HIGH PERFORMANCE COMPUTING: MODELS, METHODS, & MEANS

Pthreads
Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
Opening Remarks

• We now have a good picture of supercomputer architecture
  – including SMP structures
    • which are the building blocks of most HPC systems on the Top-500 List

• We were introduced to the first two programming methods for exploiting parallelism
  – Capacity Computing - Condor
  – Co-operative Computing - MPI

• Now we explore a 3rd programming model: multithreaded computing on shared memory systems
  – This time: general principles and POSIX Pthreads
  – Next time: OpenMP
What you’ll Need to Know

• Modeling time to execution with CPI
• Multi-thread programming and execution concepts
  – Parallelism with multiple threads
  – Synchronization
  – Memory consistency models
• Basic Pthread commands
• Dangers
  – Race conditions
  – Deadlock
Topics

• Introduction
• **Performance: CPI and memory behavior**
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
CPI

cpi \equiv \text{cycles per instruction}

cpi_R \equiv \text{cpi for register operations}

cpi_M \equiv \text{cpi for memory operations}

cpi_{M\text{hit}} \equiv \text{cpi for memory operations with cache hit}

cpi_{M\text{miss}} \equiv \text{cpi for memory operations with cache miss (miss penalty)}

#I \equiv \text{number of executed instructions}

#I_R \equiv \text{number of executed register instructions}

#I_M \equiv \text{number of executed memory instructions}

t_c \equiv \text{cycle time}

T \equiv \text{execution time}

T_R \equiv \text{execution time for register instructions}

T_M \equiv \text{execution time for memory instructions}

r_{\text{miss}} \equiv \text{cache miss rate}
CPI (continued)

\[ T = \# I \times cpi \times t_c \]

\[ m_R \equiv \# I_R / \# I \]

\[ m_M \equiv \# I_M / \# I \]

\[ cpi = m_R \times cpi_R + m_M \times cpi_M \text{ where } m_R + m_M = 1.0 \]

\[ cpi_M = \left(1 - r_{miss}\right) \times cpi_{Mhit} + r_{miss} \times cpi_{Mmiss} \]

\[ T = \# I \times \left(m_R \times cpi_R + m_M \times \left(\left(1 - r_{miss}\right) \times cpi_{Mhit} + r_{miss} \times cpi_{Mmiss}\right)\right) \times t_c \]
An Example

Robert hates parallel computing and runs all of his jobs on a single processor core on his Acme computer. His current application plays solitaire because he is too lazy to flip the cards himself. The machine he is running on has a 2 GHz clock. For this problem the basic register operations make up only 75% of the instruction mix but delivers one and a half instructions per cycle while the load and store operations yield one per cycle. But his cache hit rate is only 80% and the average penalty for not finding data in the L1 cache is 120 nanoseconds. A counter on the Acme processor tells Robert that it takes approximately 16 billion instruction executions to run his short program. How long does it take to execute Robert’s application?
And the answer is …

\# I = 16,000,000,000

\textit{clock\_rate} = 2.0 \text{ GHz} \Rightarrow t_c = 0.5 \text{ nanoseconds}

r_{hit} = 0.8 = \left(1 - r_{miss}\right) \Rightarrow r_{miss} = 0.2

cpi_R = \frac{2}{3}

cpi_{Mhit} = 1

cpi_{Mmiss} = \left(2 \text{ cycles/ns}\right) \times \left(120 \text{ ns}\right) = 240 \text{ cycles}

m_R = 0.75 \Rightarrow m_M = 0.25

T = 1.6 \times 10^{10} \left(\left(0.75 \times \frac{2}{3}\right) + 0.25 \times \left(0.8 \times 1 + 0.2 \times 240\right)\right) \times 5 \times 10^{-10}

T = 1.6 \times 10^{10} \left(0.5 + 12.2\right) \times 5 \times 10^{-10} = 8 \times 12.7 = \boxed{101.6 \text{ seconds}}
Topics

• Introduction
• Performance: CPI and memory behavior
• **Overview of threaded execution model**
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
UNIX Processes vs. Multithreaded Programs

Standard UNIX process (single-threaded)

New process spawned via fork()

Copy of PID’s Address Space

Multithreaded Application
Anatomy of a Thread

Thread (or, more precisely: *thread of execution*) is typically described as a *lightweight process*. There are, however, significant differences in the way standard processes and threads are created, how they interact and access resources. Many aspects of these are implementation dependent.

