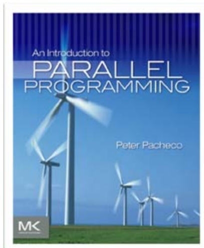



# Chapter 4

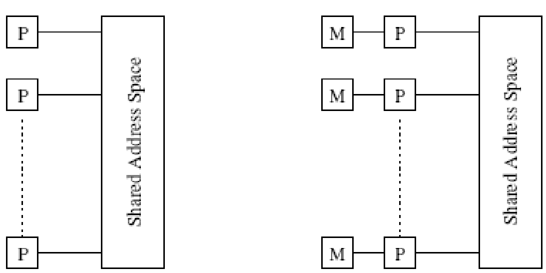
## Shared Memory Programming with Pthreads




## P threads (POSIX)

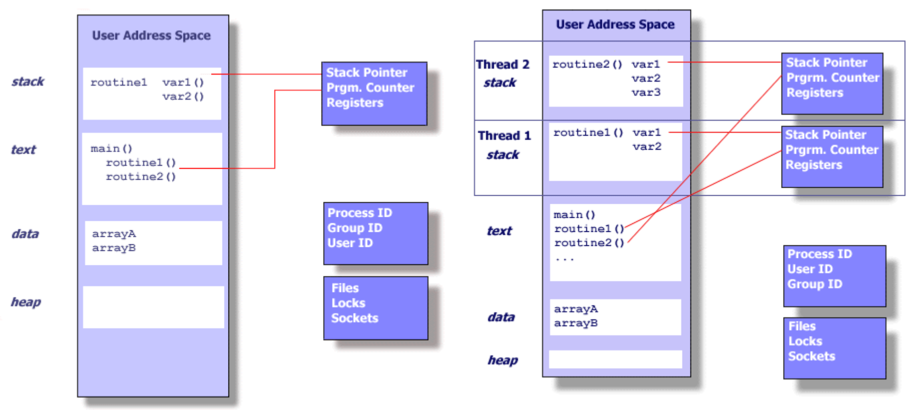
**(see https://computing.llnl.gov/tutorials/pthreads/)**

- Threads provide support for expressing concurrency and synchronization.
- Can be used for hiding memory latency
- A thread is a light weight process (has its own stack and execution state, but shares the address space with its parent).
- Hence, threads have local data but also can share global data.



# P threads (POSIX)

**(see https://computing.llnl.gov/tutorials/pthreads/)**




3

# The Pthread API

- Pthreads has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Provides two basic functions for specifying concurrency:

```
#include <pthread.h>
```

← declares the various Pthreads functions, constants, types, etc.

```
int pthread_create (pthread_t *  thread_p,
                    const pthread_attr_t *  attribute,
                    void (*thread_function)(void *),
                   void *arg_p);
```

← Create a thread identified by a thread handle

```
int pthread_join (pthread_t  thread_p,
                  void * *ptr);
```

← Wait for  the thread associated with thread_p to complete

4

## A closer look

```
int pthread_create (
    pthread_t*  thread_p /* out */ ,
    const pthread_attr_t*  attr_p /* in */ ,
    void*  (*thread_function ) ( void ) /* in */ ,
    void*  arg_p /* in */ ) ;
```

Pointer to the argument that should be passed to the function *thread_function*.

The function that the thread is to run.

We won't be using, so we just pass NULL.

Allocate handle before calling.





## Notes

- Data members of pthread_t objects aren't directly accessible to user code.
- However, a pthread_t object stores enough information to uniquely identify the thread with which it's associated.
- Variables declared within the thread function are local to the thread
- Variable declared outside the thread function are shared by all threads

