into a register
  2. increment value in register
  3. save new register value as new `cnt`
- So what happens if thread interrupted in the middle?
- Race condition!
Let's fix our code:

```c
#include <pthread.h>

int main(void) {
    pthread_t tids[4];
    int i;
    pthread_mutex_init(&mutex, NULL);
    for (i = 0; i < 4; i++)
        pthread_create(&tids[i], NULL, count, NULL);
    for (i = 0; i < 4; i++)
        pthread_join(tids[i], NULL);
    pthread_mutex_destroy(&mutex);
    printf("cnt=%u\n", cnt);
    return 0;
}
```

```c
void *count(void *arg) { /* thread body */
    int i;
    for (i = 0; i < 100000000; i++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
pthread_mutex_t mutex;
unsigned int cnt = 0;
```
Race conditions

• Note that new code functions correctly, but is much slower
• C statements not atomic – threads may be interrupted at assembly level, in the middle of a C statement
• Atomic operations like mutex locking must be specified as atomic using special assembly instructions
• Ensure that all statements accessing/modifying shared variables are synchronized
Semaphores

- **Semaphore** – special nonnegative integer variable $s$, initially 1, which implements two atomic operations:
  - $P(s)$ – wait until $s > 0$, decrement $s$ and return
  - $V(s)$ – increment $s$ by 1, unblocking a waiting thread
- **Mutex** – locking calls $P(s)$ and unlocking calls $V(s)$
- **Implemented in <semaphore.h>, part of library rt, not pthread**
Using semaphores

- Initialize semaphore to value:
  ```c
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  ```
- Destroy semaphore:
  ```c
  int sem_destroy(sem_t *sem);
  ```
- Wait to lock, blocking:
  ```c
  int sem_wait(sem_t *sem);
  ```
- Try to lock, returning immediately (0 if now locked, −1 otherwise):
  ```c
  int sem_trywait(sem_t *sem);
  ```
- Increment semaphore, unblocking a waiting thread:
  ```c
  int sem_post(sem_t *sem);
  ```
Producer and consumer revisited

• Use a semaphore to track available slots in shared buffer
• Use a semaphore to track items in shared buffer
• Use a semaphore/mutex to make buffer operations synchronous
Producer and consumer revisited

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex, slots, items;

#define SLOTS 2
#define ITEMS 10

void* produce (void* arg)
{
    int i;
    for (i = 0; i < ITEMS; i++) {
        sem_wait(&slots);
        sem_wait(&mutex);
        printf("produced(%ld):%d\n", pthread_self(), i +1);
        sem_post(&mutex);
        sem_post(&items);
    }
    return NULL;
}

void* consume (void* arg)
{
    int i;

    sem_destroy(&mutex);

    return 0;
}

int main()
{
    pthread_t tcons, tpro;

    sem_init(&mutex, 0, 1);
    sem_init(&slots, 0, SLOTS);
    sem_init(&items, 0, 0);

    pthread_create(&tcons, NULL, consume, NULL);
    pthread_create(&tpro, NULL, produce, NULL);
    pthread_join(tcons, NULL);
    pthread_join(tpro, NULL);

    sem_destroy(&mutex);
    sem_destroy(&slots);
    sem_destroy(&items);
    return 0;
}
```

Other challenges

- Synchronization objects help solve race conditions
- Improper use can cause other problems
- Some common issues:
  - thread safety and reentrant functions
  - deadlock
  - starvation
Thread safety

- Function is *thread safe* if it always behaves correctly when called from multiple concurrent threads
- Unsafe functions fall in several categories:
  - accesses/modifies unsynchronized shared variables
  - functions that maintain state using static variables – like `rand()`, `strtok()`
  - functions that return pointers to static memory – like `gethostbyname()`
  - functions that call unsafe functions may be unsafe
Reentrant functions

- Reentrant function – does not reference any shared data when used by multiple threads
- All reentrant functions are thread-safe (are all thread-safe functions reentrant?)
- Reentrant versions of many unsafe C standard library functions exist:

