北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Chapter 4
Shared Memory Programming with Pthreads

软件学院　邵兵

2022/3/29

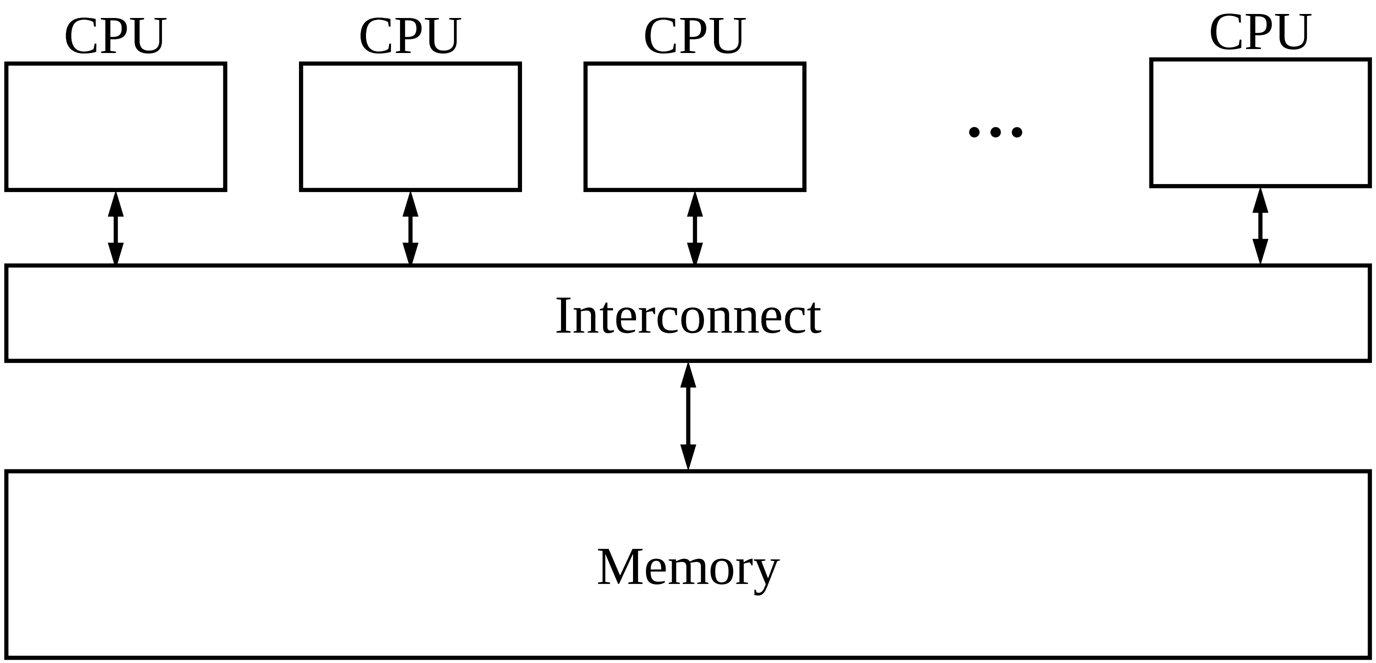
Contents

1. **Processes, threads and Pthreads**
2. **Critical sections**
   - **Busy-Waiting**
   - **Mutexes**
   - **semaphores**
3. **Barriers and condition variables**
4. **Read-write locks**
5. **Cache-Coherence & Thread-Safety**

1. PROCESSES, THREADS AND PTHREADS

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

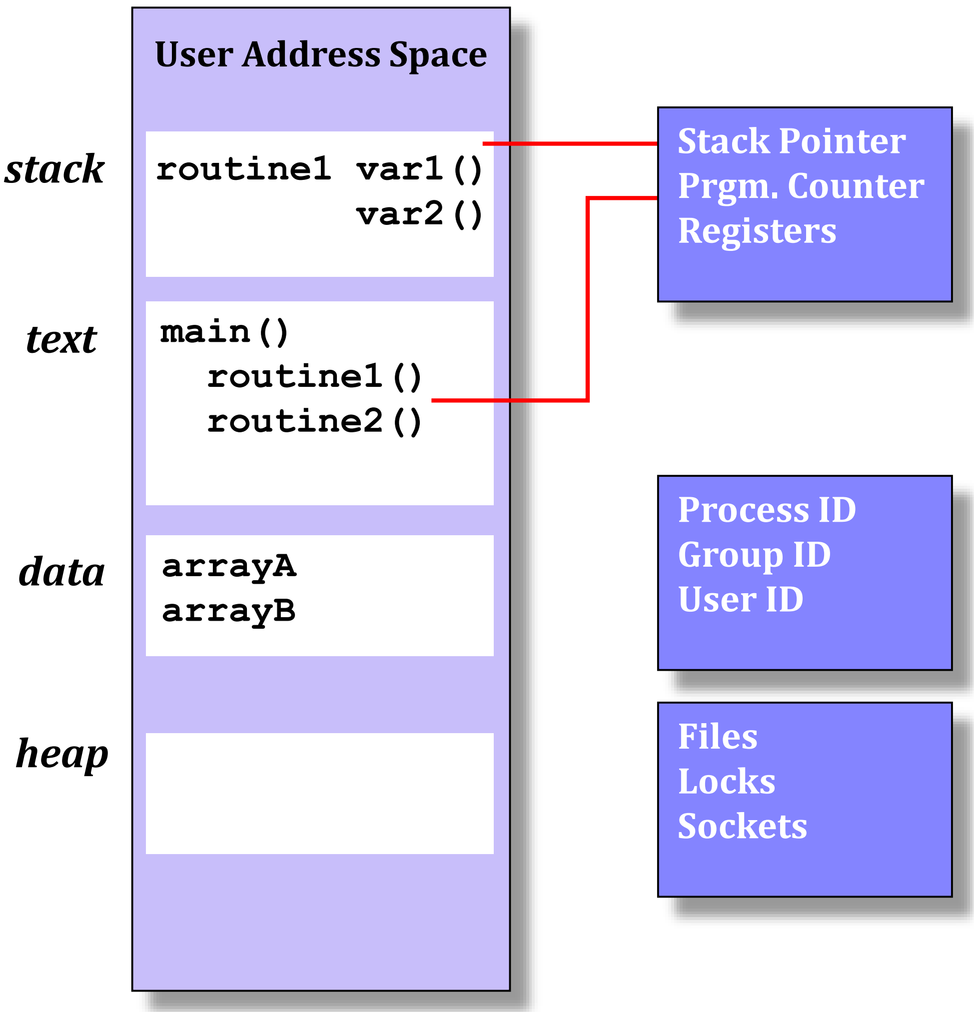
A Shared Memory System



Processes and Threads

- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program a single process may have multiple threads of control.