## Hello World! (1)- Sec. 4.2

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

 7

## Hello World! (2)

```
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */

void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

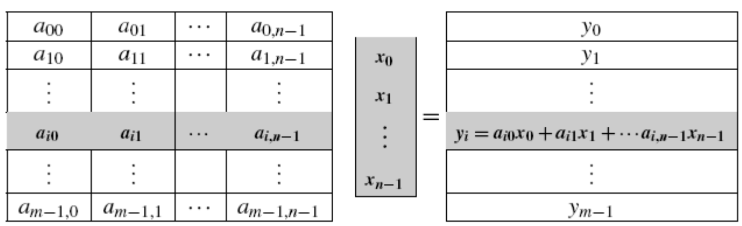
 8

# Function started by pthread_create

- void* thread_function ( void* args_p ) ;
  - void* can be cast to any pointer type in C.
  - args_p can point to a list containing the argument to thread_function.
  - the return value of thread_function can point to a list of one or more values.




# Matrix/vector multiplication (Sec. 4.3)




```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

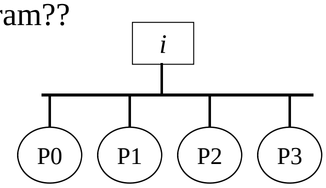
# Pthreads matrix-vector multiplication

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```




# Synchronization (race conditions – Sec. 4.4)

What is the output of the following program??




*dp = 0 ;*
*for (id = 0; id < 4; id++)*
*create_thread (…, sum_computed_values, …);*

*void sum_computed_values ( );*
*{ pdp = compute the value ( ) ;*
*dp += pdp ;*
*}*

Read *dp* from memory
Add 1 to *dp*
Write *dp* to memory

- A **critical section** is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- ***locks*** can be used to enforce mutual exclusion

get the lock ;
*dp += pdp* ;
release the lock ;

Most parallel languages provides ways to declare and use locks or critical sections

# Mutual Exclusion

- Critical sections in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical section first tries to get a lock. It goes ahead when the lock is granted.

- The API provides the following functions for handling mutex-locks:

int *pthread_mutex_init* ( pthread_mutex_t  *mutex_lock,
const pthread_mutexattr_t *lock_attr);

int *pthread_mutex_lock* ( pthread_mutex_t *mutex_lock);

int *pthread_mutex_unlock* (pthread_mutex_t *mutex_lock);

13

# Types of Mutexes

- Pthreads supports three types of mutexes.
  - A **normal mutex** deadlocks if a thread that already has a lock tries a second lock on it.
  - A **recursive mutex** allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
  - An **error check mutex** reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).

- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

14

# Attributes Objects for Mutexes

- Initialize the attrributes object using function:
  *pthread_mutexattr_init* (pthread_mutexattr_t  *attr) ;

- The function pthread_mutexattr_settype_np can be used for setting the type of mutex specified by the mutex attributes object.
  *pthread_mutexattr_settype_np* ( pthread_mutexattr_t *attr, int type);

- Here, type specifies the type of the mutex and can take one of:
  – PTHREAD_MUTEX_NORMAL_NP
  – PTHREAD_MUTEX_RECURSIVE_NP
  – PTHREAD_MUTEX_ERRORCHECK_NP

15

## Controlling Thread Attributes

- In general, the Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

- Use pthread_attr_init to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:
  - ❖ pthread_attr_setdetachstate,
  - ❖ pthread_attr_setguardsize_np,
  - ❖ pthread_attr_setstacksize,
  - ❖ pthread_attr_setinheritsched,
  - ❖ pthread_attr_setschedpolicy,
  - ❖ pthread_attr_setschedparam

16

## Overheads of Locking

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using

  int *pthread_mutex_trylock* (pthread_mutex_t *mutex_lock);

  which attempts to lock mutex_lock, but if unsuccessful, will return immediately with a “busy” error code.

17

## Busy-Waiting (sec. 4.5)

- Synchronization by enforcing order
- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
- Beware of optimizing compilers, though!

