An Overview of Multi-threading Mechanisms

Douglas C. Schmidt
Washington University, St. Louis

http://www.cs.wustl.edu/~schmidt/schmidt@cs.wustl.edu

Motivation for Concurrency

- Concurrent programming is increasing relevant to:
  - Leverage hardware/software advances
    - e.g., multi-processors and OS thread support
  - Increase performance
    - e.g., overlap computation and communication
  - Improve response-time
    - e.g., GUIs and network servers
  - Simplify program structure
    - e.g., synchronous vs. asynchronous network IPC

Definitions

- **Concurrency**
  - “Logically” simultaneous processing
  - Does not imply multiple processing elements

- **Parallelism**
  - “Physically” simultaneous processing
  - Involves multiple processing elements and/or independent device operations

- Both concurrency and parallelism require controlled access to shared resources
  - e.g., I/O devices, files, database records, in-core data structures, consoles, etc.

Concurrency vs. Parallelism
Concurrency Overview

- A thread of control is a single sequence of execution steps performed in one or more programs
  - One program → standalone systems
  - More than one program → distributed systems

- Traditional OS processes contain a single thread of control
  - This simplifies programming since a sequence of execution steps is protected from unwanted interference by other execution sequences...

Traditional Approaches to OS Concurrency

1. Device drivers and programs with signal handlers utilize a limited form of concurrency
   - e.g., asynchronous I/O
   - Note that concurrency encompasses more than multi-threading...

2. Many existing programs utilize OS processes to provide “coarse-grained” concurrency
   - e.g.,
     - Client/server database applications
     - Standard network daemons like UNIX inetd
   - Multiple OS processes may share memory via memory mapping or shared memory and use semaphores to coordinate execution
   - The OS kernel scheduler dictates process behavior

Evaluating Traditional OS Process-based Concurrency

- Advantages
  - Easy to keep processes from interfering
    - A process combines security, protection, and robustness

- Disadvantages
  1. Complicated to program, e.g.,
     - Signal handling may be tricky
     - Shared memory may be inconvenient
  2. Inefficient
     - The OS kernel is involved in synchronization and process management
     - Difficult to exert fine-grained control over scheduling and priorities

Modern OS Concurrency

- Modern OS platforms typically provide a standard set of APIs that handle
  1. Process/thread creation and destruction
  2. Various types of process/thread synchronization and mutual exclusion
  3. Asynchronous facilities for interrupting long-running processes/threads to report errors and control program behavior

- Once the underlying concepts are mastered, it’s relatively easy to learn different concurrency APIs
  - e.g., traditional UNIX process operations, Solaris threads, POSIX pthreads, WIN32 threads, etc.
Lightweight Concurrency

- Modern OSs provide lightweight mechanisms that manage and synchronize multiple threads within a process
  - Some systems also allow threads to synchronize across multiple processes

Benefits of threads

1. Relatively simple and efficient to create, control, synchronize, and collaborate
   - Threads share many process resources by default

2. Improve performance by overlapping computation and communication
   - Threads may also consume less resources than processes

3. Improve program structure
   - e.g., compared with using asynchronous I/O

Single-threaded vs. Multi-threaded RPC

Hardware and OS Concurrency Support

- Modern OS platforms like Solaris provide kernel support for multi-threading

Kernel Abstractions

- Kernel threads
  - The "fundamental scheduling entities" executed by the PE(s)
  - Operate in kernel space
  - Kernel-resident subsystems use kernel threads directly

- Lightweight processes (LWP)
  - Every LWP is associated with one kernel thread
    - i.e., 1-to-1 mapping between kernel thread and LWP per-process
  - Not every kernel thread has an LWP
    - "System threads" (e.g., pagedaemon, NFS daemon, and the callout thread) have only a kernel thread
Application Abstractions