UNIX Process

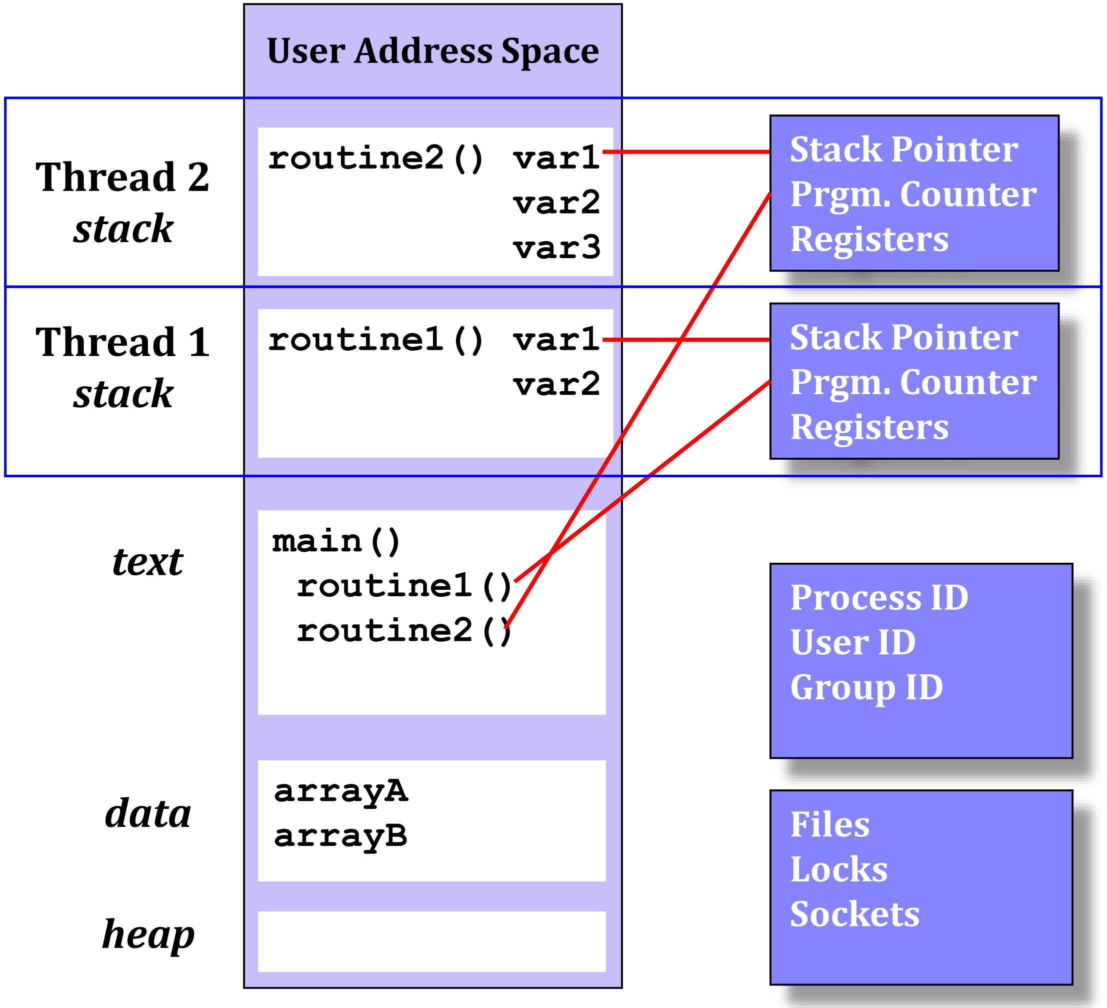


Process information:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

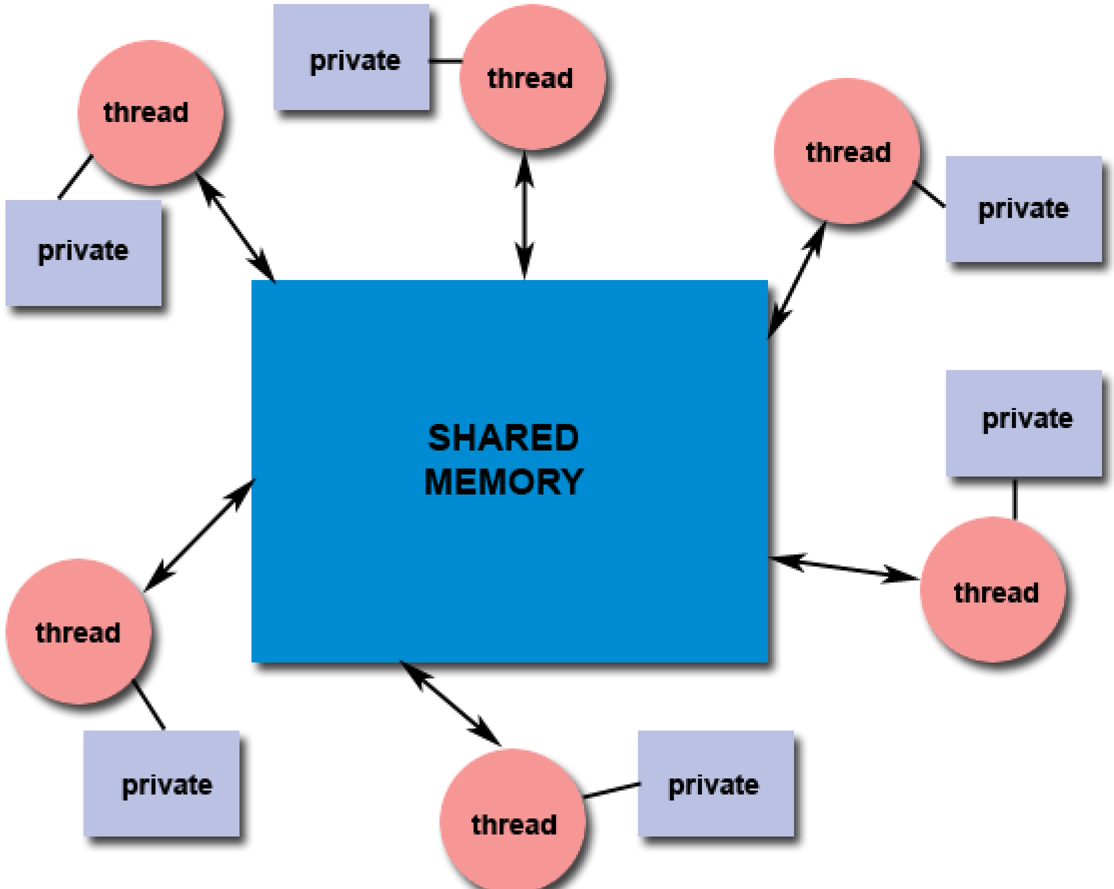
Threads within a UNIX Process



Thread information:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.

Shared Memory Model:



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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

POSIX®Threads

- Also known as Pthreads.
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.

POSIX——Portable Operating System Interface of UNIX

Caveat

- The Pthreads API is only available on POSIXR systems — Linux, MacOS X, Solaris, HPUX, …

Hello Pthreads!

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

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

void* thread(void* vargp);

int main(int argc, char** argv)
{
    pthread_t tid;                              //存放线程ID
    pthread_create(&tid, NULL, thread, NULL);  //创建一个新线程
    pthread_join(tid, NULL);                    //终止线程
    exit(0);
}

// 线程例程
void* thread(void* vargp)
{
    printf("Hello, Pthreads!¥n");
    return NULL;
}
```

Hello World! (1)

```c
#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       n;  /* Use long in case of a 64-bit system*/
   pthread_t* tid;

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

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

Hello World! (2)

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

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

   for (n = 0; n < thread_count; n++)
      pthread_join(tid[n], NULL);

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

void* Hello(void* rank) {
   long my_rank = (long)rank;
       printf("Hello from thread %ld of %d¥n", my_rank,
             thread_count);
       return NULL;
} /* Hello */
```

教材有误

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Compiling a Pthread program

