Outline

- Review

- Multithreaded programming
  - Concepts

- Pthread
  - API
  - Mutex
  - Condition variables
Review

Multithreaded programming
- Concepts

Pthread
- API
- Mutex
- Condition variables
Review: malloc()

- **Mapping memory**: `mmap()`, `munmap()`. Useful for demand paging.
- **Resizing heap**: `sbrk()`
- **Designing malloc()**
  - implicit linked list, explicit linked list
  - best fit, first fit, next fit
- **Problems**:
  - fragmentation
  - memory leaks
  - `valgrind --tool=memcheck`, checks for memory leaks.
Garbage collection

- C does not have any garbage collectors
- Implementations available
- Types:
  - Mark and sweep garbage collector (depth first search)
  - Cheney’s algorithm (breadth first search)
  - Copying garbage collector
Review

Multithreaded programming
- Concepts

Pthread
- API
- Mutex
- Condition variables
Preliminaries: Parallel computing

- **Parallelism**: Multiple computations are done simultaneously.
  - Instruction level (pipelining)
  - Data parallelism (SIMD)
  - Task parallelism (embarrassingly parallel)
- **Concurrency**: Multiple computations that *may* be done in parallel.
- **Concurrency vs. Parallelism**
Process vs. Threads

- **Process**: An instance of a program that is being executed in its own address space. In POSIX systems, each process maintains its own heap, stack, registers, file descriptors etc.
  
  Communication:
  - Shared memory
  - Network
  - Pipes, Queues

- **Thread**: A light weight process that shares its address space with others. In POSIX systems, each thread maintains the bare essentials: registers, stack, signals.
  
  Communication:
  - shared address space.
Multithreaded concurrency

Serial execution:
- All our programs so far has had a single thread of execution: main thread.
- Program exits when the main thread exits.

Multithreaded:
- Program is organized as multiple and concurrent threads of execution.
- The main thread *spawns* multiple threads.
- The thread *may* communicate with one another.
- Advantages:
  - Improves performance
  - Improves responsiveness
  - Improves utilization
  - Less overhead compared to multiple processes
Even in C, multithread programming may be accomplished in several ways

- Pthreads: POSIX C library.
- OpenMP
- Intel threading building blocks
- Cilk (from CSAIL!)
- Grand central despatch
- CUDA (GPU)
- OpenCL (GPU/CPU)
Not all code can be made parallel

```c
float params[10];
for (int i = 0; i < 10; i++)
    do_something(params[i]);
```

```c
float params[10];
float prev = 0;
for (int i = 0; i < 10; i++)
{
    prev = complicated(params[i], prev);
}
```

| paralleizable | not parallelizable |
Not all multi-threaded code is safe

```c
int balance = 500;
void deposit(int sum) {
    int currbalance = balance; /* read balance */
    
    currbalance += sum;
    balance = currbalance; /* write balance */
}

void withdraw(int sum) {
    int currbalance = balance; /* read balance */
    if (currbalance > 0)
        currbalance -= sum;
    balance = currbalance; /* write balance */
}

.. deposit(100); /* thread 1 */
.. withdraw(50); /* thread 2 */
.. withdraw(100); /* thread 3 */
```

• minimize use of global/static memory
• Scenario: T1(read), T2(read, write), T1(write) , balance = 600
• Scenario: T2(read), T1(read, write), T2(write) , balance = 450
Review

Multithreaded programming
  Concepts

Pthread
  API
  Mutex
  Condition variables
Pthread

API:

• Thread management: creating, joining, attributes
  
  pthread_

• Mutexes: create, destroy mutexes
  
  pthread_mutex_

• Condition variables: create, destroy, wait, signal
  
  pthread_cond_

• Synchronization: read/write locks and barriers
  
  pthread_rwlock_, pthread_barrier_

API:

• \#include <pthread.h>

• gcc −Wall −O0 −o <output> file.c −pthread (no −l prefix)
Creating threads

```c
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

- creates a new thread with the attributes specified by `attr`.
- Default attributes are used if `attr` is `NULL`.
- On success, stores the thread it into `thread`
- calls function `start_routine(arg)` on a separate thread of execution.
- returns zero on success, non-zero on error.