Private state of a thread includes:

- Execution state (instruction pointer, registers)
- Stack
- Private variables (typically allocated on thread’s stack)

Threads share access to global data in application’s address space.
Topics

- Introduction
- Performance: CPI and memory behavior
- Overview of threaded execution model
- **Programming with threads: basic concepts**
- Shared memory consistency models
- Pitfalls of multithreaded programming
- Thread implementations: approaches and issues
- Pthreads: concepts and API
- Summary
Race Conditions

**Race condition** (or **race hazard**) is a flaw in system or process whereby the output of the system or process is unexpectedly and critically dependent on the sequence or timing of other events.

**Example:** consider the following piece of pseudo-code to be executed concurrently by threads T1 and T2 (the initial value of memory location A is x)

A→R: read memory location A into register R
R++: increment register R
A←R: write R into memory location A

**Scenario 1:**

Step 1) T1:(A→R) → T1:R=x
Step 2) T1:(R++) → T1:R=x+1
Step 3) T1:(A←R) → T1:A=x+1
Step 4) T2:(A→R) → T2:R=x+1
Step 5) T2:(R++) → T2:R=x+2
Step 6) T2:(A←R) → T2:A=x+2

**Scenario 2:**

Step 1) T1:(A→R) → T1:R=x
Step 2) T2:(A→R) → T2:R=x
Step 3) T1:(R++) → T1:R=x+1
Step 4) T2:(R++) → T2:R=x+1
Step 5) T1:(A←R) → T1:A=x+1
Step 6) T2:(A←R) → T2:A=x+1

Since threads are scheduled arbitrarily by an external entity, the lack of explicit synchronization may cause different outcomes.

**Critical Sections**

**Critical section** is a segment of code accessing a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

The implementation of critical section must prevent any change of processor control once the execution enters the critical section.

- Code on uniprocessor systems may rely on disabling interrupts and avoiding system calls leading to context switches, restoring the interrupt mask to the previous state upon exit from the critical section.
- General solutions rely on synchronization mechanisms (hardware-assisted when possible), discussed on the next slides.

Thread Synchronization Mechanisms

• Based on atomic memory operation (require hardware support)
  – Spinlocks
  – Mutexes (and condition variables)
  – Semaphores
  – Derived constructs: monitors, rendezvous, mailboxes, etc.

• Shared memory based locking
  – Dekker’s algorithm
    http://en.wikipedia.org/wiki/Dekker%27s_algorithm
  – Peterson’s algorithm
    http://en.wikipedia.org/wiki/Peterson%27s_algorithm
  – Lamport’s algorithm
    http://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm
Spinlocks

- **Spinlock** is the simplest kind of lock, where a thread waiting for the lock to become available repeatedly checks lock’s status.
- Since the thread remains active, but doesn’t perform a useful computation, such a lock is essentially *busy-waiting*, and hence generally wasteful.
- Spinlocks are desirable in some scenarios:
  - If the waiting time is short, spinlocks save the overhead and cost of context switches, required if other threads have to be scheduled instead.
  - In real-time system applications, spinlocks offer good and predictable response time.
- Typically use fair scheduling of threads to work correctly.
- Spinlock implementations require atomic hardware primitives, such as *test-and-set*, *fetch-and-add*, *compare-and-swap*, etc.

Mutexes

- **Mutex** (abbreviation for *mutual exclusion*) is an algorithm used to prevent concurrent accesses to a common resource. The name also applies to the program object which negotiates access to that resource.
- Mutex works by atomically setting an internal flag when a thread (mutex owner) enters a critical section of the code. As long as the flag is set, no other threads are permitted to enter the section. When the mutex owner completes operations within the critical section, the flag is (atomically) cleared.

Condition Variables

- **Condition variables** are frequently used in association with mutexes to increase the efficiency of execution in multithreaded environments.