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread



18

## An example: Estimating π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```





## A thread function to compute π

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
} /* Thread_sum */
```

## Computing π using busy-waiting

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```




## Inefficient computation of π

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

| Threads | Busy-Wait | Mutex |
|---|---|---|
| 1 | 2.90 | 2.90 |
| 2 | 1.45 | 1.45 |
| 4 | 0.73 | 0.73 |
| 8 | 0.38 | 0.38 |
| 16 | 0.50 | 0.38 |
| 32 | 0.80 | 0.40 |
| 64 | 3.56 | 0.38 |

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.

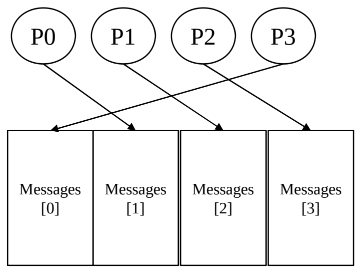



| Time | flag | Thread | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | crit sect | busy wait | susp | susp | susp |
| 1 | 1 | terminate | crit sect | susp | busy wait | susp |
| 2 | 2 | — | terminate | susp | busy wait | busy wait |
| ⋮ | ⋮ | | | ⋮ | ⋮ | ⋮ |
| ? | 2 | — | — | crit sect | susp | busy wait |

Possible sequence of events with busy-waiting and more threads than cores (5 threads and two cores).

# Notes

- Busy-waiting orders  the accesses of threads to a critical section.
- Using mutexes, the order is left to chance and the system.
- There are applications where we need to control the order of thread access to the critical section. For example:
  – Any non-commutative operation, such as matrix multiplication.
  – Emulating message passing on shared memory systems.






## A first attempt at sending messages using pthreads

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                 */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

# Semaphores (sec 4.7)

- Unsigned integers that count up to a given value, MAX (if MAX=1, then it is a binary semaphore)
- A semaphore is available if its value is larger than zero, otherwise, it is unavailable.
- When a thread "waits" on a semaphore: The count is decremented if it is larger than 0. Otherwise the thread blocks until the count becomes larger than 0.
- When a thread "posts" to a semaphore: The count is incremented (up to MAX).



## Syntax of the various semaphore functions

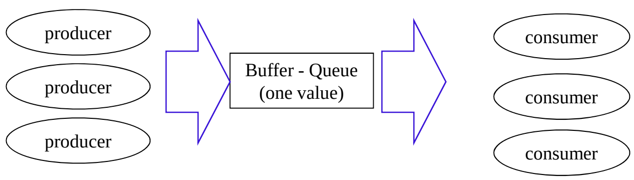
Semaphores are not part of Pthreads; you need to add this.

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p    /* out */,
      int        shared         /* in  */,
      unsigned   initial_val    /* in  */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

## Producer-Consumer Using Locks

The producer-consumer scenario imposes the following constraints:

- The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads must not pick up tasks until there is something present in the shared data structure.
- Individual consumer threads should pick up tasks one at a time.




We can start with the assumption that there is one producer and one consumer

29

## Producer-Consumer implementation

```
/* Producer */
while (!done()) {
    create_task (&my_task);                      lock
    while (task_available  == 1)  ;   /* wait until buffer is empty */
    insert_into_queue(&my_task);      /* put task in buffer */
    task_available = 1;                /* indicate that buffer is occupied*/
}                                               unlock

/* Consumer */
while (!done()) {                                 lock
    while (task_available  == 0)  ;     /* wait until buffer is full */
    extract_from_queue(&my_task); /* consume task from buffer */
    task_available = 0;                 /* indicate that buffer is not occupied */
    process_task(&my_task) ;                     unlock
}
```

**What is wrong with this implementation?**

30

## A fix for the Producer-Consumer program

```
/* Producer */
while (!done()) {
      create_task (&my_task);
      inserted = 0 ;              /* to flag successful insertion */
      while (inserted == 0)  {
          pthread_mutex_lock (&task_queue_lock);
          if (task_available == 0) {
             insert_into_queue(my_task);
             task_available = 1;
             inserted = 1; }
          pthread_mutex_unlock(&task_queue_lock);
}

/* Consumer */
while (!done()) {
    extracted = 0;                /* to flag success extraction */
    while (extracted == 0) {
        pthread_mutex_lock (&task_queue_lock);
        if (task_available == 1) {
            extract_from_queue(&my_task);
            task_available = 0;
            extracted = 1 ;}
        pthread_mutex_unlock(&task_queue_lock);
    }
    process_task(&my_task);
}
```



## Producer-Consumer Using Locks

