Multicore Computing
Lecture03 - Pthreads

SUNG KYUN KWAN UNIVERSITY

남 범 석

bnam@skku.edu

POSIX Threads

- This lecture will introduce Pthreads for beginners
  - Will not cover every detail of Pthreads though

- Reference
  - https://computing.llnl.gov/tutorials/pthreads/

What is a Thread? (Recap from OS class)

- An independent stream of instructions that can be scheduled to run by the operating system

- This independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data

- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer

Overview of Programming Models

- A single-thread process:

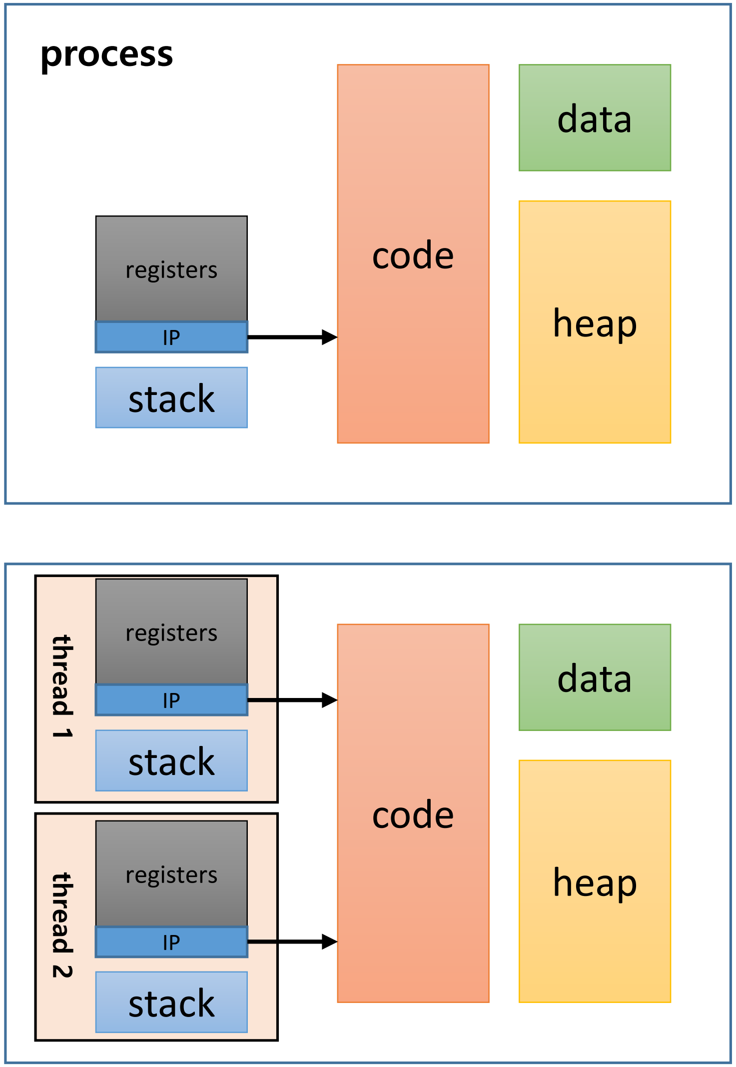
```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
        c[row][column] =
        dot_product( get_row(a, row), get_col(b, col));
```

- A multi-thread process:

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
        c[row][column] =
        create_thread( dot_product(get_row(a, row),
                                   get_col(b, col)));
```

Threads vs. Processes

- Process
  - One address space per process
  - Each process has its own data (global variables), stack, heap
- Thread
  - Multiple threads share on address space
    - But its own stack and register context
  - Threads within the same address space share data (global variables), heap



What is P(POSIX®)Threads?

- POSIX: Portable Operating System Interface
- Standard threads API supported by most vendors

- Historically, hardware vendors have implemented their own proprietary versions of threads
  - Substantially different from each other
  - Make it difficult for programmers to develop portable threaded apps

- The need for a standardized programming interface
  - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995)
    - POSIX: Portable Operating System Interface
  - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads

- Pthreads are defined as a set of C language programming types and procedure calls
  - Implemented with a pthread.h header/include file and a thread library

Why Pthreads?

- Mainly for higher performance gains when exploiting parallelism
  - Much more efficient than creating and managing processes

- Example: creating 50,000 processes/threads (units in seconds)

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

The Pthreads API

- The subroutines of the Pthreads API can be informally grouped into the following major groups:
  - Thread management:
    - Routines that work directly on threads - creating, detaching, joining, etc
  - Mutexes:
    - Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion"
  - Condition variables:
    - Routines that address communications between threads that share a mutex.
  - Synchronization:
    - Routines that manage read/write locks and barriers.

Pthreads: Naming Conventions

- Naming conventions:
  - All identifiers in the threads library begin with pthread_.

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Thread Creation and Destruction

- Typical Pthreads code would look like below

```
#include <pthread.h>
#define NT 5
int err;

void main()
{
  pthread_t tid[NT]; // array of thread IDs, one per thread
  for (i=0; i<NT; i++)
    err=pthread_create(&tid[i], NULL, printHello, (void*) i);