- Typical use involves a thread or threads waiting for a certain condition (based on the values of variables inside the critical section) to occur. Note that:
  - The thread cannot wait inside the critical section, since no other thread would be permitted to enter and modify the variables.
  - The thread could monitor the values by repeatedly accessing the critical section through its mutex; such a solution is typically very wasteful.

- Condition variable permits the waiting thread to temporarily release the mutex it owns, and provide the means for other threads to communicate the state change within the critical section to the waiting thread (if such a change occurred).

```c
/* waiting thread code: */
lock(mutex);
/* check if you can progress */
while (condition not true)
  wait(cond_var);
/* now you can; do your work */
...
unlock(mutex);

/* modifying thread code: */
lock(mutex);
/* update critical section variables */
...
/* announce state change */
signal(cond_var);
unlock(mutex);
```
Semaphores

- **Semaphore** is a protected variable introduced by Edsger Dijkstra (in the “THE” operating system) and constitutes the classic method for restricting access to shared resource.
- It is associated with an integer variable (semaphore’s value) and a queue of waiting threads.
- Semaphore can be accessed only via the atomic $P$ and $V$ primitives:

```c
P(semaphore S) {
    if S.v > 0 then S.v := S.v-1;
    else {
        insert current thread in S.q;
        change its state to blocked;
        schedule another thread;
    }
}
```

```c
V(semaphore S) {
    if S.v = 0 and not empty(S.q)
        then {
            pick a thread T from S.q;
            change T’s state to ready;
        }
    else S.v := S.v+1;
}
```

- Usage:
  - Semaphore’s value $S.v$ is initialized to a positive number.
  - Semaphore’s queue $S.q$ is initially empty.
  - Entrance to critical section is guarded by $P(S)$.
  - When exiting critical section, $V(S)$ is invoked.
  - Note: mutex can be implemented as a binary semaphore.

Suggested reading:
- [Semaphores](http://www.mcs.drexel.edu/~shartley/OSusingSR/semaphores.html)
- [Semaphore (programming)](http://en.wikipedia.org/wiki/Semaphore_(programming))
Disadvantages of Locks

- Blocking mechanism (forces threads to wait)
- Conservative (lock has to be acquired when there’s only a possibility of access conflict)
- Vulnerable to faults and failures (what if the owner of the lock dies?)
- Programming is difficult and error prone (deadlocks, starvation)
- Does not scale with problem size and complexity
- Require balancing the granularity of locked data against the cost of fine-grain locks
- Not composable
- Suffer from priority inversion and convoying
- Difficult to debug

Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
Shared Memory Consistency Model

• Defines memory functionality related to read and write operations by multiple processors
  – Determines the order of read values in response to the order of write values by multiple processors
  – Enables the writing of correct, efficient, and repeatable shared memory programs

• Establishes a formal discipline that places restrictions on the values that can be returned by a read in a shared-memory program execution
  – Avoids non-determinacy in memory behavior
  – Provides a programmer perspective on expected behavior
  – Imposes demands on system memory operation

• Two general classes of consistency models:
  – Sequential consistency
  – Relaxed consistency
Sequential Consistency Model

• Most widely adopted memory model

• Required:
  – Maintaining program order among operations from individual processors
  – Maintaining a single sequential order among operations from all processors

• Enforces effect of atomic complex memory operations
  – Enables compound atomic operations
  – Avoids race conditions
  – Precludes non-determinacy from dueling processors
Relaxed Consistency Models

• Sequential consistency over-constrains parallel execution limiting parallel performance and scalability
  – Critical sections impose sequential bottlenecks
  – Amdahl’s Law applies imposing upper bound on performance
• Relaxed consistency models permit optimizations not possible under limitations of sequential consistency
• Forms of relaxed consistency
  – Program order
    • Write to read
    • Write to write
    • Read to following read or write
  – Write atomicity
    • Read value of its own previous write prior to being visible to all other processors
Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
Dining Philosophers Problem

A variation on Edsger Dijkstra’s five computers competing for access to five shared tape drives problem (introduced in 1971), retold by Tony Hoare.