```
pthread_mutex_t  task_queue_lock;
int  task_available;
...
main() {
        ....
        task_available = 0;
        pthread_mutex_init (&task_queue_lock, NULL);
        Create producer threads and consumer threads ;
}
void *producer (void *producer_thread_data) {
        int inserted;
        struct task my_task;
        while (!done()) {
                create_task (&my_task);    /* a procedure that produces a data structure for a new task */
                inserted = 0;
                while (inserted == 0) {
                        pthread_mutex_lock (&task_queue_lock);
                        if (task_available == 0) {              /* if buffer is empty */
                                insert_into_queue(&my_task);     /* put task in buffer */
                                task_available = 1;             /* indicate that buffer is occupied */
                                inserted = 1;                   /* and that insertion is successful */
                        }
                        pthread_mutex_unlock(&task_queue_lock);
                }
        }
}
```

## Producer-Consumer Using Locks

```
void *consumer (void *consumer_thread_data) {
    int extracted;
    struct task my_task;

    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock (&task_queue_lock);
            if (task_available == 1) {              /* if buffer is not empty */
                extract_from_queue(&my_task); /* get task from buffer */
                task_available = 0;               /* indicate that buffer is empty */
                extracted = 1;                    /* and that task is consumed */
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);                /* a procedure to process the task */
    }
}
```

Modify the above code to replace the buffer by a FIFO queue of length L.



# BARRIERS AND CONDITION VARIABLES (SEC. 4.8)

## Condition Variables for Synchronization (signals)

- A condition variable allows a thread to block itself until a specified condition becomes true.
- When a thread executes

  pthread_cond_wait(condition_variable),

  it is blocked until another thread executes

  pthread_cond_signal(condition_variable) or

  pthread_cond_broadcast(condition_variable)
- pthread_cond_signal () is used to unblock one of the threads blocked waiting on the condition_variable.
- pthread_cond_broadcast() is used to unblock all the threads blocked waiting on the condition_variable.
- If no threads are waiting on the condition variable, then a pthread-cond_signal() or pthread-cond_broadcast() will have no effect.



## Condition Variables for Synchronization

- A condition variable always has a mutex associated with it. A thread locks this mutex before issuing a wait, a signal or a broadcast.
- While the thread is waiting on a condition variable, the mutex is automatically unlocked, and when the thread is signaled, the mutex is automatically locked again.
- Pthreads provides the following functions for condition variables

  int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);

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

  int pthread_cond_signal (pthread_cond_t *cond);

  int pthread_cond_broadcast (pthread_cond_t *cond);

  int pthread_cond_destroy (pthread_cond_t *cond);

## Typical use of condition variables

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

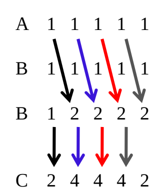
We will see an example later.

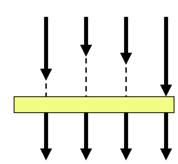
37

## Synchronization (barriers)

What is the output of the following Pthread program??

int main(int argc, char *argv) {
double *A*[5] , *B*[5], *C*[5] ; /* global, shared variables*/
for (*i* = 0; *i* < 5 ; *i*++) *A*[*i*] = *B*[*i*] = *1* ;
for (*i* = 0; *i* < 4 ; *i*++) pthread_create( … , DoStuff, int *i* ) ;
…
for (*i* = 0; *i* < 4 ; *i*++) pthread_join (… , DoStuff, …) ;
Print the values of *C* ;
}
void DoStuff (int *threadID*) {
int *k* ;
*B*[*threadID+1*] = 2 * *A*[*threadID*] ;
Barrier
*C*[*threadID*] = 2 * *B*[*threadID*] ;
}



38

## Barriers - a composite synchronization construct

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss one such constructs - barriers.

- A barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer (counter) is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast and resets the count to zero (to prepare for the next barrier).

39

## Defining your own barrier construct