```
gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

link in the Pthreads library

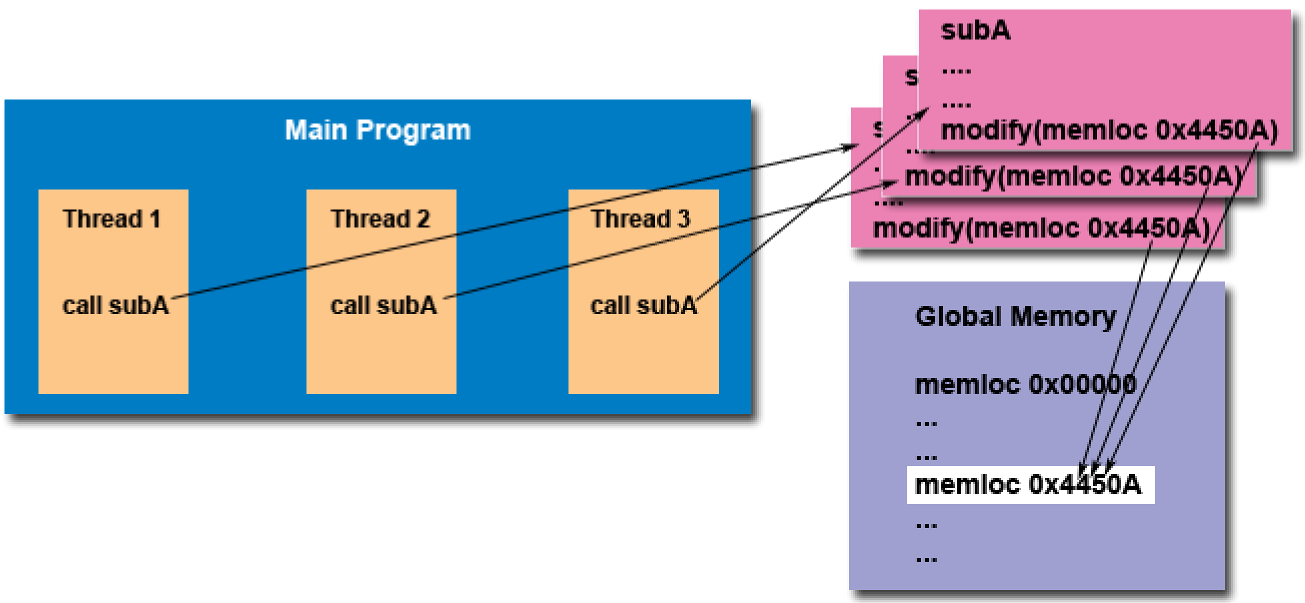
Running a Pthreads program

```
./pth_hello <number of threads>

./pth_hello 1

        Hello from the main thread
        Hello from thread 0 of 1

./pth_hello 4

        Hello from the main thread
        Hello from thread 0 of 4
        Hello from thread 1 of 4
        Hello from thread 2 of 4
        Hello from thread 3 of 4
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Global variables

- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they're really needed.
  - Shared variables.



北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Starting the Threads

- Processes in MPI are usually started by a script.
- In Pthreads the threads are started by the program executable.

```
mpiexec -n 4 ./mpi_hello

./pth_hello 4
```

Starting the Threads

pthread.h

pthread_t

One object for each thread.

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

pthread_t objects

- Opaque
- The actual data that they store is system-specific.
- Their data members aren't directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it's associated.

A closer look

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

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

Allocate <u>before</u> calling.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

A closer look

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

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

The function that the thread is to run.

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Function started by pthread_create

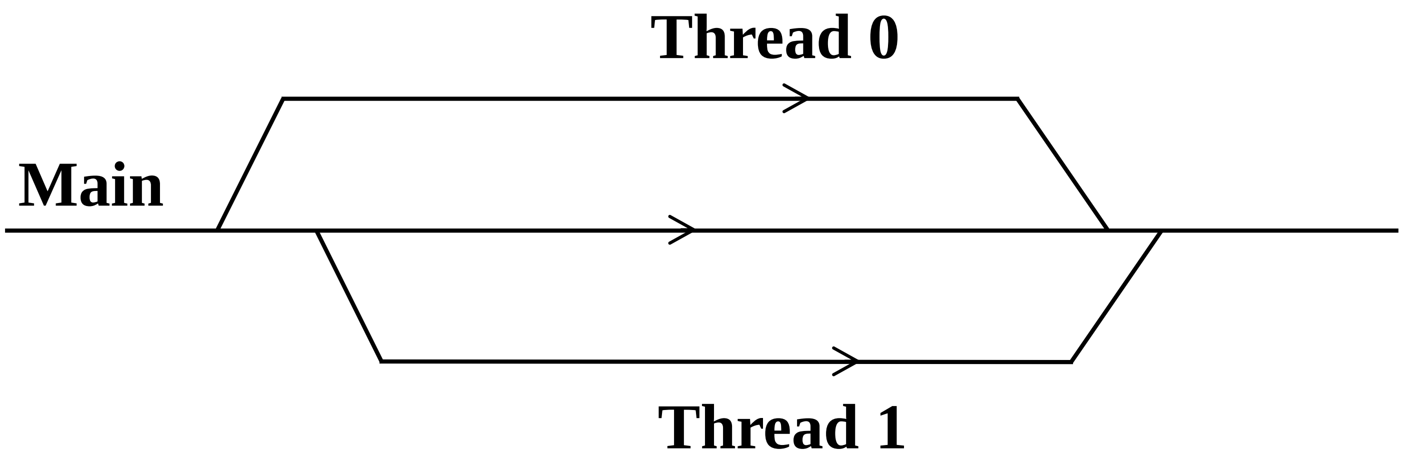
- Prototype:

```
void* thread_function(void* args_p);
```

- void* can be cast to any pointer type in C.
- So args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Running the Threads



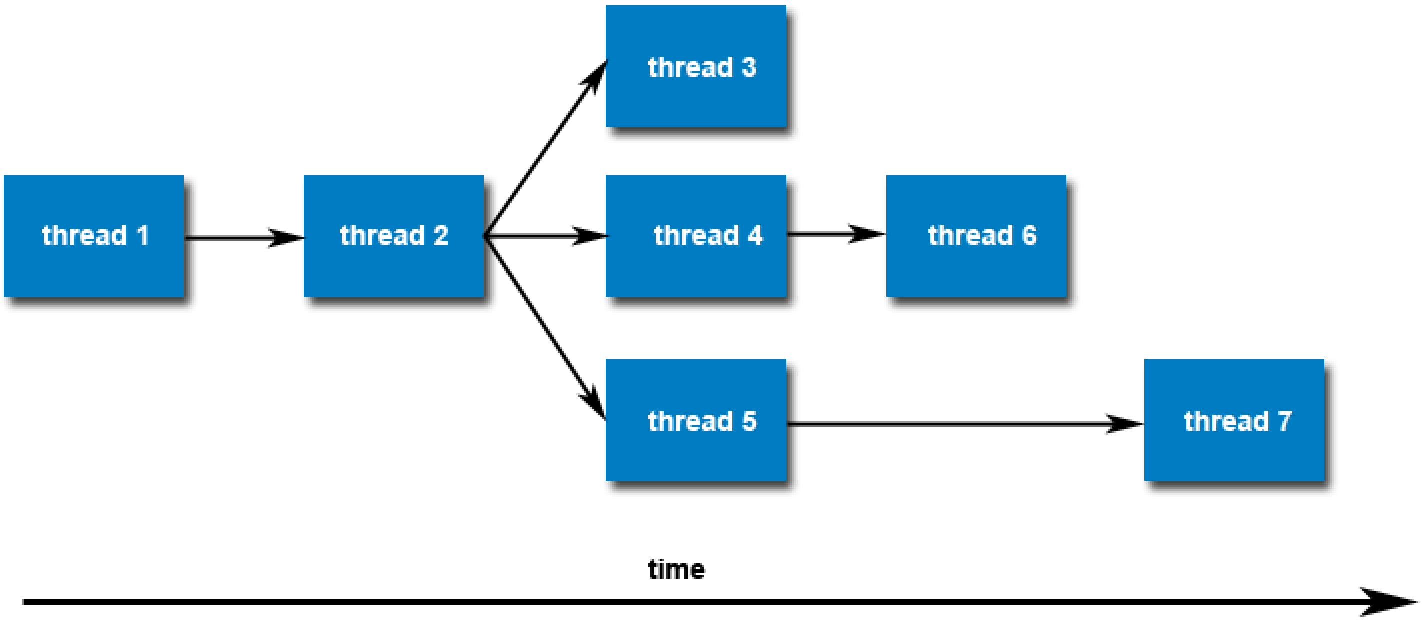
Main thread forks and joins two threads.

Processes (including threads) limitation

```
buaa@Ubantu:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 39730
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 39730
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
buaa@Ubantu:~$
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Threads are peers, and may create others



Stopping the Threads

- We call the function pthread_join once for each thread.
- A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

Matrix-Vector Multiplication in pthreads

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Serial pseudo-code

```
/* 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];
}
```

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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Using 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

```
y[0] = 0.0;
for (j = 0; j < n; j++)
      y[0] += A[0][j]* x[j];
```

general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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 */
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

2. CRITICAL SECTIONS

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Estimating π

中文教材有误!