<table>
<thead>
<tr>
<th>Unsafe function</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand()</td>
<td>rand_r()</td>
</tr>
<tr>
<td>strtok()</td>
<td>strtok_r()</td>
</tr>
<tr>
<td>asctime()</td>
<td>asctime_r()</td>
</tr>
<tr>
<td>ctime()</td>
<td>ctime_r()</td>
</tr>
<tr>
<td>gethostbyaddr()</td>
<td>gethostbyaddr_r()</td>
</tr>
<tr>
<td>gethostbyname()</td>
<td>gethostbyname_r()</td>
</tr>
<tr>
<td>inet_ntoa()</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime()</td>
<td>localtime_r()</td>
</tr>
</tbody>
</table>
Thread safety

To make your code thread-safe:

• Use synchronization objects around shared variables
• Use reentrant functions
• Use synchronization around functions returning pointers to shared memory (*lock-and-copy*):
  1. lock mutex for function
  2. call unsafe function
  3. dynamically allocate memory for result; (deep) copy result into new memory
  4. unlock mutex
Deadlock

- Deadlock – happens when every thread is waiting on another thread to unblock
- Usually caused by improper ordering of synchronization objects
- Tricky bug to locate and reproduce, since schedule-dependent
- Can visualize using a progress graph – traces progress of threads in terms of synchronization objects
Figure removed due to copyright restrictions. Please see http://csapp.cs.cmu.edu/public/1e/public/figures.html, Figure 13.39, Progress graph for a program that can deadlock.
Deadlock

- Defeating deadlock extremely difficult in general
- When using only mutexes, can use the “mutex lock ordering rule” to avoid deadlock scenarios:
  
  A program is deadlock-free if, for each pair of mutexes \((s, t)\) in the program, each thread that uses both \(s\) and \(t\) simultaneously locks them in the same order.

Starvation and priority inversion

• Starvation similar to deadlock
• Scheduler never allocates resources (e.g. CPU time) for a thread to complete its task
• Happens during priority inversion
  • example: highest priority thread $T_1$ waiting for low priority thread $T_2$ to finish using a resource, while thread $T_3$, which has higher priority than $T_2$, is allowed to run indefinitely
  • thread $T_1$ is considered to be in starvation
Review

Multithreaded Programming
- Race Conditions
- Semaphores
- Thread Safety, Deadlock, and Starvation

Sockets and Asynchronous I/O
- Sockets
- Asynchronous I/O
• *Socket* – abstraction to enable communication across a network in a manner similar to file I/O

• Uses header `<sys/socket.h>` (extension of C standard library)

• Network I/O, due to latency, usually implemented asynchronously, using multithreading

• Sockets use client/server model of establishing connections
Creating a socket

- Create a socket, getting the file descriptor for that socket:

  ```c
  int socket(int domain, int type, int protocol);
  ```

  - **domain** – use constant `AF_INET`, so we’re using the internet; might also use `AF_INET6` for IPv6 addresses
  - **type** – use constant `SOCK_STREAM` for connection-based protocols like TCP/IP; use `SOCK_DGRAM` for connectionless datagram protocols like UDP (we’ll concentrate on the former)
  - **protocol** – specify 0 to use default protocol for the socket type (e.g. TCP)
  - returns nonnegative integer for file descriptor, or −1 if couldn’t create socket

- Don’t forget to close the file descriptor when you’re done!
Connecting to a server

• Using created socket, we connect to server using:

```c
int connect(int fd, struct sockaddr *addr, int addr_len);
```

• `fd` – the socket’s file descriptor
• `addr` – the address and port of the server to connect to; for internet addresses, cast data of type `struct sockaddr_in`, which has the following members:
  • `sin_family` – address family; always `AF_INET`
  • `sin_port` – port in network byte order (use `htons()` to convert to network byte order)
  • `sin_addr.s_addr` – IP address in network byte order (use `htonl()` to convert to network byte order)
• `addr_len` – size of `sockaddr_in` structure
• returns 0 if successful
Associate server socket with a port

- Using created socket, we bind to the port using:

  ```c
  int bind(int fd, struct sockaddr *addr, int addr_len);
  ```