  for (i=0; i<NT; i++)
    err=pthread_join(tid[i], (void **)&status[i]);
}
```

- Threads are created in the first loop
  - Parent ➔ the creating thread, children ➔ created threads
- The parent thread waits for its children to complete

Creating a Pthread

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

Specify thread handle (allocated before calling)

Specify the attributes of a creating thread

The function that the thread is to run

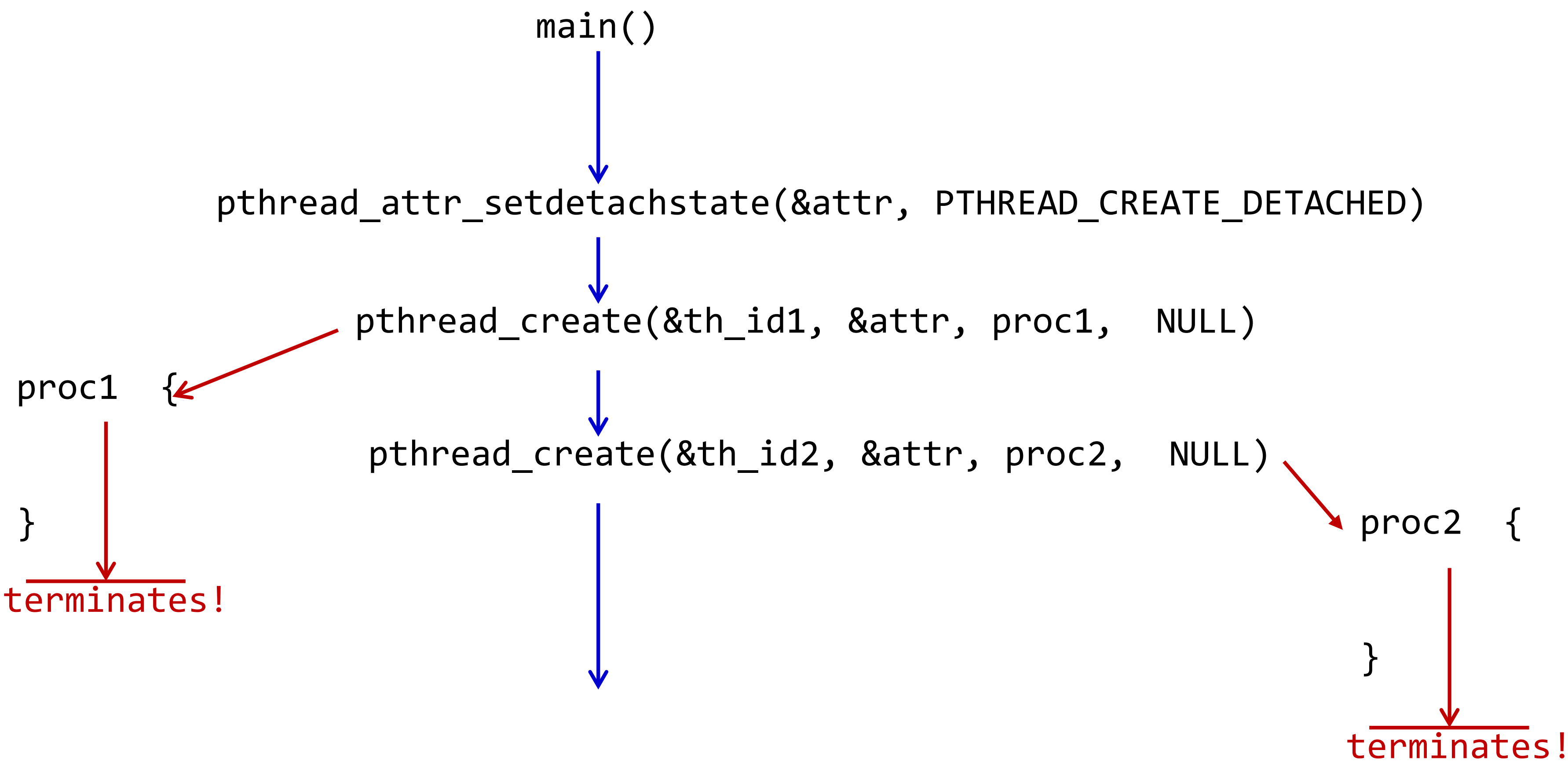
Pointer to the argument passed to the function ***start_routine***

Pthread Attributes

- Stack size
- Detach state
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
    - Release resources at termination (detached) or retain (joinable)
- Scheduling policy
  - SCHED_OTHER: standard policy
  - SCHED_FIFO, SCHED_RR
- Scheduling parameters
  - Priority only
- Inherit scheduling policy
  - PTHREAD_INHERIT_SCHED, PTHREAD_EXPICIT_SCHED
- Thread scheduling scope
  - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
- Special functions exist for getting/setting each attribute
  - int pthread_attr_setstack_size(pthread_attr_t* attr, size_t stacksize)

Pthreads – detached thread

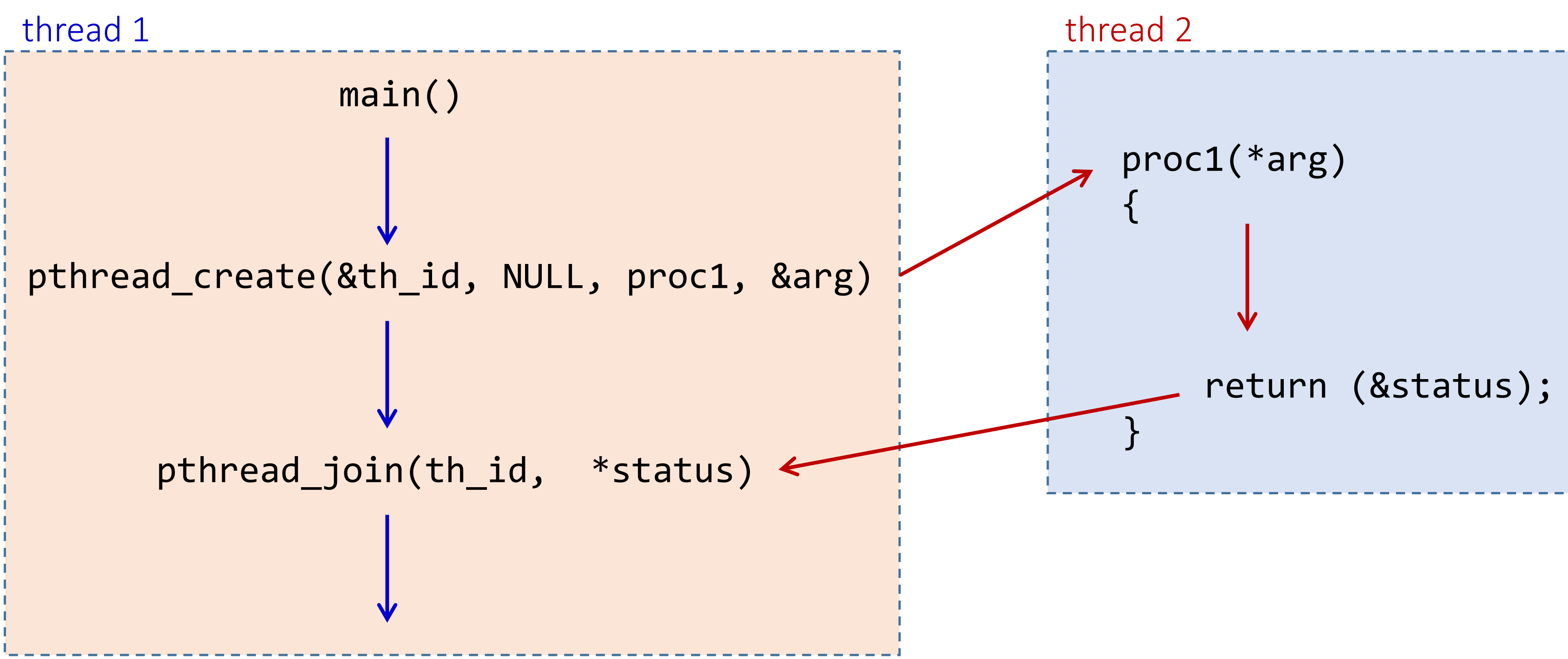
- pthread_attr_setdetachstate
  - Detached threads are not joinable



Waiting a Pthread

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

- Returns after a specified thread terminates
- ptr stores return code of a terminating thread



Exiting a Pthread

- 4 ways to exit threads
  - Thread will naturally exit after starting thread function returns
  - Thread itself can exit by calling `pthread_exit()`
  - Other threads can terminate a thread by calling `pthread_cancel()`
    - A specified thread terminates when it reaches a **cancelation point**
  - Thread exits if the process that owns the thread exits

- APIs
  - `void pthread_exit (void *retval);`
  - `int pthread_cancel (pthread_t thread)`

Pthreads – compilation

- Pthreads are supported by almost all compilers
  - GNU Compiler
    - **`gcc -Wall -o hello hello.c -lpthread`**
    - **`-lxxx`** : specifies which static library to link
    - **`-Wall`** : specifies to print out all types of warnings

Thread Argument Passing

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

- We often want to pass multiple parameters to threads when creating them
  - create a struct which contains all of the arguments, and pass a pointer to that struct
- Each thread needs to have different parameter values
  - Allocate an array of the struct whose length equals to the thread count
  - Each thread references its own parameters using its thread ID as an index to the array

Synchronization

- Accessing shared data
  - Example: two threads increase the same variable x