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

```
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 ;
```

Using a dual core processor

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Note that as we increase n, the estimate with one thread gets better and better.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

A thread function for computing 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)  /* my_first_i is even */
       factor = 1.0;
    else                     /* my_first_i is odd */
       factor = -1.0;

    for(i = my_first_i; i < my_last_i; i++, factor = -factor){
       sum += factor/(2*i+1);
    }
    return NULL;
} /* Thread_sum */
```

Possible race condition

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

① Busy-Waiting

- 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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Pthreads global sum with busy-waiting

```
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 += my_sum;
      flag = (flag+1)%thread_count;
   }

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

Global sum function with critical section after loop (1)

```
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;
   else
      factor = -1.0;
```

Global sum function with critical section after loop (2)

```
    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 */
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

② Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.

Mutexes

- Mutexes can be abstracted as four operations:

系统	Win32	Linux	Solaris
创建	CreateMutex	pthread_mutex_init	mutex_init
加锁	WaitForSingleObject	pthread_mutex_lock	mutex_lock
解锁	ReleaseMutex	pthread_mutex_unlock	mutex_unlock
销毁	CloseHandle	pthread_mutex_destroy	mutex_destroy

Mutexes

- Used to guarantee that one thread “excludes” other threads while it executes the critical section. The Pthreads standard includes a special type for mutexes: pthread_mutex_t.

```
int pthread_mutex_init(
      pthread_mutex_t*             mutex_p   /* out */,
      const pthread_mutexattr_t*   attr_p    /* in  */);
```

Mutexes

- In order to gain access to a critical section a thread calls

```
int pthread_mutex_lock(pthread_mutex_t*  mutex_p  /* in/out */);
```

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

- When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Global sum function that uses a mutex (1)

```
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;
    else
        factor = -1.0;
```

Global sum function that uses a mutex (2)

```
    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 */
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Threads	Busy-Wait	Mutex
1	0.393	0.394
2	0.198	0.197
4	0.0998	0.0995
8	0.051	0.051
16	0.026	0.039
32	0.045	0.031
64	0.109	0.029

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count}$$

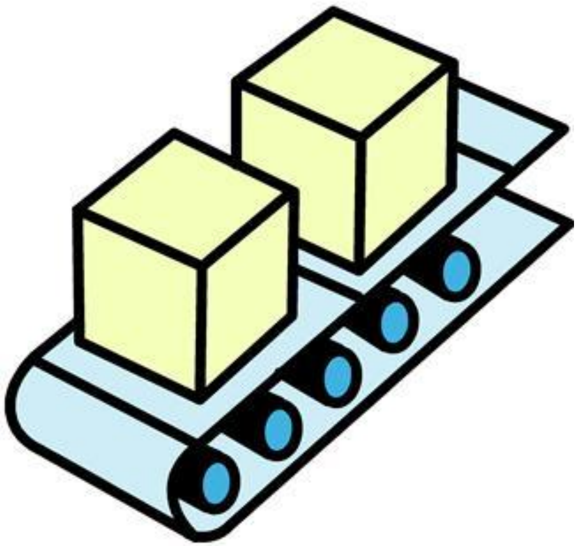
Run-times (in seconds) of π programs using n = 10^8 terms on a system with two four-core processors.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.

③ Producer-Consumer Synchronization and Semaphores



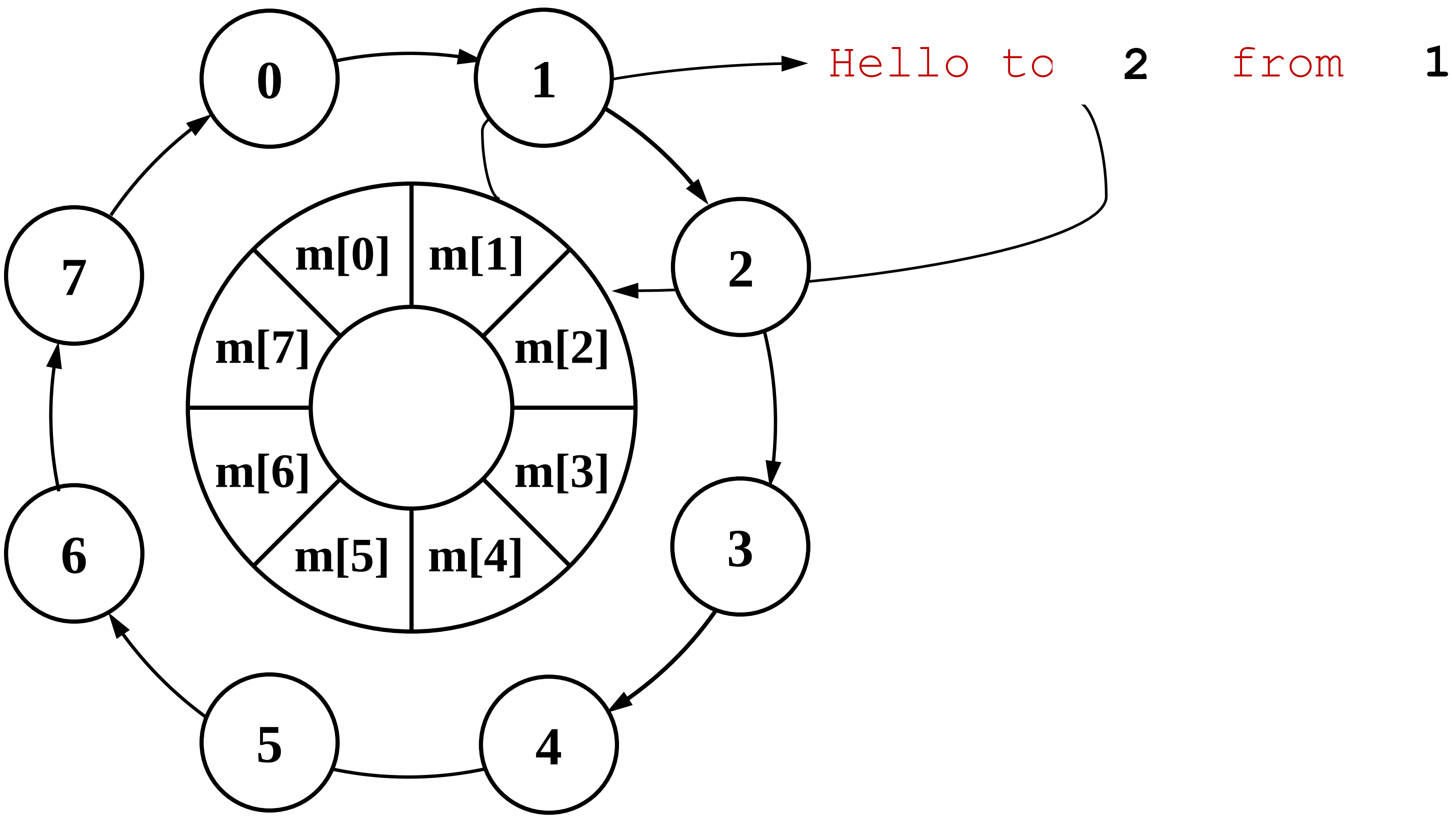
Issues

- Busy-waiting enforces the order threads access a critical section.
- Using mutexes, the order is left to chance and the system.
- There are applications where we need to control the order threads access the critical section.