- **Application threads**
  - LWPs(s) can be thought of as ‘virtual CPUs’ on which application threads are scheduled and multiplexed
  - Each application thread has its own stack
    - However, it shares its process address space with other threads
  - Application threads are “logically” independent
  - Multiple application threads running on separate LWPs can execute simultaneously (even system calls and page faults…)
    - Assuming a multi-CPU system or async I/O

Kernel-level vs. User-level Threads

- Application and system characteristics influence the choice of kernel-level vs. user-level threading
  - *e.g.*,
    - High degree of “virtual” application concurrency implies user-level threads (*i.e.*, unbound threads)
      - *e.g.*, desktop windowing system
    - High degree of “real” application parallelism implies lightweight processes (LWPs) (*i.e.*, bound threads)
  - In addition, LWPs must be used for:
    - Real-time scheduling class
    - Give thread alternative signal stack
    - Give thread a unique alarm or timer

Performance Considerations

- Performance of different combinations of application-level vs. kernel-level threads is influenced various factors, *e.g.*
  - Number of PEs
  - Inter-thread communication
  - Inter-thread synchronization
  - Amount of context switching
- It is important to consider the “process architecture” of a multi-threaded application

Scheduling Classes in SunOS 5.x

- There are three classes of process (LWP) scheduling in SunOS 5.x
  - **Real-time**
    - Highest priority, the scheduler always dispatches the highest priority real-time LWP
  - **System**
    - Middle priority
      - Cannot be applied to a user process
  - **Timesharing (default)**
    - Lowest priority, provides fair distribution of process resources
- A new process inherits the scheduling class and priority of its parent
Application Thread Overview

- A multi-threaded process contains one or more threads of control

- Each thread may be executed independently and asynchronously
  - Different threads may have different priorities
  - System calls may be made independently, page faults handled separately, etc.
  - Some system calls affect the process
    - e.g., exit
  - Other system calls affect only the calling thread
    - e.g., read/write

- Threads in a process are generally invisible to other processes

Thread Resources

- Most process resources are equally accessible to all threads in the process, e.g.,
  * Virtual memory
  * User permissions and access control privileges
  * Open files
  * Signal handlers

- In addition, each thread contains unique information, e.g.,
  * Identifier
  * Register set (including PC and SP)
  * Stack
  * Signal mask
  * Priority
  * Thread-specific data (e.g., errno)

- Note, there is no MMU protection for separate threads within a single process...

LWP Characteristics

- The threads library uses execution resources called LWPs
  - LWPs are scheduled on top of kernel threads (and PEs) by the OS
  - Likewise, the threads library schedules “unbound” runnable threads on the LWP execution resources
    - This typically does not involve the kernel

- In order to expedite thread operations, LWPs contain certain information that application threads do not have, e.g.,
  - Scheduling class
    - e.g., Real-time vs. system vs. timesharing
  - Alarms
  - Interval timers
  - Profiling buffers

Programming LWPs

- The threads library ensures that there are enough LWPs to enable a program to make progress
  - i.e., LWPs may be allocated/deallocated as needed via SIGWAIT signal sent by kernel

- The thr_setconcurrency library function provides additional control
  - Note, it is only a hint...

- Note, there is also a low-level interface to the LWP facilities
  - Application programmers typically do not use this interface directly
Thread Creation

- Thread creation is handled via the `thr_create` function:
  
  ```c
  int thr_create (void *stack_base, size_t stack_size, 
  void *(*start_routine)(void *), void *arg, long 
  flags, thread_t *new_thread);
  ```

  - `thr_create` creates and starts a new thread using the `start_routine` function specified in the call
    - Returns 0 on success and non-0 on failure
  - The identity of the thread is returned to the caller
    - A thread ID is only valid within a single process
    - There is no thread 0...
  - The caller may supply a stack or if a NULL is used the library allocates a default stack

Thread Creation (cont'd)