```
int x = 0;
void* inc () {
   x = x + 1;
   return NULL;
}

main()
{
   pthread_create(&th1, NULL, inc, NULL);
   pthread_create(&th2, NULL, inc, NULL);

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   printf("x = %d\n", x);
}
```

thread 1	thread 2
load r1 ← x	load r1 ← x
add r1 ← r1, 1	add r1 ← r1, 1
store r1 → x	store r1 → x

time

Critical Sections

- Critical section
  - Need to guarantee that one process (thread) can access a certain resource at a time
  - Implemented mechanism is known as "mutual exclusion"

- Locks
  - Simple mechanism for mutual exclusion
  - A lock can have only two values
    - 1 – a thread entered the critical section
    - 0 – no thread is in the critical section
  - Acquire the lock before entering the critical section (set to 1)
  - Release the lock after leaving the critical section (set to 0)

Acquiring Locks

- Does the following simple C code work?

```
void lock(lock_var) {
    while (lock_var != 0);
    lock_var = 1;
}

void unlock(lock_var) {
     lock_var = 0;
}
```

- Special *atomic* instruction should be used
  - Pthread library provides APIs

Mutual Exclusion (Mutex)

- Pthreads provides mutex to avoid race conditions

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
…
pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

- pthread_mutex_lock
  - A thread will wait until it can acquire the lock
- pthread_mutex_unlock
  - If multiple threads are waiting, only one thread is selected to receive the lock
  - Only the thread that acquires the lock can unlock it

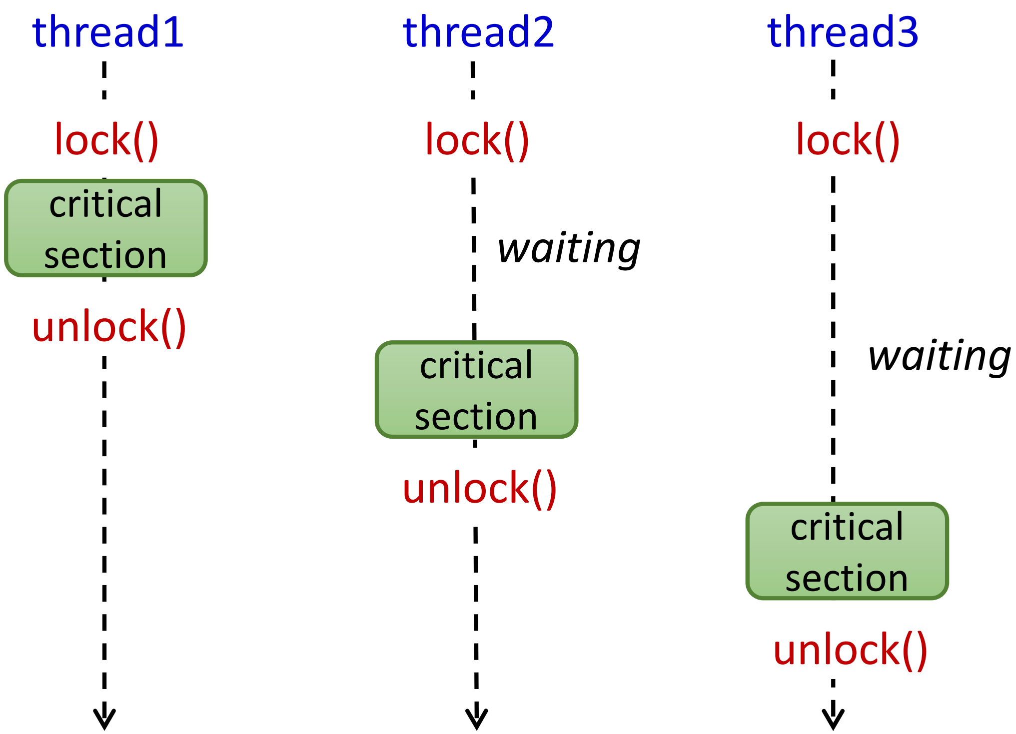
Example: Sum an array using Pthreads

```
#include <pthread.h>

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;

void *do_work(void *tid)
{
    int i, start, mytid, end;
    int local_sum=0;
    mytid = (int) tid;
    start = mytid*array_size/no_threads;
    end = start+array_size/no_threads;
    for (i=start; i<end; i++) {
        local_sum = a[i];
    }

    pthread_mutex_lock(&mutex1);
    sum += local_sum;
    pthread_mutex_unlock(&mutex1);

    pthread_exit(NULL);
}
```

```
void main() {
  int i;
  pthread_t thread[no_threads];
  pthread_attr_t attr;

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
                      PTHREAD_CREATE_JOINABLE);