**Description:**
- N philosophers (N > 3) spend their time eating and thinking at the round table
- There are N plates and N forks (or chopsticks, in some versions) between the plates
- Eating requires two forks, which may be picked one at a time, at each side of the plate
- When any of the philosophers is done eating, he starts thinking
- When a philosopher becomes hungry, he attempts to start eating
- They do it in complete silence as to not disturb each other (hence no communication to synchronize their actions is possible)

**Problem:**
How must they acquire/release forks to ensure that each of them maintains a healthy balance between meditation and eating?
What Can Go Wrong at the Philosophers Table?

- **Deadlock**
  If all philosophers decide to eat at the same time and pick forks at the same side of their plates, they are stuck forever waiting for the second fork.

- **Livelock**
  Livelock frequently occurs as a consequence of a poorly thought out deadlock prevention strategy. Assume that all philosophers: (a) wait some length of time to put down the fork they hold after noticing that they are unable to acquire the second fork, and then (b) wait some amount of time to reacquire the forks. If they happen to get hungry at the same time and pick one fork using scenario leading to a deadlock and all (a) and (b) timeouts are set to the same value, they won’t be able to progress (even though there is no actual resource shortage).

- **Starvation**
  There may be at least one philosopher unable to acquire both forks due to timing issues. For example, his neighbors may alternately keep picking one of the forks just ahead of him and take advantage of the fact that he is forced to put down the only fork he was able to get hold of due to deadlock avoidance mechanism.
Priority Inversion

Priority inversion is the scenario where a low priority thread holds a shared resource that is required by a high priority thread.

• How it happens:
  – A low priority thread locks the mutex for some shared resource
  – A high priority thread requires access to the same resource (waits for the mutex)
  – In the meantime, a medium priority thread (not depending on the common resource) gets scheduled, preempting the low priority thread and thus preventing it from releasing the mutex

• A classic occurrence of this phenomenon lead to system reset and subsequent loss of data in Mars Pathfinder mission in 1997: http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

Spurious Wakeups

- Spurious wakeup is a phenomenon associated with a thread waiting on a condition variable.
- In most cases, such a thread is supposed to return from call to `wait()` only if the condition variable has been signaled or broadcast.
- Occasionally, the waiting thread gets unblocked unexpectedly, either due to thread implementation performance trade-offs, or scheduler deficiencies.
- Lesson: upon exit from `wait()`, test the predicate to make sure the waiting thread indeed may proceed (i.e., the data it was waiting for have been provided). The side effect is a more robust code.

Thread Safety

A code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads.

- **Indicators helpful in determining thread safety**
  - How the code accesses global variables and heap
  - How it allocates and frees resources that have global limits
  - How it performs indirect accesses (through pointers or handles)
  - Are there any visible side effects

- **Achieving thread safety**
  - **Re-entrancy**: property of code, which may be interrupted during execution of one task, reentered to perform another, and then resumed on its original task without undesirable effects
  - Mutual exclusion: accesses to shared data are serialized to ensure that only one thread performs critical state update. Acquire locks in an identical order on all threads
  - Thread-local storage: as much of the accessed data as possible should be placed in thread’s private variables
  - Atomic operations: should be the preferred mechanism of use when operating on shared state
Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• **Thread implementations: approaches and issues**
• Pthreads: concepts and API
• Summary
Common Approaches to Thread Implementation

- Kernel threads
- User-space threads
- Hybrid implementations

References:
Kernel Threads

• Also referred to as *Light Weight Processes*
• Known to and individually managed by the kernel
• Can make system calls independently
• Can run in parallel on a multiprocessor (map directly onto available execution hardware)
• Typically have wider range of scheduling capabilities
• Support preemptive multithreading natively
• Require kernel support and resources
• Have higher management overhead
User-space Threads

- Also known as fibers or coroutines
- Operate on top of kernel threads, mapped to them via user-space scheduler
- Thread manipulations (“context switches”, etc.) are performed entirely in user space
- Usually scheduled cooperatively (i.e., non-preemptively), complicating the application code due to inclusion of explicit processor yield statements
- Context switches cost less (on the order of subroutine invocation)
- Consume less resources than kernel threads; their number can be consequently much higher without imposing significant overhead
- Blocking system calls present a challenge and may lead to inefficient processor usage (user-space scheduler is ignorant of the occurrence of blocking; no notification mechanism exists in kernel either)
MxN Threading