Problems with a mutex solution

```
/* n 和 product_matrix 是共享的，且都是通过主线程的  */
/* product_matrix初始化为同样的矩阵                 */
void* Thread_work(void* rank){
   long my_rank = (long)rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);

   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);

   Free_matrix(&my_mat);
   return NULL;
} /* Thread_work */
```

Send messages using pthreads



A first attempt at sending messages using pthreads

```c
/* 消息体的类型为 char**，且在 main 函数中已分配地址空间，并初始化为NULL。*/
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
   else
      printf("Thread %ld > No messa

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

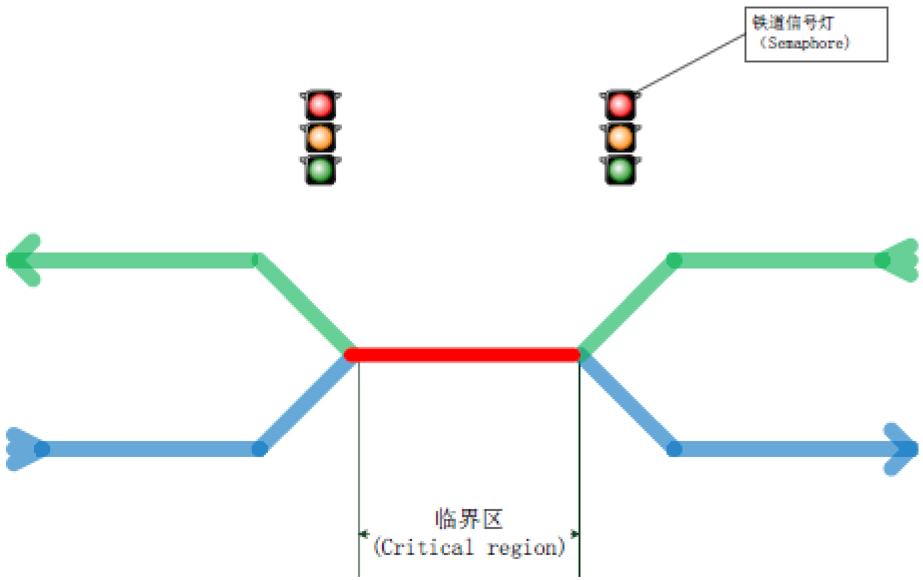
A first attempt at sending messages using pthreads

```
/* 消息体的类型为 char**，且在 main 函数中已分配地址空间，并初始化为NULL。*/
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;

   while (messages[my_rank] == NULL)
   printf("Thread %ld > %s¥n", my_ra



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

```
buaa@Ubantu:~/文档$ ./pth_msg 4
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 0 > Hello to 0 from 3
buaa@Ubantu:~/文档$ ./pth_msg 4
Thread 2 > Hello to 2 from 1
Thread 1 > Hello to 1 from 0
Thread 3 > Hello to 3 from 2
Thread 0 > Hello to 0 from 3
```

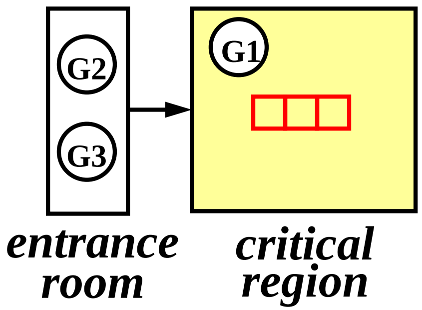
北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Binary semaphore

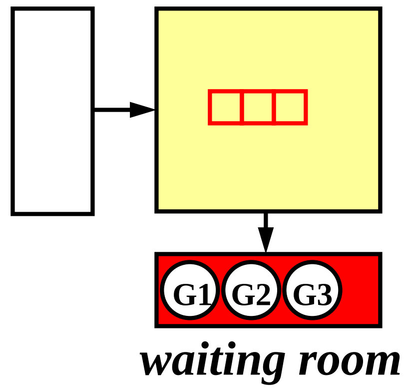


Visualization of the Synchronized Buffer

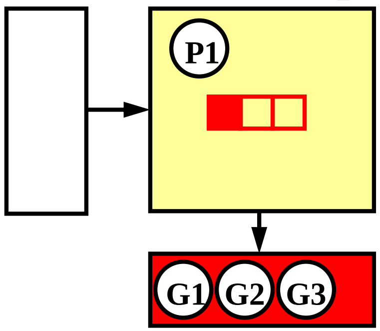
3 *Get* threads arrive at the empty buffer.



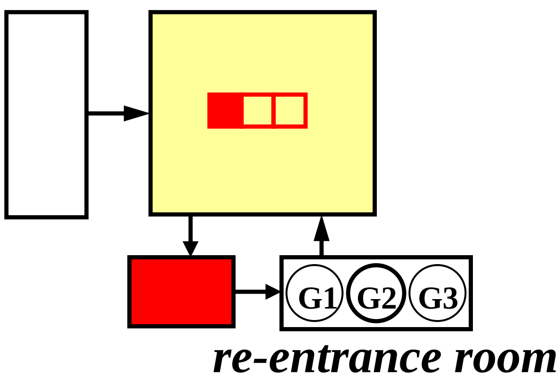
they enter the critical region one after the other and go to sleep because the buffer is empty.



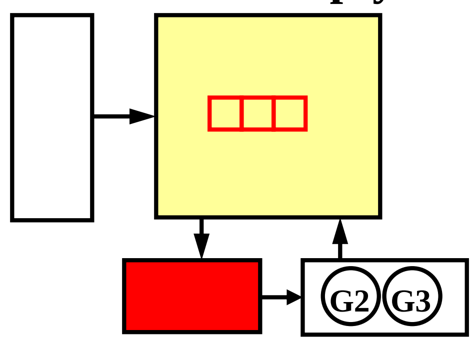
a *Put* thread arrives; it enters the critical region, deposits its data and calls *sem_post.*



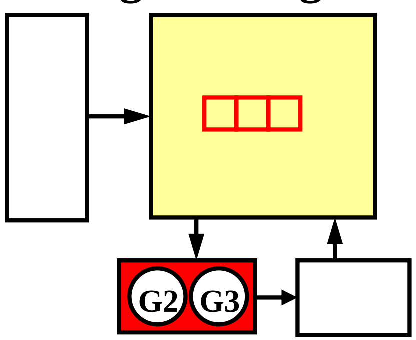
all *Get* threads are waked up; the first one enters the critical region, reads the data and leaves.



the others also enter the critical region but find the buffer empty...



... so they go to the waiting room again.



Example: Synchronized Buffer

If producer is faster	If consumer is faster
Put	Put
Put	Get
Put	Put
Put	Get
Get	...
Put	
Get	
...	

