# Threads review

- Need to synchronize access to shared resources (critical sections)
- Busy wait
- Mutex
- Semaphores
- Barrier

# Condition variables

- Notify other threads
  - *notify_one (notify a single thread)*
  - *notify_all*
- Condition variables use mutex

# ATM example

- Use condition variables to do operations in the right order

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;

std::condition_variable cv;
std::mutex m;
long balance = 0;

void addMoney(int money){
    std::lock_guard<mutex> lg(m);
    balance += money;
    cout << "Amount added, current balance: " << balance << endl;
    cv.notify_one();
}

void withdrawMoney(int money){
    std::unique_lock<mutex> ul(m);
    cv.wait(ul, []{ return (balance != 0) ? true : false;});

    if(balance >= money){
        balance -= money;
        cout << "Amount deducted: " << money << endl;
    }
    else{
        cout << "Amount can't be deducted, current balance is less than " << money << endl;
    }
    cout << "Current balance: " << balance << endl;
}

int main(){
    std::thread t1(withdrawMoney, 500);
    std::thread t2(addMoney, 500);
    t1.join();
    t2.join();

    return 0;
}
```

# BARRIER USING CONDITION VARIABLES

```cpp
class Barrier
{
public:
  /**
   * @brief Construct a new Barrier object
   *
   * @param n number of threads to be synchronized
   */
  Barrier(const int n): n(n), count(n) {}

  /**
   * @brief blocks until all threads are waiting
   *
   */
  void wait()
  {
    std::unique_lock<std::mutex> lk(m);
    --count;
    if (count != 0)
    {
      cv.wait(lk);
    }
    else
    {
      cv.notify_all();
      count = n;
    }
  }

private:
  const int n;
  int count;
  std::condition_variable cv;
```

```cpp
class Barrier
{
public:
  /**
   * @brief Construct a new Barrier object
   *
   * @param n number of threads to be synchronized
   */
  explicit Barrier(const int n): n(n), count(n), generation(0) {}

  /**
   * @brief blocks until all threads have waited
   *
   */
  void wait()
  {
    std::unique_lock<std::mutex> lk(m);
    const int last_gen = generation;
    if (--count == 0)
    {
      ++generation;
      count = n;
      cv.notify_all();
    }
    else
    {
      cv.wait(lk, [this, last_gen]{ return last_gen != generation; });
    }
  }

private:
  const int n;
  int count;
  int generation;
  std::condition_variable cv;
  std::mutex m;
};
```

# Barrier using condition variables (2)

Preventing spurious wake up

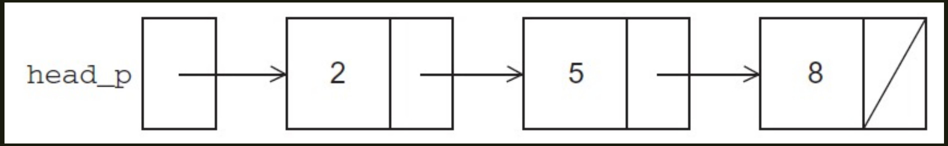
# READ-WRITE LOCKS

# Controlling access to a large, shared data structure

- Let’s look at an example.

- Suppose the shared data structure is a sorted linked list of ints, and the operations of interest are Member, Insert, and Delete.

# LINKED LISTS

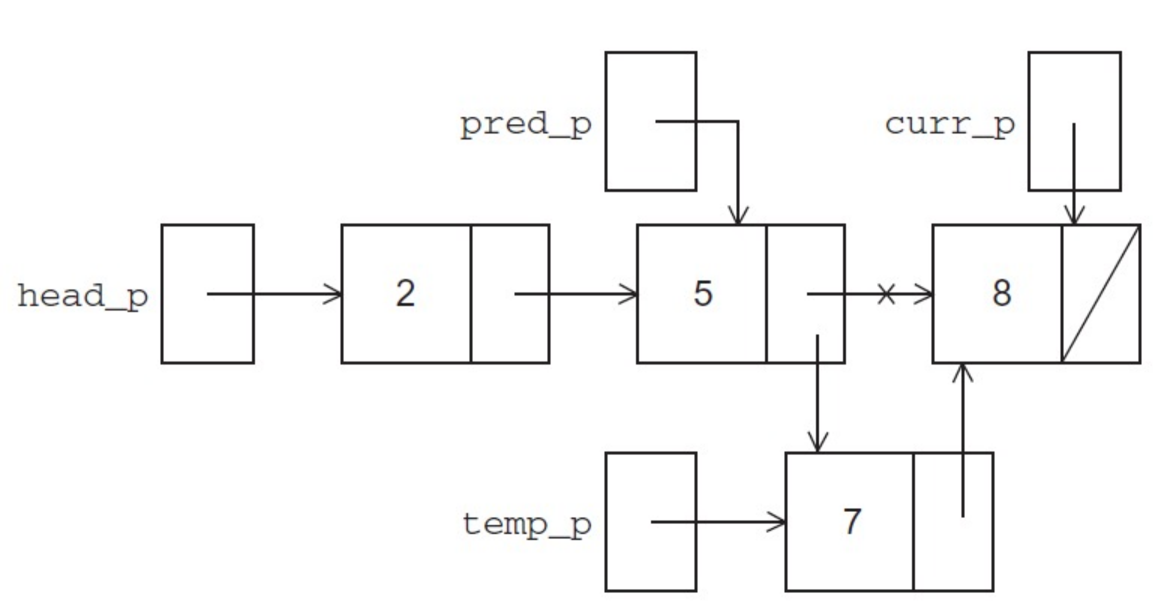