- Available on NetBSD, HPUX, and Solaris to complement the existing 1x1 (kernel threads only) and Mx1 (multiplexed user threads) libraries.
- Multiplex M lightweight user-space threads on top of N kernel threads, M > N (sometimes M >> N).
- User threads are unbound and scheduled on Virtual Processors (which in turn execute on kernel threads); user thread may effectively move from one kernel thread to another in its lifetime.
- In some implementations, Virtual Processors rely on the concept of Scheduler Activations to deal with the issue of user-space threads blocking during system calls.
Scheduler Activations

- Developed in 1991 at the University of Washington
- Typically used in implementations involving user-space threads
- Require kernel cooperation in form of a lightweight upcall mechanism to communicate blocking and unblocking events to the user-space scheduler

- Unbound user threads are scheduled on Virtual Processors (which in turn execute on kernel threads)
- A user thread may effectively move from one kernel thread to another in its lifetime
- Scheduler Activation resembles and is scheduled like a kernel thread
- Scheduler Activation provides its replacement to the user-space scheduler when the unbound thread invokes a blocking operation in the kernel
- The new Scheduler Activation continues the operations of the same VP

Reference:
Examples of Multi-Threaded System Implementations

- The most commonly used thread package on Linux is Native POSIX Thread Library (NPTL)
  - Requires kernel version 2.6
  - 1x1 model, mapping each application thread to a kernel thread
  - Bundled by default with recent versions of glibc
  - High-performance implementation
  - POSIX (Pthreads) compliant

- Most of the prominent operating systems feature their own thread implementations, for example:
  - FreeBSD: three thread libraries, each supporting different execution model (user-space, 1x1, MxN with scheduler activations)
  - Solaris: kernel-level execution through LWPs (Lightweight Processes); user threads execute in context of LWPs and are controlled by system library
  - HP/UX: Pthreads compliant MxN implementation
  - MS Windows: threads as smallest kernel-level execution objects, fibers as smallest user-level execution objects controlled by the programmer; many-to-many scheduling supported

- There are numerous open-source thread libraries (mostly for Linux): LinuxThreads, GNU Pth, Bare-Bone Threads, FSU Pthreads, DCEthreads, Nthreads, CLthreads, PCthreads, LWP, QuickThreads, Marcel, etc.
Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
POSIX Threads (Pthreads)

• POSIX Threads define POSIX standard for multithreaded API (IEEE POSIX 1003.1-1995)

• The functions comprising core functionality of Pthreads can be divided into three classes:
  – Thread management
  – Mutexes
  – Condition variables

• Pthreads define the interface using C language types, function prototypes and macros

• Naming conventions for identifiers:
  – `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

References:
2. [http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html](http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html)
Programming with Pthreads

The scope of this short tutorial is:

• General thread management
• Synchronization
  – Mutexes
  – Condition variables
• Miscellaneous functions
Pthreads: Thread Creation

Function: `pthread_create()`

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

Description:
Creates a new thread within a process. The created thread starts execution of `routine`, which is passed a pointer argument `arg`. The attributes of the new thread can be specified through `attr`, or left at default values if `attr` is null. Successful call returns 0 and stores the id of the new thread in location pointed to by `thread`, otherwise an error code is returned.

```c
#include <pthread.h>
...
void *do_work(void *input_data)
{ /* this is thread’s starting routine */
  ...
  ...
  pthread_t id;
  struct {. . .} args = {. . .}; /* struct containing thread arguments */
  int err;
  /* create new thread with default attributes */
  err = pthread_create(&id, NULL, do_work, (void *)&args);
  if (err != 0) { /* handle thread creation failure */
    ...
  }
```

Pthreads: Thread Join

Function:  pthread_join()