- `thr_create` (cont'd)
  - Each application thread gets its own stack
    - You may specify a size for the stack or use the default
      - The default is 1 Megabyte of virtual memory, with no reserved stack space
  - size_t `thr_min_stack (void)`
    - The size of any stack must be larger than the value of this function call
  - Each stack area is protected with unallocated memory
    - Thus, if your process overflows the stack a bus error (SIGBUS) will occur

Thread Creation (cont'd)

- `thr_create` flags include
  - THR_SUSPENDED
    - The new thread is created suspended and will not execute the `start_routine` function until it is started by `thr_continue`
  - THR_DETACHED
    - The new thread is created detached and thread ID and other resources may be reused as soon as the thread terminates
  - THR_BOUND
    - The new thread is created permanently bound to an LWP

Thread Creation (cont'd)

- `thr_create` flags include
  - THR_NEWLWP
    - The desired concurrency level for unbound threads is increased by one, typically by adding a new LWP to the pool of LWPs running unbound threads
  - THR_DAEMON
    - The thread is marked as a daemon and the process will exit when all non-daemon threads exit
      - i.e., daemon threads are not counted in the process exit criteria
Differences Between fork and thr_create

- thr_create normally allocates a thread stack out of the cache, initializes some fields, and places the thread on the per-process run queue
  - Typically this is not very many instructions, none of them in the kernel
  - The thread will then be run by a CPU when a kernel LWP next checks that queue

- fork is quite a bit more heavy weight
  - It creates more new kernel resources than just a new address space

Thread Exit

- The thr_exit function terminates the invoking thread and sets the exit status to the specified value
  - void thr_exit (void *status);
  - If the thread was not detached, its identifier and status are retained until thr_join is called via another thread
  - If there are no remaining threads, the process is exited with a 0 exit status...

- The thr_self function returns the thread identifier structure of the caller
  - thread_t thr_self (void);

Thread Join

- The thr_join function blocks until the specified thread exits
  - int thr_join (thread_t wait_for, thread_t *departed, void **status);
  - If wait_for is 0, the functions waits for any undetached thread in the process to terminate, else it waits for that wait_for thread id to terminate
  - If departed is non-NULL it points to location storing the ID of the terminated thread
  - If status is non-NULL it points to a location storing the exit status of the terminated thread
  - thr_join cannot wait for detached threads, threads in other processes, or the current thread

Thread Suspend and Resume

- The thr_suspend function immediately suspends the specified thread until it is explicitly resumed
  - int thr_suspend (thread_t target_thread);
  - Note, a suspended thread does not receive signals...

- The thr_continue function resumes execution of a suspended thread
  - int thr_continue (thread_t target_thread);
Thread Scheduling

- The scheduling of threads by the threads library is *non-preemptive*, in the traditional *time-slicing* sense...
  - However, the scheduling of LWPs by the OS is preemptive
  - Moreover, LWPs use "priority aging," whereas threads do not...

Thread Scheduling (cont’d)

- **int thr_setprio (thread_t target_thread, int priority);**
  - The priority must be $\geq 0$, with greater values indicating increased priority

- **int thr_getprio (thread_t target_thread);**
  - This function gets the thread priority of the specified thread

- **int thr_yield (void);**
  - Yields the caller's executing status to any thread with same or higher priority

Thread Concurrency

- The scheduling of threads is influenced by the following library routines
  - **int thr_setconcurrency (int new_level);**
    - Indicates the desired level of concurrency that application threads require
    - *i.e.*, number of threads that can be active simultaneously
    - *i.e.*, the number of LWPs associated with the threads library
    - Only a hint, actual number of LWPs may be more or less than number requested
  - **int thr_getconcurrency (void);**
    - Returns current number of LWPs

Synchronization Mechanisms

- Threads share resources in a process address space

- Therefore, they must use *synchronization mechanisms* to coordinate their access to shared data

- Traditional OS synchronization mechanisms are very low-level, tedious to program, error-prone, and non-portable

- ACE encapsulates these mechanisms with higher-level patterns and classes
Common OS Synchronization Mechanisms