```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```


# LINKED LIST MEMBERSHIP

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

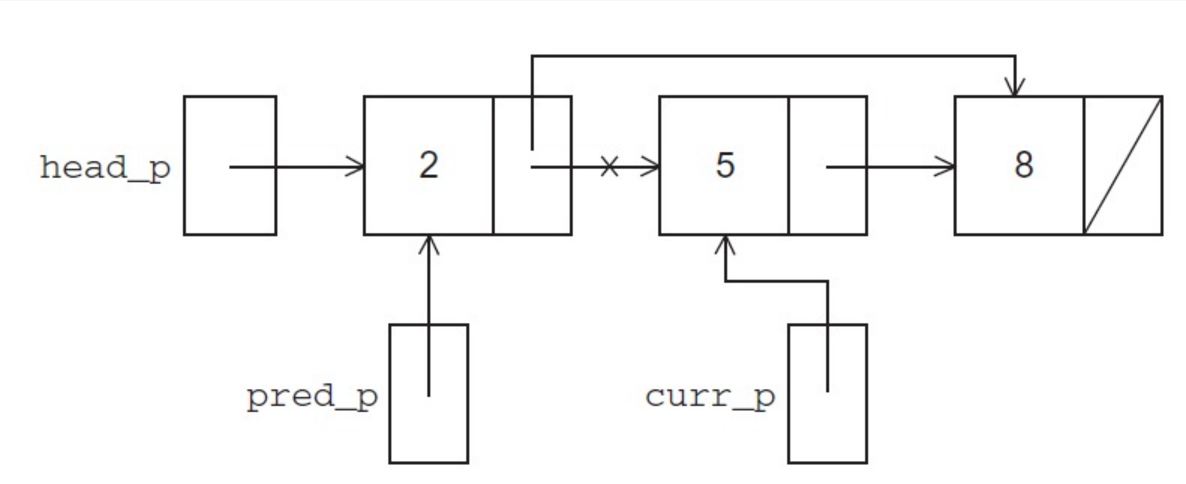
# Inserting a new node into a list



# Inserting a new node into a list

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

# Deleting a node from a linked list



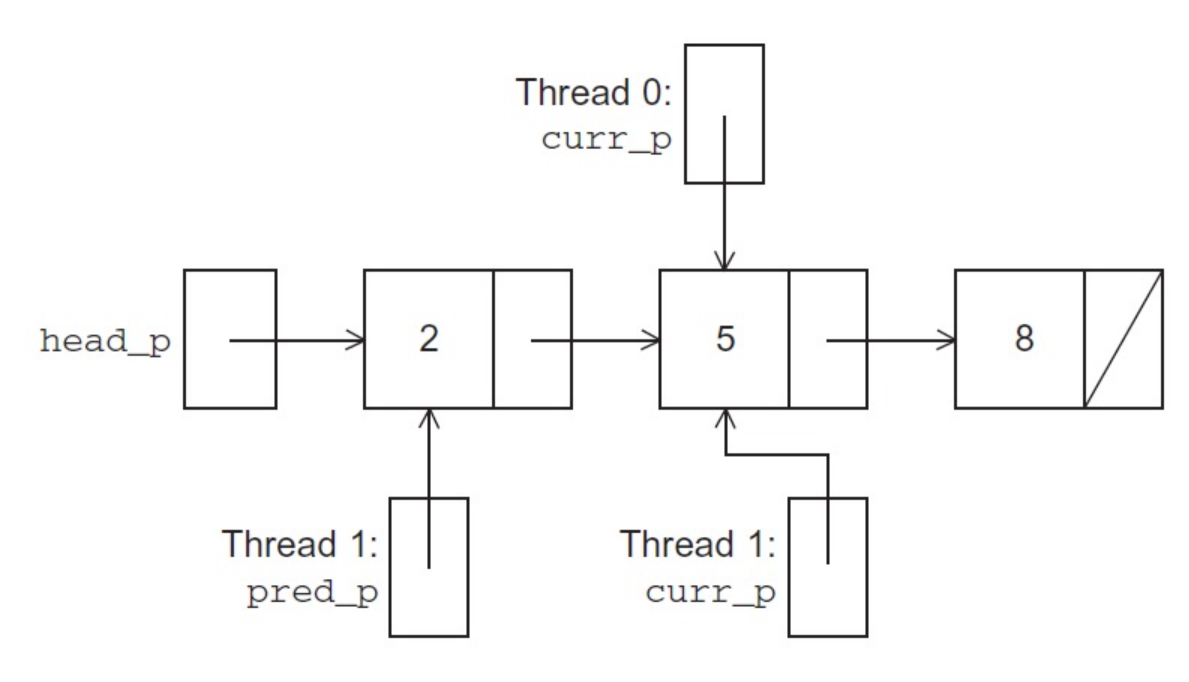
# Deleting a node from a linked list

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

# A Multi-Threaded Linked List

- Let's try to use these functions in a Pthreads program.
- In order to share access to the list, we can define head_p to be a global variable.
- This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p: we'll only need to pass in the value of interest.

# Simultaneous access by two threads



# Solution #1

- An obvious solution is to simply lock the list any time that a thread attempts to access it.
- A call to each of the three functions can be protected by a mutex.

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

In place of calling Member(value).

# Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

# Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A “finer-grained” approach.

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

# Issues

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.

# Implementation of Member with one mutex per list node (1)

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }
```

# Implementation of Member with one mutex per list node (2)

```
   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

# Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

# Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
- The first lock function locks the read-write lock for reading, while the second locks it for writing.

# Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.

- Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.

# Pthreads Read-Write Locks

- If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

# Protecting our linked list functions

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

# Linked List Performance

| Implementation | Number of Threads 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Read-Write Locks | 0.213 | 0.123 | 0.098 | 0.115 |
| One Mutex for Entire List | 0.211 | 0.450 | 0.385 | 0.457 |
| One Mutex per Node | 1.680 | 5.700 | 3.450 | 2.700 |

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

# Linked List Performance

| Implementation | Number of Threads 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Read-Write Locks | 2.48 | 4.97 | 4.69 | 4.71 |
| One Mutex for Entire List | 2.50 | 5.13 | 5.04 | 5.11 |
| One Mutex per Node | 12.00 | 29.60 | 17.00 | 12.00 |

100,000 ops/thread

80% Member

10% Insert

10% Delete

# THREAD-SAFETY

# Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.

# Example

- Suppose we want to use multiple threads to “tokenize” a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

# Simple approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.
- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.

# Simple approach

- We can serialize access to the lines of input using semaphores.