  - `fd, addr, addr_len` – same as for `connect()`
  - note that address should be IP address of desired interface (e.g. `eth0`) on local machine
  - ensure that port for server is not taken (or you may get “address already in use” errors)
  - return 0 if socket successfully bound to port
• Using the bound socket, start listening:
  ```c
  int listen (int fd, int backlog);
  ```
  • `fd` – bound socket file descriptor
  • `backlog` – length of queue for pending TCP/IP connections; normally set to a large number, like 1024
  • returns 0 if successful
Accepting a client’s connection

- Wait for a client’s connection request (may already be queued):

```c
int accept(int fd, struct sockaddr *addr, int *addr_len);
```

- **fd** – socket’s file descriptor
- **addr** – pointer to structure to be filled with client address info (can be NULL)
- **addr_len** – pointer to int that specifies length of structure pointed to by `addr`; on output, specifies the length of the stored address (stored address may be truncated if bigger than supplied structure)
- returns (nonnegative) file descriptor for connected client socket if successful
Reading and writing with sockets

- **Send data using the following functions:**
  ```c
  int write(int fd, const void *buf, size_t len);
  int send(int fd, const void *buf, size_t len, int flags);
  ```

- **Receive data using the following functions:**
  ```c
  int read(int fd, void *buf, size_t len);
  int recv(int fd, void *buf, size_t len, int flags);
  ```

- **fd** – socket’s file descriptor
- **buf** – buffer of data to read or write
- **len** – length of buffer in bytes
- **flags** – special flags; we’ll just use 0
- **all these return the number of bytes read/written (if successful)**
Asynchronous I/O

- Up to now, all I/O has been synchronous – functions do not return until operation has been performed
- Multithreading allows us to read/write a file or socket without blocking our main program code (just put I/O functions in a separate thread)
- Multiplexed I/O – use `select()` or `poll()` with multiple file descriptors
I/O multiplexing with `select()`

- To check if multiple files/sockets have data to read/write/etc: (include `<sys/select.h>`)  
  
  ```c
  int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
  ```
  
  - `nfds` – specifies the total range of file descriptors to be tested (0 up to `nfds-1`)
  - `readfds`, `writefds`, `errorfds` – if not NULL, pointer to set of file descriptors to be tested for being ready to read, write, or having an error; on output, set will contain a list of only those file descriptors that are ready
  - `timeout` – if no file descriptors are ready immediately, maximum time to wait for a file descriptor to be ready
  - returns the total number of set file descriptor bits in all the sets

- Note that `select()` is a blocking function
I/O multiplexing with `select()`

- `fd_set` – a mask for file descriptors; bits are set (“1”) if in the set, or unset (“0”) otherwise

Use the following functions to set up the structure:

- `FD_ZERO(&fdset)` – initialize the set to have bits unset for all file descriptors
- `FD_SET(fd, &fdset)` – set the bit for file descriptor `fd` in the set
- `FD_CLR(fd, &fdset)` – clear the bit for file descriptor `fd` in the set
- `FD_ISSET(fd, &fdset)` – returns nonzero if bit for file descriptor `fd` is set in the set
I/O multiplexing using `poll()`

- Similar to `select()`, but specifies file descriptors differently: (include `<poll.h>`)  
  ```c
  int poll(struct pollfd fds[], nfds_t nfds, int timeout);
  ```
  - `fds` – an array of `pollfd` structures, whose members `fd`, `events`, and `revents`, are the file descriptor, events to check (OR-ed combination of flags like `POLLIN`, `POLLOUT`, `POLLERR`, `POLLHUP`), and result of polling with that file descriptor for those events, respectively
  - `nfds` – number of structures in the array
  - `timeout` – number of milliseconds to wait; use 0 to return immediately, or −1 to block indefinitely
Summary

- Multithreaded programming
  - race conditions
  - semaphores
  - thread safety
  - deadlock and starvation

- Sockets, asynchronous I/O
  - client/server socket functions
  - `select()` and `poll()`
6.087 Practical Programming in C
January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.