    …

  for (i = 0; i < no_threads; i++)
      pthread_create(&thread[i], &attr,
                      do_work, (void*) i);

  for (i = 0; i < no_threads; i++)
       pthread_join(thread[i], NULL);

  printf("The sum of 1 to %i is %d\n",
                      array_size, sum);

  pthread_attr_destroy(&attr);
  pthread_exit(NULL);
}
```

Serialization

- Critical sections serialize the code execution
  - Too many or large critical sections can slow down the performance – sequential code may run faster



Alleviating Locking Overhead

```
int pthread_mutex_trylock (
      pthread_mutex_t *mutex_lock);
```

- Reduce overhead by overlapping computation with waiting
  - Acquires lock if unlocked
  - Returns EBUSY if locked

Condition Variables

- Wait until a condition is satisfied
  - A global variable is used to indicate condition (predicate value)
- Three variables are linked all together
  - mutex lock, condition variable, predicate

thread1

```
action() {
   ...
   mutex_lock(&lock);
   while (predicate == 0)  // test predicate
      cond_wait(&cond, &lock);
   mutex_unlock(&lock);
   // perform action
   ...
}
```

thread2

```
signal() {
   ...
   mutex_lock(&lock);
   predicate = 1;  // set predicate
   cond_signal(&cond);
   mutex_unlock(&lock);
   ...
}
```

- When a thread waits using cond_wait, associated mutex is unlocked
- If a thread is signaled, returns after acquiring the mutex lock

Pthread Condition Variable API

- Initialize and destroy

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

- Wait for a condition

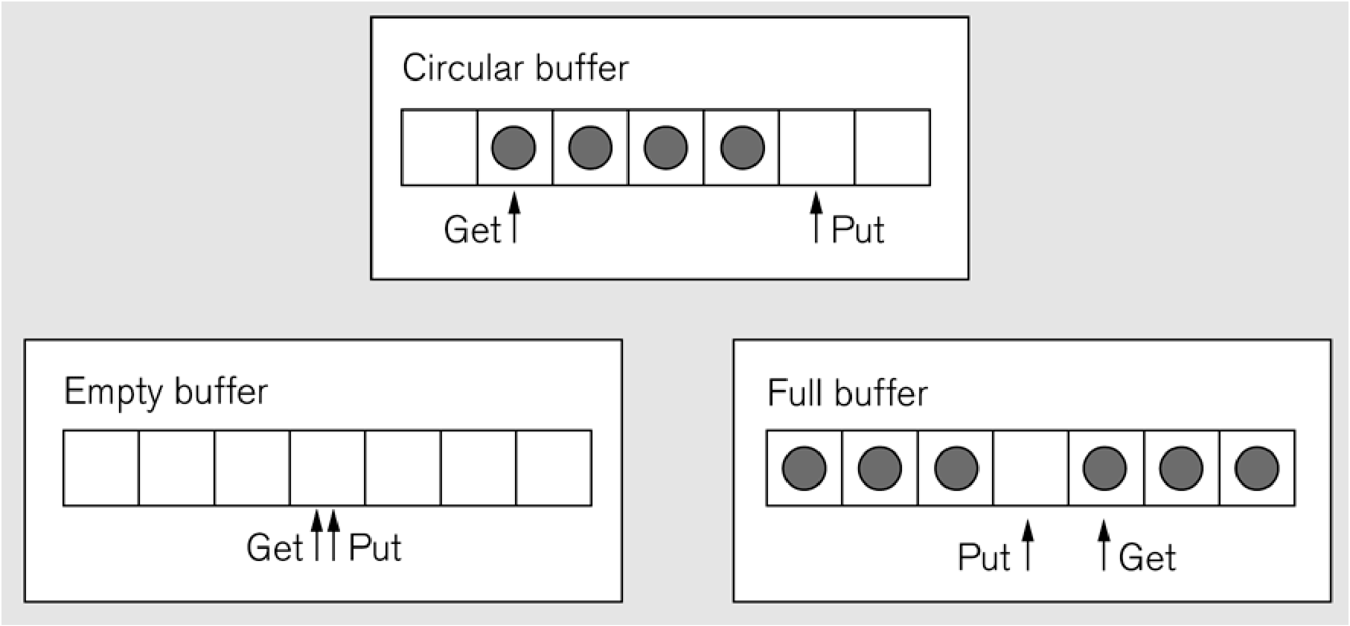
```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex,
                      const struct timespec *wtime);
```

- Signal one or all waiting threads

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

Bounded Buffer Example

- One or more threads put items into a circular buffer
  - They need to wait if the buffer is full
- Other threads remove items from the same buffer
  - They need to wait if the buffer is empty



Bounded Buffer Example

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int put=0;                          // Buff index for next insert
int get=0;                          // Buff index for next remove

void insert(Item x)                      // Producer thread
{
  pthread_mutex_lock(&lock);
  while((put>get&&(put-get)==SIZE-1)||   // While buffer is
        (put<get&&(put+get)==SIZE-1))    // full
  {
    pthread_cond_wait(&nonfull, &lock);
  }
  buffer[put]=x;
  put=(put+1)%SIZE;
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&lock);
}

Item remove()                            // Consumer thread
{
  Item x;
  pthread_mutex_lock(&lock);
  while(put==get)                        // While buffer is empty
  {
    pthread_cond_wait(&nonempty, &lock);
  }
  x=buffer[get];
  get=(get+1)%SIZE;
  pthread_cond_signal(&nonfull);
  pthread_mutex_unlock(&lock);
  return x;
}
```