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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 */);
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Using semaphores to send messages

```
/* 消息已在main函数里分配空间并初始化为 NULL                  */
/* semaphores也已在main函数里分配空间并初始化为0(locked)  */
void* Send_msg(void* rank){
   long my_rank = (long)rank;
   long dest = (my_rank + 1)%thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   Sem_post(&semaphores[dest
         /*  “解锁”目标线程的

   /* 等待我们的信号量被解锁 */
   Sem_wait(&semaphores[my_r
   printf("Thread %ld > %s¥n

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

```
buaa@Ubantu:~/文档$ ./pth_msg_sem 4
Thread 2 > Hello to 2 from 1
Thread 1 > Hello to 1 from 0
Thread 3 > Hello to 3 from 2
Thread 0 > Hello to 0 from 3
buaa@Ubantu:~/文档$ ./pth_msg_sem 4
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 0 > Hello to 0 from 3
```

3. BARRIERS AND CONDITION VARIABLES

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

Using barriers to time the slowest thread

```
/* Shared  */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;
elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point¥n");
   fflush(stdout);
}
```

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

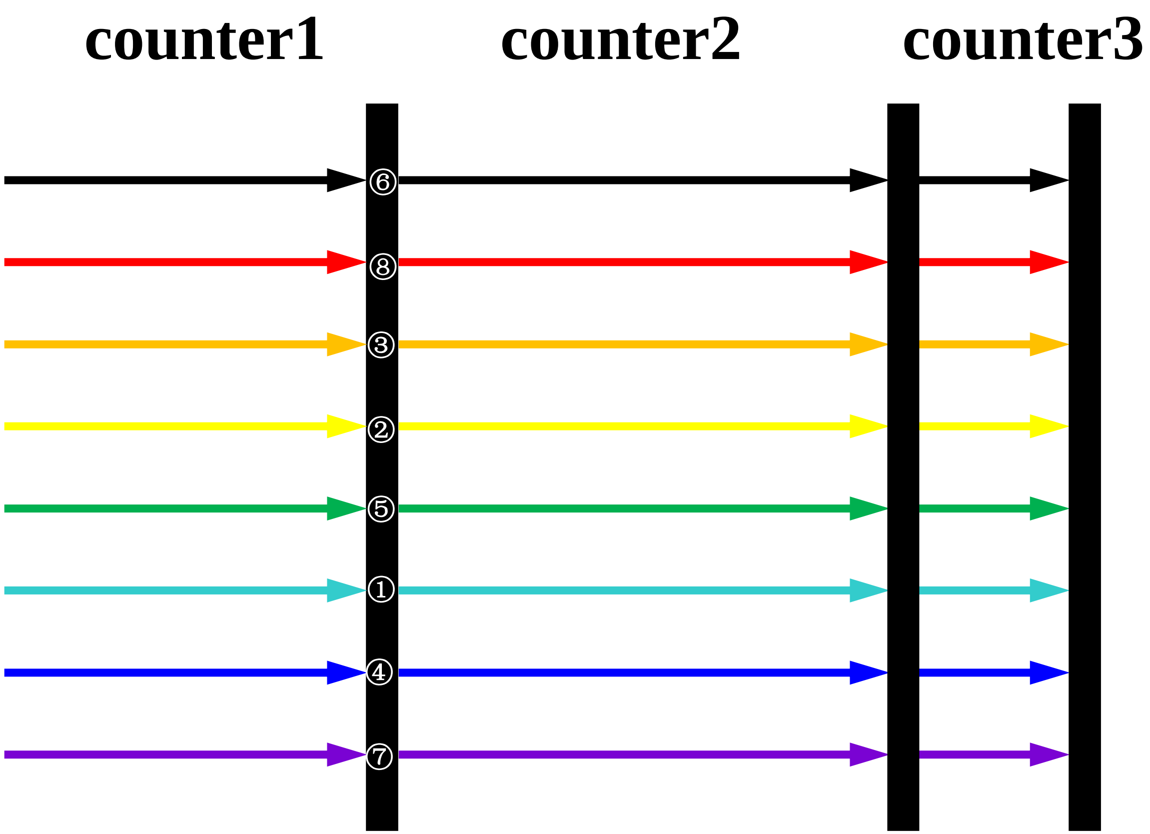
Busy-waiting and a Mutex

```
/* 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.

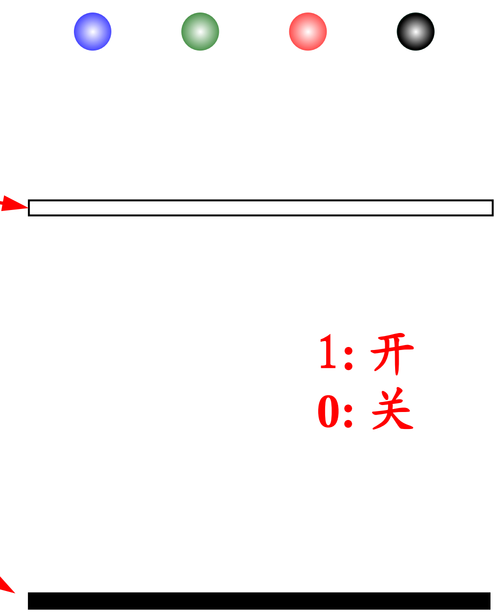
北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Busy-waiting and a Mutex



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);
   }
   ...
}
```



1: 开
0: 关

post: +1
wait: -1(信号量是unsigned int类型。不为0时-1；为0时阻塞，直至为正数)

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Condition Variables

```
pthread_mutex_lock(&mutex);
counter++;
if (counter == thread_count)
   pthread_cond_broadcast(&cond_var);
else {
   while(pthread_cond_wait(&cond_var, &mutex) != 0);
   /* when thread is unblocked, mutex is relocked*/
}
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Implementing a barrier with condition variables

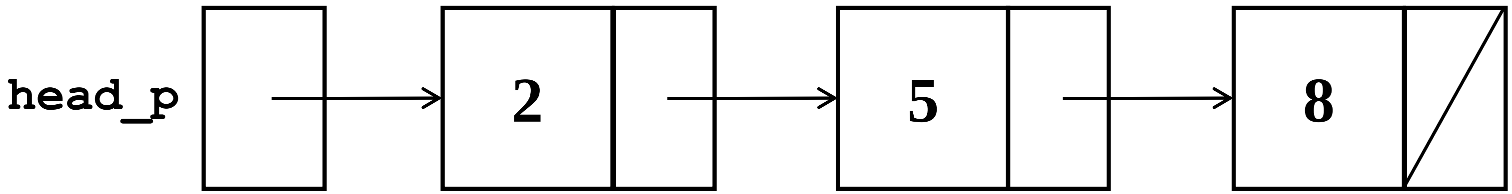
```
/* Shared */
int               counter = 0;
pthread_mutex_t mutex;
pthread_cond_t  cond_var;
...
void* Thread_work(...){
   ...
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count){
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while(pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   ...
}
```

4. 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.