```
typedef struct {
        pthread_mutex_t  count_lock;
        pthread_cond_t  ok_to_proceed;
        int count;
} mylib_barrier_t;
void mylib_init_barrier (mylib_barrier_t *b) {
        b.count = 0;
        pthread_mutex_init (&(b.count_lock), NULL);
        pthread_cond_init (&(b.ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t  *b,  int thread_count) {
pthread_mutex_lock (&(b.count_lock));
b.count ++;
if (b.count == thread_count) {
      b.count = 0;
      pthread_cond_broadcast (&(b.ok_to_proceed));
      }
else
     pthread_cond_wait (&(b.ok_to_proceed), &(b.count_lock)) ;
pthread_mutex_unlock(&(b.count_lock));
}
```

40

## Using the defined barrier

```
mylib_barrier_t  my_barrier ;                    /*declare a barrier */

int main(int argc, char *argv) {
mylib_init_barrier (my_barrier) ;               /* initialize the barrier */
double A[5] , B[5], C[5] ;    /* global, shared variables*/
for (i = 0; i < 5 ; i++)  A[i] = B[i] = 1 ;
for (i = 0; i < 4 ; i++)  pthread_create( … , DoStuff, int i ) ;
for (i = 0; i < 4 ; i++)  pthread_join (… , DoStuff, …) ;
Print the values of C ;
}

void DoStuff (int threadID) {
int  k ;
B[threadID+1] = 2 * A[threadID] ;
mylib_barrier ( my_barrier, 4) ;                /* call the barrier */
C[threadID] = 2 * B[threadID] ;
}
```

41

## Implementing barriers using busy waiting

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

42

## Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```

Used as a lock to protect the counter.



43

## Producer-Consumer Using Condition Variables

```
pthread_cond_t  queue_not_full,  queue_not_empty;
pthread_mutex_t  queue_cond_lock;
int task_available;
/* other data structures here */
main() {
        /* declarations and initializations */
        task_available = 0;
        pthread_init();
        pthread_cond_init(&queue_not_empty, NULL);
        pthread_cond_init(&queue_not_full, NULL);
        pthread_mutex_init(&queue_cond_lock, NULL);
        /* create and join producer and consumer threads */
}
```

Use default attributes

- If the buffer is full, a producer thread will wait on condition “queue not full”. That is, wait until a consumer thread will remove an item from the buffer and signal “queue not full”.
- If a buffer is empty, a consumer thread will wait on condition “queue not empty”. That is, wait until a producer thread will put an item in the buffer and signal “queue not empty”

44

## (assuming infinite buffer)

```
void *producer (void *producer_thread_data) {
        …
        while (!done()) {
                create_task();
                pthread_mutex_lock (&queue_cond_lock);
                insert_into_queue();
                task_available ++ ;
                pthread_cond_signal (&queue_not_empty);
                pthread_mutex_unlock (&queue_cond_lock);
        }
}

void *consumer (void *consumer_thread_data) {
        …
        while (!done()) {
                pthread_mutex_lock(&queue_cond_lock);
                if (task_available == 0)            Buffer is empty
                        pthread_cond_wait (&queue_not_empty,  &queue_cond_lock);
                my_task = extract_from_queue();
                task_available -- ;
                pthread_mutex_unlock (&queue_cond_lock);
                process_task(my_task);
        }
}
```

45

## Assuming a buffer with one entry