- Two condition variables are used
  - nonempty
  - nonfull
- Two cursors
  - put: next empty location
  - get: points to next element to remove
- A mutex variable to protect access to CVs by multiple threads

Protecting Condition Variables

```
1  pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3  pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4  Item buffer[SIZE];
5  int put=0;                        // Buff index for next insert
6  int get=0;                        // Buff index for next remove
7
8  void insert(Item x)                     // Producer thread
9  {
10   pthread_mutex_lock(&lock);
11   while((put>get&&(put-get)==SIZE-1)||   // While buffer is
12         (put<get&&(put+get)==SIZE-1))    // full
13   {
14     pthread_cond_wait(&nonfull, &lock);
15   }
16   buffer[put]=x;
17   put=(put+1)%SIZE;
18   pthread_cond_signal(&nonempty);
19   pthread_mutex_unlock(&lock);
20 }
21
22 Item remove()                           // Consumer thread
23 {
24   Item x;
25   pthread_mutex_lock(&lock);
26   while(put==get)                       // While buffer is empty
27   {
28     pthread_cond_wait(&nonempty, &lock);
29   }
30   x=buffer[get];
31   get=(get+1)%SIZE;
32   pthread_cond_signal(&nonfull);
33   pthread_mutex_unlock(&lock);
34   return x;
35 }
```

Why do we need the while loop in Line 11?

- At the time of the signal, the buffer is not full, but when any particular thread acquires the mutex, the buffer may have become full again
- In this case, the thread should call pthread_cond_wait() again

Why do we need to protect pthread_cond_signal() in line 18 and 32?

Protecting Condition Variables

- Calls to pthread_cond_signal() should be protected
  - To avoid dropped signals
    - Consider a consumer acquires the lock and finds the buffer is empty, so it executes `pthread_cond_wait()`.
    - If the producer does not protect the call to `pthread_cond_signal()` with a mutex, it could insert an item into the buffer immediately after the waiting thread found it empty.
    - If the producer then signals the buffer is nonempty before the waiting thread executes the call to `pthread_cond_wait()`, the signal will be dropped and the consumer thread will not realize that the buffer is actually not empty.