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

Description:
Suspends the execution of the calling thread until the target thread terminates (either by returning from its startup routine, or calling pthread_exit()), unless the target thread already terminated. If value_ptr is not null, the return value from the target thread or argument passed to pthread_exit() is made available in location pointed to by value_ptr. When pthread_join() returns successfully (i.e. with zero return code), the target thread has been terminated.

```
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
void *result_ptr;
int err;
...
/* create worker thread */
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_join(id, &result_ptr);
if (err != 0) {/* handle join error */}
else {/* the worker thread is terminated and result_ptr points to its return value */
    ...
}
Pthreads: Thread Exit

Function: `pthread_exit()`

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

Description:
Terminates the calling thread and makes the `value_ptr` available to any successful join with the terminating thread. Performs cleanup of local thread environment by calling cancellation handlers and data destructor functions. Thread termination does not release any application visible resources, such as mutexes and file descriptors, nor does it perform any process-level cleanup actions.

```c
#include <pthread.h>
...
void *do_work(void *args)
{
  ...
  pthread_exit(&return_value);
  /* the code following pthread_exit is not executed */
  ...
}
...
void *result_ptr;
pthread_t id;
pthread_create(&id, NULL, do_work, (void *)&args);
...
pthread_join(id, &result);
/* result_ptr now points to return_value */
...
Pthreads: Thread Termination

Function: `pthread_cancel()`

```c
void pthread_cancel(thread_t thread);
```

Description:
The `pthread_cancel()` requests cancellation of thread `thread`. The ability to cancel a thread is dependent on its state and type.

```c
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
thread_t id;
int err;
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_cancel(id);
if (err != 0) {/* handle cancellation failure */}
...
# Pthreads: Detached Threads

## Function: `pthread_detach()`

```c
int pthread_detach(pthread_t thread);
```

## Description:
Indicates to the implementation that storage for thread `thread` can be reclaimed when the thread terminates. If the thread has not terminated, `pthread_detach()` is not going to cause it to terminate. Returns zero on success, error number otherwise.

```c
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
pthread_t id;
int err;
...
/* start a new thread */
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_detach(id);
if (err != 0) {/* handle detachment failure */}
else {/* master thread doesn’t join the worker thread; the worker thread resources will be released automatically after it terminates */
...
}
Pthreads: Operations on Mutex Objects (I)

**Function:**

```
#include <pthread.h>
...
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
/* lock the mutex before entering critical section */
pthread_mutex_lock(&mutex);
/* critical section code */
...
/* leave critical section and release the mutex */
pthread_mutex_unlock(&mutex);
...```

**Description:**

The mutex object referenced by `mutex` shall be locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. After successful return from the call, the mutex object referenced by `mutex` is in locked state with the calling thread as its owner.

The mutex object referenced by `mutex` is released by calling `pthread_mutex_unlock()`. If there are threads blocked on the mutex, scheduling policy decides which of them shall acquire the released mutex.
Pthreads: Operations on Mutex Objects (II)

Function: pthread_mutex_trylock()

```c
def pthread_mutex_t __thread_mutex_lock(pthread_mutex_t *mutex)
```

Description:
The function `pthread_mutex_trylock()` is equivalent to `pthread_mutex_lock()` but if the mutex object is currently locked, the call returns immediately with an error code EBUSY. The value of 0 (success) is returned only if the mutex has been acquired.

```c
#include <pthread.h>
...
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int err;
...
/* attempt to lock the mutex */
err = pthread_mutex_trylock(&mutex);
switch (err) {
    case 0: /* lock acquired; execute critical section code and release mutex */
        ...
        pthread_mutex_unlock(&mutex);
        break;
    case EBUSY: /* someone already owns the mutex; do something else instead of blocking */
        ...
        break;
    default: /* some other failure */
        ...
        break;
}
Pthread Mutex Types

• Normal
  – No deadlock detection on attempts to relock already locked mutex

• Error-checking
  – Error returned when locking a locked mutex

• Recursive
  – Maintains lock count variable
  – After the first acquisition of the mutex, the lock count is set to one
  – After each successful relock, the lock count is increased; after each unlock, it is decremented
  – When the lock count drops to zero, thread loses the mutex ownership