```
void *producer (void *producer_thread_data) {
        …
        while (!done()) {
                create_task();
                pthread_mutex_lock (&queue_cond_lock);
                if (task_available == 1)            Buffer is full
                        pthread_cond_wait (&queue_not_full,  &queue_cond_lock);
                insert_into_queue();
                task_available = 1;
                pthread_cond_signal (&queue_not_empty);
                pthread_mutex_unlock (&queue_cond_lock);
        }
}
void *consumer (void *consumer_thread_data) {
        …
        while (!done()) {
                pthread_mutex_lock(&queue_cond_lock);
                if (task_available == 0)            Buffer is empty
                        pthread_cond_wait (&queue_not_empty,  &queue_cond_lock);
                my_task = extract_from_queue();
                task_available = 0;
                pthread_cond_signal (&queue_not_full);
                pthread_mutex_unlock (&queue_cond_lock);
                process_task(my_task);
        }
}
```

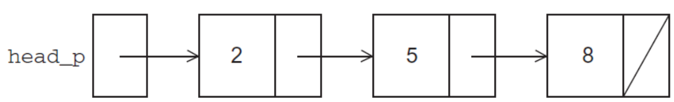
46

# READ-WRITE LOCKS (SEC. 4.9)

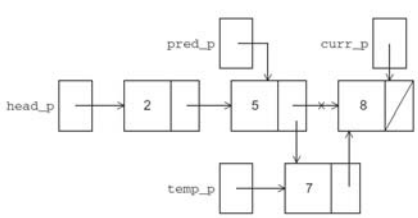




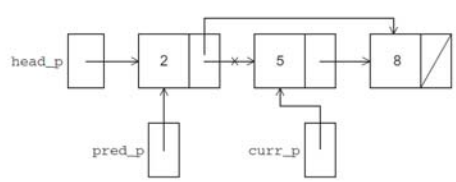
## Controlling access shared data structures




A linked list

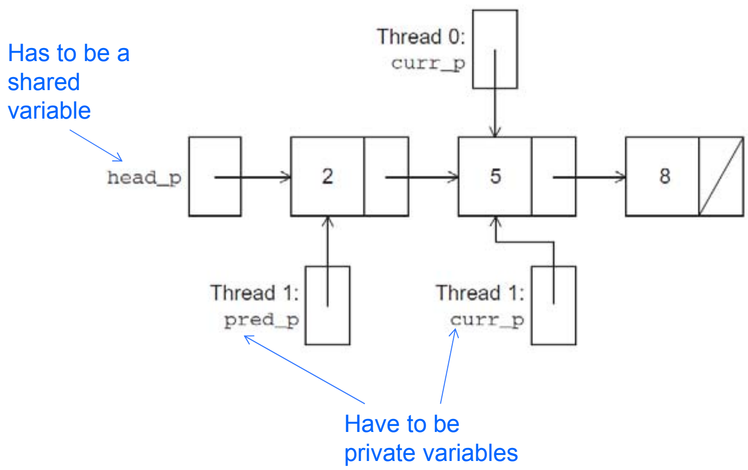



Inserting a new node




deleting a node

## 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 (use Mutex).
- Drawbacks:
  - 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;
}
```

- 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.




```
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;
   }
   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.
- 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 thread owns the lock for reading, any thread that wants to obtain the lock for writing will block in the call to the write-lock function.
- 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.

## Syntax of read-write locks

```
int pthread_rwlock_rdlock ( pthread_rwlock_t*  rwlock_p) ;
int pthread_rwlock_wrlock ( pthread_rwlock_t*  rwlock_p) ;
int pthread_rwlock_unlock ( pthread_rwlock_t*  rwlock_p) ;


int pthread_rwlock_init ( pthread_rwlock_t*  rwlock_p ,
                          pthread_rwlock_attr_t*  attr_p) ;
int pthread_rwlock_udestroy ( pthread_rwlock_t*  rwlock_p) ;
```




## 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

| 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 (Sec. 4.11)

- A block of code is thread-safe if it can be simultaneously executed by multip 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.

 57

# 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 $t^{th}$ goes to thread t, the t +1st goes to thread 0, etc.
- 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.

 58

# 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 idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.




# Multi-threaded tokenizer

```
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);
      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 two threads

Input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

Output

```
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.
- 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.

## Re-entrent functions.

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

- Some functions are provided with a safe thread capability – these are called re-entrent functions
- `For example, can use rand_r()` rather than rand()
- Need to compile with
  "gcc -D_REENTRANT –lpthread program.c"




## 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 process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
- When indeterminacy results from multiple threads attempting to access a shared resource such that at least one of the accesses is an update, we have a race condition.
- 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 execution in a critical section is effectively serialized.

## Concluding Remarks (2)

- Busy-waiting can be used to avoid conflicting access to critical sections (can be very wasteful of CPU cycles).
- Can also be unreliable if compiler optimization is turned on.
- 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.
- A semaphore is the third way to enforce 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 (3)

- 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.
- 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.