1. **Mutual exclusion** locks
   - Serialize access to a shared resource

2. **Counting semaphores**
   - Synchronize execution

3. **Readers/writer locks**
   - Serialize access to resources whose contents are searched more than changed

4. **Condition variables**
   - Used to block until shared data changes state

5. **File locks**
   - System-wide readers/write locks access by file-name

Additional ACE Synchronization Mechanism

1. **Guards**
   - An exception-safe scoped locking mechanism

2. **Barriers**
   - Allows threads to synchronize their completion

3. **Token**
   - Provides absolute scheduling order and simplifies multi-threaded event loop integration

4. **Task**
   - Provides higher-level “active object” semantics for concurrent applications

5. **Thread-specific storage**
   - Low-overhead, contention-free storage

Concurrency Mechanisms in ACE

Solaris Synchronization Primitives

- Each synchronization facility has a set of routines that operate on instances called *synchronization variables*
  - These variables may be allocated statically or dynamically
  - Variables must be allocated in memory that is globally accessible, e.g.,
    - Allocated in global process memory and shared by multiple
      - Placed into shared memory or mapped files and accessed via separate processes
  - Depending on flags, different behavior may be selected during variable initialization
Solaris Synchronization Primitives (cont’d)

- All synchronization variables may be placed in shared memory and shared between threads running in multiple processes
  - *Intra-process* behavior vs. *inter-process* behavior is selected by using the USYNC_THREAD vs. USYNC_PROCESS flags at initialization time...

- Note that memory-mapped files may be used to provide persistent locks that are shared between processes

- If a variable is initialized to 0, the "default behavior" is selected
  - Default is local to one process (*i.e.*, USYNC_THREAD)

- Three methods for implementing locks are *spin locks, sleep locks, and adaptive locks*

Mutex Synchronization

- The simplest type of synchronization variable is the "mutex" (mutual exclusion) lock
  - *i.e.*, used to implement "critical sections"...

- Only one thread at a time may "own" a mutex lock
  - *i.e.*, used to implement "critical sections"...

- Implemented to be highly efficient, but limited in functionality
  - *e.g.*, lock/unlock operations must be “fully-bracketed”

The Mutex API

- `int mutex_init (mutex_t *mp, int type, void *arg);`

- `int mutex_destroy (mutex_t *mp);`

- `int mutex_lock (mutex_t *mp);`
  - Acquire lock ownership (wait on priority queue if necessary)

- `int mutex_trylock (mutex_t *mp);`
  - Conditionally acquire lock (*i.e.*, don’t wait on queue)

- `int mutex_unlock (mutex_t *mp);`
  - Release lock and unblock thread at head of priority queue, if necessary
  - Only the owner of a mutex may unlock it

Programming with Mutexes

- Simple resource example

  ```c
  #include <sys/systypes.h>
  #include <sys/semaphore.h>

  static mutex_t count_mutex; // Initialized to 0
  static int count;

  int increment_count (void) {
    mutex_lock (&count_mutex);
    count = count + 1; /* atomic update */
    mutex_unlock (&count_mutex);
  }

  int get_count (void) {
    int c;
    mutex_lock (&count_mutex);
    c = count; /* ensure memory synchronization...*/
    mutex_unlock (&count_mutex);
    return c;
  }
  ```
Condition Variables

- Used to “sleep/wait” until a particular condition involving shared data occurs
  - Conditions may be arbitrarily complex

- Allows more complex scheduling decisions, compared with simple mutex
  - i.e., a mutex makes other threads wait, whereas a condition variable allows a thread to make itself wait for a particular condition involving shared data
  - Usually more efficient/correct than busy waiting...

- Are always used in conjunction with a mutex lock

Condition Variable API

- `int cond_init (cond_t *cvp, int type, int arg);`

- `int cond_destroy (cond_t *cvp);`

- `int cond_wait (cond_t *cvp, mutex_t *mp);`
  - Typically used in conjunction with a “condition expression”
  - Block until condition is signaled
  - Atomically release lock before blocking
  - Atomically reacquire lock before returning
    - Necessitates retesting condition...