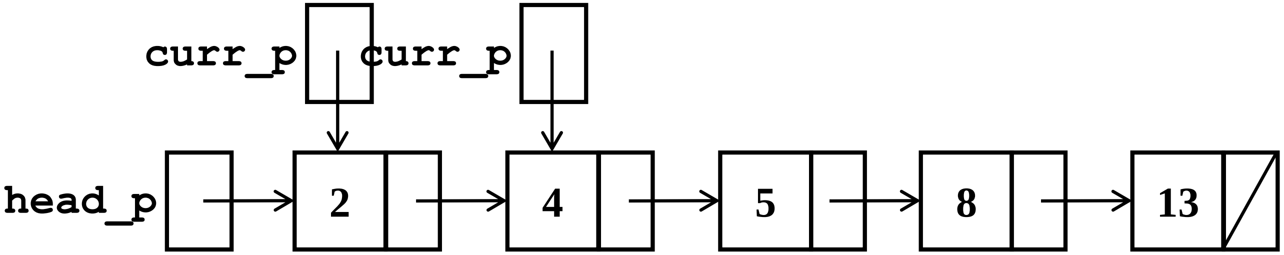
Linked Lists



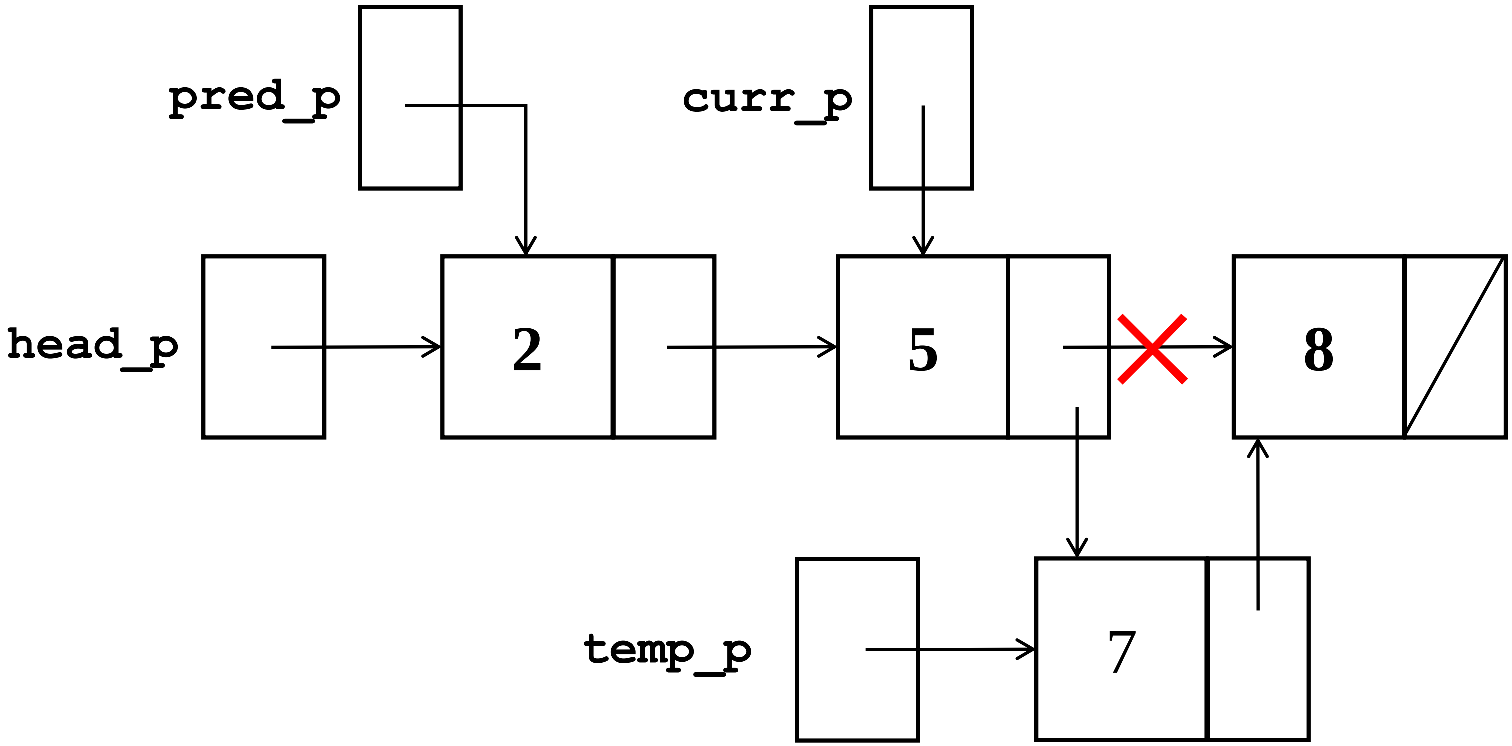

```
struct list_node_s {
        int data;
        struct list_node_s* next;
}
```