• Default
  – Attempts to lock the mutex recursively result in an undefined behavior
  – Attempts to unlock the mutex which is not locked, or was not locked by the calling thread, results in undefined behavior
Pthreads: Condition Variables

**Function:**
- `pthread_cond_wait()`,
- `pthread_cond_signal()`,
- `pthread_cond_broadcast()`

```
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);
```

**Description:**
The `pthread_cond_wait()` blocks on a condition variable associated with a mutex. The function must be called with a locked mutex argument. It atomically releases the mutex and causes the calling thread to block. While in that state, another thread is permitted to access the mutex. Subsequent mutex release should be announced by the accessing thread through `pthread_cond_signal()` or `pthread_cond_broadcast()`. Upon successful return from `pthread_cond_wait()`, the mutex is in locked state with the calling thread as its owner.

The `pthread_cond_signal()` unblocks at least one of the threads that are blocked on the specified condition variable `cond`.

The `pthread_cond_broadcast()` unblocks all threads currently blocked on the specified condition variable `cond`.

All of these functions return zero on successful completion, or an error code otherwise.
Example: Condition Variable

Initialization and startup

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* create default mutex */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; /* create default condition variable */
pthread_t prod_id, cons_id;
item_t buffer; /* storage buffer (shared access) */
int empty = 1; /* buffer empty flag (shared access) */
...
pthread_create(&prod_id, NULL, producer, NULL); /* start producer thread */
pthread_create(&cons_id, NULL, consumer, NULL); /* start consumer thread */
...
```

Simple producer thread

```c
void *producer(void *none) {
    while (1) {
        /* obtain next item, asynchronously */
        item_t item = compute_item();
        pthread_mutex_lock(&mutex);
        /* critical section starts here */
        while (!empty)
            /* wait until buffer is empty */
            pthread_cond_wait(&cond, &mutex);
        /* store item, update status */
        buffer = item;
        empty = 0;
        /* wake waiting consumer (if any) */
        pthread_condition_signal(&cond);
        /* critical section done */
        pthread_mutex_unlock(&mutex);
    }
}
```

Simple consumer thread

```c
void *consumer(void *none) {
    while (1) {
        item_t item;
        pthread_mutex_lock(&mutex);
        /* critical section starts here */
        while (empty)
            /* block (nothing in buffer yet) */
            pthread_cond_wait(&cond, &mutex);
        /* grab item, update buffer status */
        item = buffer;
        empty = 1;
        /* critical section done */
        pthread_condition_signal(&cond);
        pthread_mutex_unlock(&mutex);
        /* process item, asynchronously */
        consume_item(item);
    }
}
```
Pthreads: Dynamic Initialization

Function: `pthread_once()`

```c
int pthread_once(pthread_once_t *control, void (*init_routine)(void));
```

Description:
The first call to `pthread_once()` by any thread in a process will call the `init_routine()` with no arguments. Subsequent calls to `pthread_once()` with the same control will not call `init_routine()`.

```c
#include <pthread.h>
...
pthread_once init_ctrl = PTHREAD_ONCE_INIT;
...
void initialize() {/* initialize global variables */}
...
void *do_work(void *arg)
{ /* make sure global environment is set up */
  pthread_once(&init_ctrl, initialize);
  /* start computations */
  ...
}
...
pthread_t id;
pthread_create(&id, NULL, do_work, NULL);
...
Pthreads: Get Thread ID

Function:  pthread_self()

pthread_t pthread_self(void);

Description:
Returns the thread ID of the calling thread.

```c
#include <pthread.h>
...
pthread_t id;
id = pthread_self();
...```

Topics

• Introduction
• Performance: CPI and memory behavior
• Overview of threaded execution model
• Programming with threads: basic concepts
• Shared memory consistency models
• Pitfalls of multithreaded programming
• Thread implementations: approaches and issues
• Pthreads: concepts and API
• Summary
Summary – Material for the Test

• Performance & cpi: slide 8
• Multi thread concepts: 13, 16, 18, 19, 22, 24, 31
• Thread implementations: 35 – 37
• Pthreads: 43 – 45, 48