```c
void pthread_exit(void *value_ptr);
```

- called implicitly when thread function exits.
- analogous to `exit()`.
```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

code: https://computing.llnl.gov/tutorials/pthreads/

© Lawrence Livermore National Laboratory. All rights reserved. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.
Output

In main: creating thread 0
In main: creating thread 1
Hello World! It’s me, thread #0!
Hello World! It’s me, thread #1!
In main: creating thread 2
In main: creating thread 3
Hello World! It’s me, thread #2!
Hello World! It’s me, thread #3!
In main: creating thread 4
Hello World! It’s me, thread #4!

In main: creating thread 0
Hello World! It’s me, thread #0!
In main: creating thread 1
Hello World! It’s me, thread #1!
In main: creating thread 2
Hello World! It’s me, thread #2!
In main: creating thread 3
Hello World! It’s me, thread #3!
In main: creating thread 4
Hello World! It’s me, thread #4!
Synchronization: joining

**Figure:** https://computing.llnl.gov/tutorials/pthreads

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

- `pthread_join()` blocks the calling thread until the specified thread terminates.
- If `value_ptr` is not null, it will contain the return status of the called thread

Other ways to synchronize: mutex, condition variables

© Lawrence Livermore National Laboratory. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [http://ocw.mit.edu/fairuse](http://ocw.mit.edu/fairuse).
Example

```c
#define NELEMENTS 5000
#define BLK_SIZE 1000
#define NTHREADS (NELEMENTS/BLK_SIZE)

int main ( int argc, char *argv[] )
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc; long t; void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr, work, (void *)(t*BLK_SIZE));
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
        }
    }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for (t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t], &status);
        if (rc) {
            printf("ERROR; return code from pthread_join() is %d\n", rc); exit(-1);
        }
    }
    printf("Main: program completed. Exiting.\n");
}```
Mutex

- Mutex (mutual exclusion) acts as a "lock" protecting access to the shared resource.
- Only one thread can "own" the mutex at a time. Threads must take turns to lock the mutex.

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
thread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `pthread_mutex_init()` initializes a mutex. If attributes are NULL, default attributes are used.
- The macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize static mutexes.
- `pthread_mutex_destroy()` destroys the mutex.
- Both function return return 0 on success, non zero on error.
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);
```

- `pthread_mutex_lock()` locks the given mutex. If the mutex is locked, the function is blocked until it becomes available.
- `pthread_mutex_trylock()` is the non-blocking version. If the mutex is currently locked the call will return immediately.
- `pthread_mutex_unlock()` unlocks the mutex.
Example revisited

```c
int balance=500;
void deposit(int sum){
    int currbalance=balance;/*read balance*/
    ...
    currbalance+=sum;
    balance=currbalance;/*write balance*/
}

void withdraw(int sum){
    int currbalance=balance;/*read balance*/
    if (currbalance >0)
        currbalance-=sum;
    balance=currbalance;/*write balance*/
}
.
.
deposit(100);/*thread 1*/
.
.withdraw(50);/thread 2*/
.
.withdraw(100);/*thread 3*/
.
```

- Scenario: T1(read),T2(read,write),T1(write),balance=600
- Scenario: T2(read),T1(read,write),T2(write),balance=450
Using mutex

```c
int balance=500;
pthread_mutex_t mutexbalance=PTHREAD_MUTEX_INITIALIZER;

void deposit(int sum){
    pthread_mutex_lock(&mutexbalance);
    {
        int currbalance=balance; /* read balance */
        ... 
        currbalance+=sum;
        balance=currbalance; /* write balance */
    }
    pthread_mutex_unlock(&mutexbalance);
}

void withdraw(int sum){
    pthread_mutex_lock(&mutexbalance);
    {
        int currbalance=balance; /* read balance */
        if (currbalance > 0)
            currbalance-=sum;
        balance=currbalance; /* write balance */
    }
    pthread_mutex_unlock(&mutexbalance);
}
```

```c
.. deposit(100); /* thread 1 */
.. withdraw(50); /* thread 2 */
.. withdraw(100); /* thread 3 */
```

- Scenario: T1(read,write), T2(read,write), balance=550
- Scenario: T2(read), T1(read,write), T2(write), balance=550
Condition variables

Sometimes locking or unlocking is based on a run-time condition (examples?). Without condition variables, program would have to poll the variable/condition continuously.