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

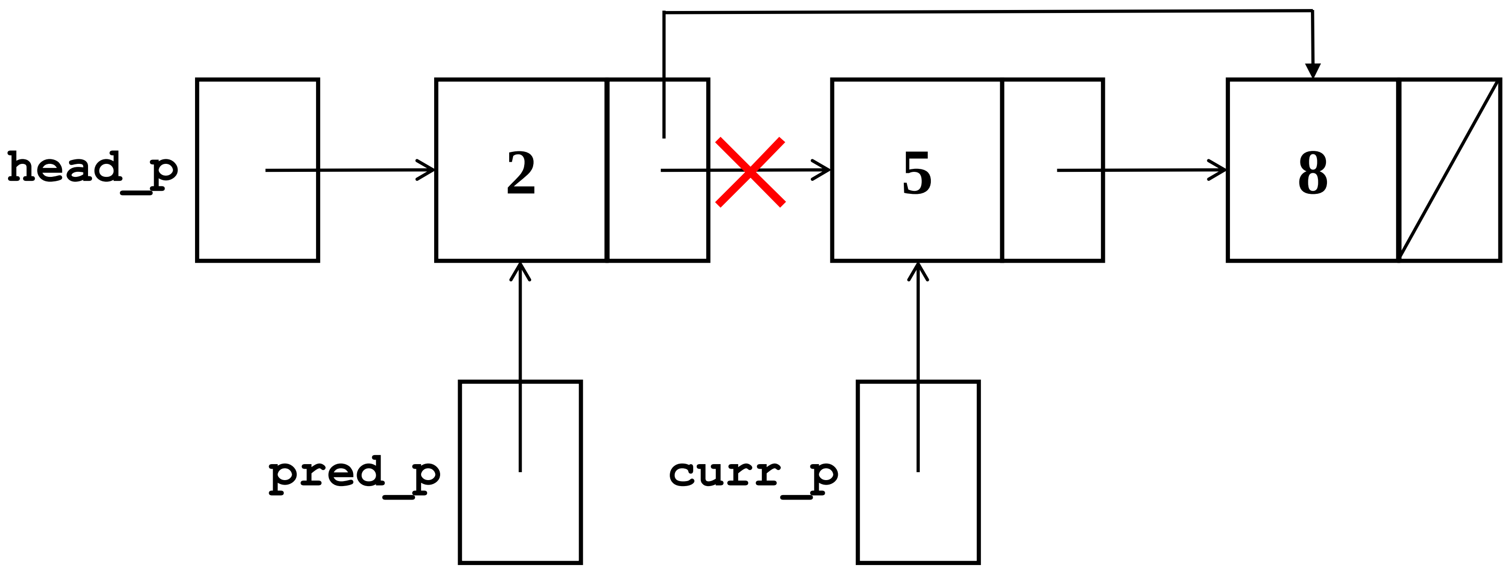


Inserting a new node into a list

```
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)   /* 新的头结点 */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else {      /* 被插入的值已经存在于链表中 */
      return 0;
   }
} /* Insert */
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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) { /* 删除的是链表中的第一个结点*/
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* 值不在链表中 */
      return 0;
   }
} /* Delete */
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

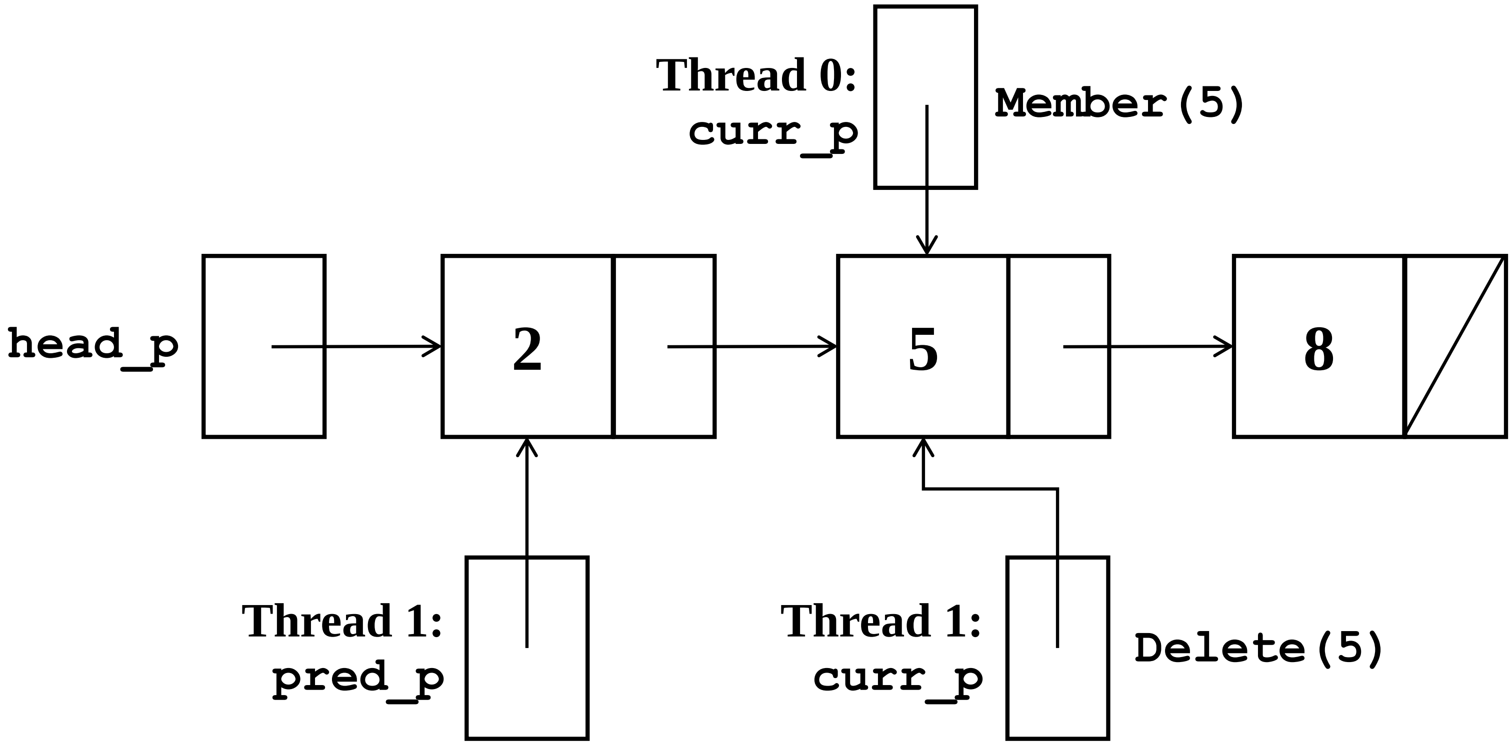
A Multi-Threaded Linked List

- Let's try to use these three 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.

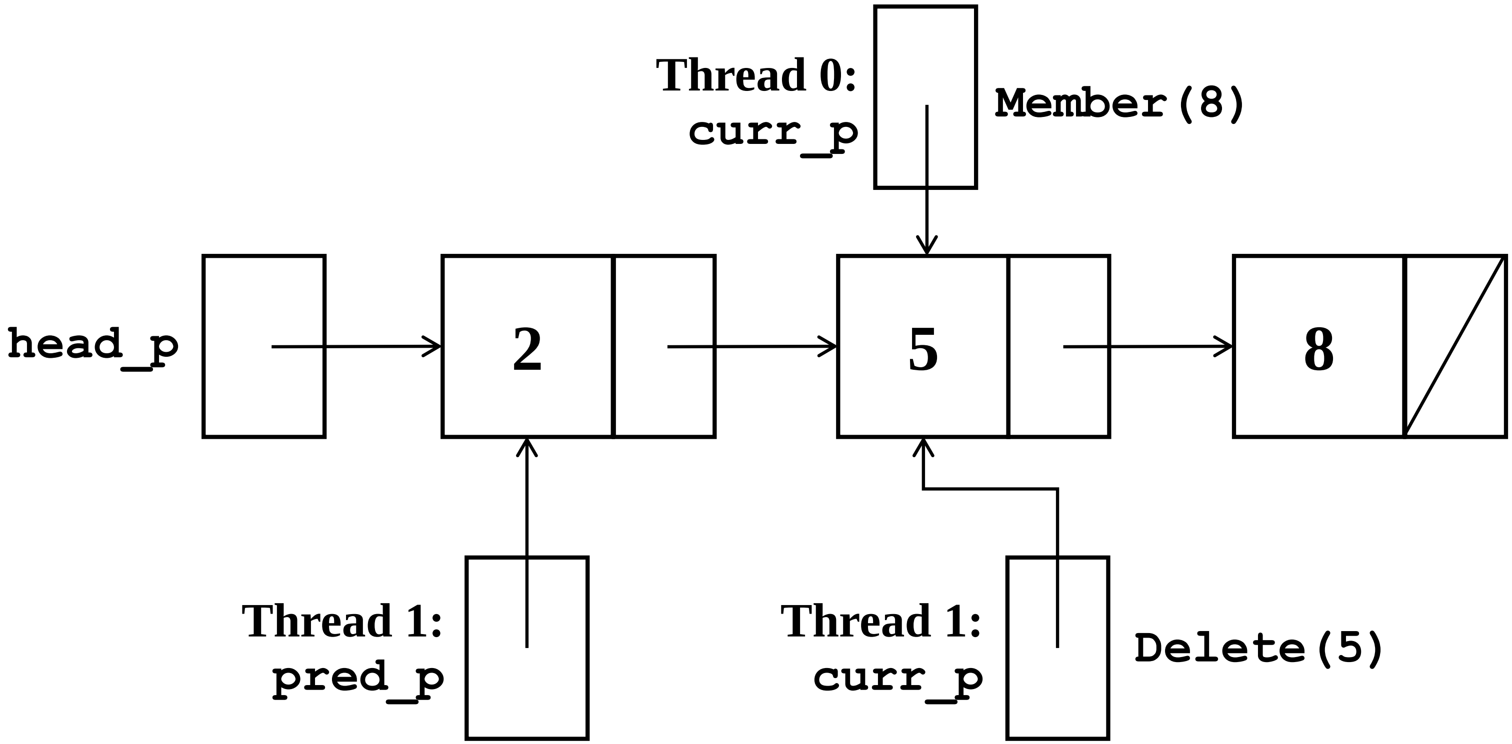
```
int Member(int value, struct list_node_s* head_p)
int Insert(int value, struct list_node_s** head_pp)
int Delete(int value, struct list_node_s** head_pp)
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Simultaneous access by two threads



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.

链表锁

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

In place of calling Member(value).

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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;
}
```

结点锁

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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)

```c
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)  /* 后面还有结点 */
            pthread_mutex_lock(&(temp_p->next->mutex));
        if (temp_p == head_p)      /* 只有一个头结点 */
            pthread_mutex_unlock(&head_p_mutex);
        pthread_mutex_unlock(&(temp_p->mutex));
        temp_p = temp_p->next;
    }
```

北京航空航天大学 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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 */
```

This program fails to lock the mutex associated with the first node of the list !

Correct implementation of Member with one mutex per list node (1)

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

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;

   /* 若链表非空，给temp_p指针上锁 */
   if (temp_p != NULL)
      pthread_mutex_lock(temp_p->mutex);

   /* 不再需要head_p指针的锁 */
   pthread_mutex_unlock(&head_p_mutex);

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

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Correct implementation of Member with one mutex per list node (2)

```
      /* 前进到下一个结点 */
      old_temp_p = temp_p;
      temp_p = temp_p->next;

      /* 此时给上一个结点的temp_p指针解锁 */
      pthread_mutex_unlock(&(old_temp_p->mutex));
   }

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

北京航空航天大学
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

读写锁的三种状态：

1. 当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞；

2. 当读写锁在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是以写模式对它进行加锁的线程将会被阻塞；

3. 当读写锁在读加锁状态时，如果有另外的线程试图以写模式加锁，读写锁通常会阻塞随后的读模式锁的请求。这样可以避免读模式锁长期占用，而等待的写模式锁请求则长期阻塞。

处理读-写问题的两种常见策略:

1. 强读者同步(strong reader synchronization)

　　总是给读者更高的优先权，只要写者当前没有进行写操作，读者就可以获得访问权限。

2. 强写者同步(strong writer synchronization).

　　往往将优先权交付给写者，而读者只能等到所有正在等待的或者是正在执行的写者结束以后才能执行。

　　关于读者-写者模型中，由于读者往往会要求查看最新的信息记录，所以航班订票系统往往会使用强写者同步策略，而图书馆查阅系统则采用强读者同步策略。

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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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

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.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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.