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

# The strtok function

- The first time it’s called the string argument should be the text to be tokenized.
  - *Our line of input.*
- For subsequent calls, the first argument should be NULL.

```
char* strtok(
      char*          string      /* in/out */,
      const char*    separators  /* in     */);
```

# The strtok function

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens <u>taken from the cached copy</u>.

# Multi-threaded tokenizer (1)

```
void *Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) % thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);
```

# Multi-threaded tokenizer (2)

```
      count = 0;
      my_string = strtok(my_line, " \t\n");
      while ( my_string != NULL ) {
         count++;
         printf("Thread %ld > string %d = %s\n", my_rank, count,
               my_string);
         my_string = strtok(NULL, " \t\n");
      }

      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }

   return NULL;
}  /* Tokenize */
```

# Running with one thread

- It correctly tokenizes the input stream.

Pease porridge hot.

Pease porridge cold.

Pease porridge in the pot

Nine days old.

# Running with two threads

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

Oops!

# What happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.

- Unfortunately for us, this cached string is shared, not private.

# What happened?

- Thus, thread 0's call to strtok with the third line of the input has apparently <u>overwritten</u> the contents of thread 1's call with the second line.
- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.

# Other unsafe C library functions

- Regrettably, it’s not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.

# “re-entrant” (thread safe) functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```
char*  strtok_r(
       char*          string        /* in/out */,
       const char*    separators,   /* in     */
       char**         saveptr_p     /* in/out */);
```

- A function is reentrant when
  - *Does not hold any static or global non-const data*
  - *It does not modify itself*
  - *Does not use non-reentrant code*

# Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.

# Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.

# Concluding Remarks (3)

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.
- So the execution of code in a critical section should, effectively, be executed as serial code.

# Concluding Remarks (4)

- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.

# Concluding Remarks (5)

- A mutex can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

# Concluding Remarks (6)

- A semaphore is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: sem_wait and sem_post.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.

# Concluding Remarks (7)

- A barrier is a point in a program at which the threads block until all of the threads have reached it.
- A read-write lock is used when it’s safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

# Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not thread-safe.