Time ↓	Signaling thread	Waiting thread
		`while(put==get)`
	`insert(item);`	
	`pthread_cond_signal(&nonempty);`	
	`// Signal is dropped`	
		`pthread_cond_wait(&nonempty, lock);`
		`// Will wait forever`
	producer	consumer

Thread-Specific Data

- Goal: associate some data with a thread
- Choices
  - Pass data as arguments to functions
  - Store data in a shared array indexed by thread id
  - Use thread-specific data API
- API

```
int pthread_key_create(pthread_key_t *key,
    void (*destr_function) (void *));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key,
                        const void *pointer);
void * pthread_getspecific(pthread_key_t key);
```

Example: Thread-Specific Data

```
/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;


/* Free each thread-specific buffer when associated
thread terminates */
static void buffer_destroy(void * buf){
    free(buf);
}


int main(){
    /* Initialize key, only once */
    pthread_key_create(&buffer_key, buffer_destroy);
    ...
```

Example: Thread-Specific Data

```
void* thread_func(){
    /* get the thread-specific buffer */
    if( (buf = pthread_getspecific(buffer_key)) == NULL){
        /* Allocate a thread-specific buffer */
        buf = malloc(100);
        pthread_setspecific(buffer_key, buf);
    }
    memcpy(buf, …) // do some work
}

int main(){
    ...
    /* delete key, only once */
    pthread_key_delete(&buffer_key);
}
```

Other Useful APIs in Pthread

- Read-write lock
- Spinlock
- Barrier

- Caveat
  - Not all Pthread libraries provide those
  - One can implement those functions using mutexs, condition variables, and other variables

Read-Write Locks

- Useful for applications having a frequently read but infrequently written data structure
- Provide a critical section that
  - Multiple reader can be in the critical section simultaneously
    - Increasing concurrency of execution
  - One writer can be in the critical section at a time
    - Avoiding race condition

Pthread Read-Write Mutex API

- Initialize and destroy a rwlock
  - pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER
  - pthread_rwlock_init(pthread_rwlock_t *rwlock, pthread_rwlockattr_t *attr)
  - pthread_rwlock_destroy(pthread_rwlock_t* rwlock)

- Read locking
  - pthread_rwlock_rdlock(pthread_rwlock_t* rwlock)

- Write locking
  - pthread_rwlock_wrlock(pthread_rwlock_t* rwlock)

- Unlocking
  - pthread_rwlock_unlock(pthread_rwlock_t* rwlock)

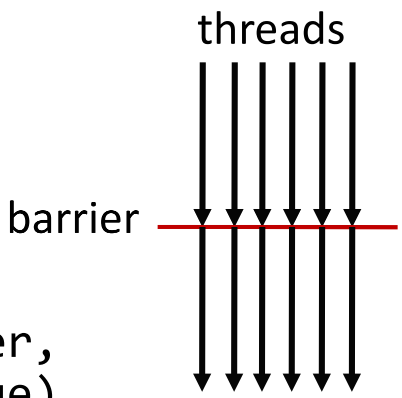
Pthread Spinlock API

- Pthread mutex is blocking lock
  - Inefficient when a critical section is short
    - Due to management cost for blocking/waking-up a thread
- Busy-waiting lock (e.g., spinlock)
  - More efficient when a critical section is short
    - Spinning a few cycles is faster than blocking and waking up a thread
- Pthread library provides spinlock-based mutual exclusion

```
pthread_spinlock_t
pthread_spin_init(pthread_spinlock_t* spinlock, int nr_shared)
pthread_spin_lock(pthread_spinlock_t* spinlock)
pthread_spin_unlock(pthread_spinlock_t* spinlock)
```

Pthread Barrier API

- Barrier
  - An execution synchronization point of threads
  - Wait until all thread reach the point



- API
  - `pthread_barrier_init(pthread_barrier_t* barrier, pthread_barrierattr_t* attr, int value)`
    - Initialize a barrier
    - The integer value specifies the number of threads to synchronize
  - `pthread_barrier_wait(pthread_barrier_t* barrier)`
    - Waits until the specified number of threads arrives at the barrier