Consumer:

(a) lock mutex on global item variable
(b) wait for (item>0) signal from producer (mutex unlocked automatically).
(c) wake up when signalled (mutex locked again automatically), unlock mutex and proceed.

Producer:

(1) produce something
(2) Lock global item variable, update item
(3) signal waiting (threads)
(4) unlock mutex
Condition variables

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t cond, const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `pthread_cond_init()` initialized the condition variable. If `attr` is NULL, default attributes are set.
- `pthread_cond_destroy()` will destroy (uninitialize) the condition variable.
- destroying a condition variable upon which other threads are currently blocked results in undefined behavior.
- macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables. No error checks are performed.
- Both function return 0 on success and non-zero otherwise.
Condition variables

```c
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `pthread_cond_init()` initialized the condition variable. If `attr` is NULL, default attributes are used.
- `pthread_cond_destroy()` will destroy (uninitialize) the condition variable.
- Destroying a condition variable upon which other threads are currently blocked results in undefined behavior.
- `macro PTHREAD_COND_INITIALIZER` can be used to initialize condition variables. No error checks are performed.
- Both function return 0 on success and non-zero otherwise.
Condition variables

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- blocks on a condition variable.
- must be called with the mutex already locked otherwise behavior undefined.
- automatically releases mutex
- upon successful return, the mutex will be automatically locked again.

```c
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```c
int pthread_cond_signal(pthread_cond_t *cond);
```

- unblocks threads waiting on a condition variable.
- `pthread_cond_broadcast()` unlocks all threads that are waiting.
- `pthread_cond_signal()` unlocks one of the threads that are waiting.
- both return 0 on success, non zero otherwise.
Example

```c
#include <pthread.h>

pthread_cond_t cond_recv=PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_send=PTHREAD_COND_INITIALIZER;
pthread_mutex_t cond_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER;

int full = 0;
int count = 0;

void* produce(void*)
{
    while (1)
    {
        pthread_mutex_lock(&cond_mutex);
        while (full)
        {
            pthread_cond_wait(&cond_recv, &cond_mutex);
        }
        pthread_mutex_unlock(&cond_mutex);
        pthread_mutex_lock(&count_mutex);
        count ++; full = 1;
        printf("produced(%d):%d \n", pthread_self(), count);
        pthread_mutex_unlock(&count_mutex);
        pthread_mutex_unlock(&cond_mutex);
        printf(" consumed(%ld):%d \n", pthread_self(), count);
        pthread_mutex_unlock(&count_mutex);
        pthread_mutex_lock(&count_mutex);
        if (count >= 10) break;
    }
}

void* consume(void*)
{
    while (1)
    {
        pthread_mutex_lock(&cond_mutex);
        while (!full)
        {
            pthread_cond_wait(&cond_send, &cond_mutex);
        }
        pthread_mutex_unlock(&cond_mutex);
        pthread_mutex_lock(&count_mutex);
        count ++; full = 0;
        printf("consumed(%ld):%d \n", pthread_self(), count);
        pthread_mutex_unlock(&count_mutex);
        pthread_mutex_lock(&count_mutex);
        if (count >= 10) break;
    }
}
```
Example

```c
int main()
{
    pthread_t cons_thread, prod_thread;
    pthread_create(&prod_thread, NULL, produce, NULL);
    pthread_create(&cons_thread, NULL, consume, NULL);

    pthread_join(cons_thread, NULL);
    pthread_join(prod_thread, NULL);
    return 0;
}
```

Output:

```
produced(3077516144):1
consumed(3069123440):1
produced(3077516144):2
consumed(3069123440):2
produced(3077516144):3
consumed(3069123440):3
produced(3077516144):4
consumed(3069123440):4
produced(3077516144):5
consumed(3069123440):5
produced(3077516144):6
consumed(3069123440):6
produced(3077516144):7
consumed(3069123440):7
```
Summary

- Parallel programming concepts
- Multithreaded programming
- Pthreads
- Synchronization
- Mutex
- Condition variables