Condition Variable Patterns

- A particular idiom is typically associated with condition variables

```c
// Global variables
static mutex_t m; // Initialized to 0
static cond_t c; // Initialized to 0
void some_function (void)
{
    mutex_lock (&m);
    while (condition expression is not true)
        cond_wait (&c, &m);
    /* Atomically modify shared information */
    mutex_unlock (&m);
    /* ... */
}
```

- Warning!!!! Always make sure to invoke condition variable functions while holding the associated mutex lock!!!
  - Otherwise, “lost wakeup bugs” occur...

Condition Variable API

- `int cond_timedwait (cond_t *cvp, mutex_t *mp, timespec_t *abstime);`
  - Block on condition, or until absolute time-of-day has passed

- `int cond_signal (cond_t *cvp);`
  - Signal one thread blocked in cond_wait
  - If no thread is waiting, signal is ignored...

- `int cond_broadcast (cond_t *cvp);`
  - Signal all threads blocked in cond_wait
  - Use with care due to avoid the “thundering herd” problem...
  - Useful for allowing threads to contend for variable amounts of resources when resources are freed dynamically
Programming with Condition Variables

- Implement general P and V using mutex and condition vars

```c
static mutex_t count_lock;  // Initialized to 0
static cond_t count_nonzero;  // Initialized to 0
static unsigned int count;  // Initialized to 0

void P (void) {
    mutex_lock (&count_lock);
    while (count == 0) {
        cond_wait (&count_nonzero, &count_lock);
        count = count - 1;
        mutex_unlock (&count_lock);
    }
}

void V (void) {
    mutex_lock (&count_lock);
    // Order of the following lines doesn't matter
    if (count == 0) {
        cond_signal (&count_nonzero);
        count = count + 1;
        mutex_unlock (&count_lock);
    }
}
```

Programming with Condition Variables (cont’d)

- Illustration of cond_broadcast()

```c
static mutex_t rsrc_lock;  // Initialized to 0
static cond_t rsrc_add;  // Initialized to 0
static unsigned int resources, waiting;

int obtain_resources (int amount) {
    mutex_lock (&rsrc_lock);
    while (resources < amount) {
        waiting++;
        cond_wait (&rsrc_add, &rsrc_lock);
    }
    resources = amount;
    mutex_unlock (&rsrc_lock);
}

int release_resources (int amount) {
    mutex_lock (&rsrc_lock);
    resources = amount;
    if (waiting > 0) {
        waiting = 0;
        cond_broadcast (&rsrc_add);
    }
    mutex_unlock (&rsrc_lock);
}
```
Semaphores

- Semaphores are conceptually non-negative integers that may be incremented and decremented *atomically*
- They are less efficient than mutexes, but more general
  - *e.g.*, they need not be acquired and released by the same thread
    - *i.e.*, they may be used in signal handlers or other asynchronous event notification contexts
- It is not necessary to acquire a mutex lock to use a semaphore

Semaphore API

- `int sema_init (sema_t *sp, unsigned int count, int type, void *arg);`
  - `count` gives initial value of semaphore
- `int sema_destroy (sema_t *sp);`
- `int sema_wait (sema_t *sp);`
  - Block the thread until the semaphore count becomes greater than 0, then decrement it
- `int sema_trywait (sema_t *sp);`
  - Decrement the semaphore if count is greater than 0, otherwise, return an error
- `int sema_post (sema_t *sp);`
  - Increment the semaphore, potentially unblocking a waiting thread

Programming with Semaphores

- Simple producer/consumer semaphore example

```c
static int rd_ptr = 0;
static int wr_ptr = 0;
static data_t buf[BUFSIZ];
static sema_t empty, full; // Initialized to 0

// ...
sema_init (&empty, 1, 0, 0);

/* Producer thread 1 */
while (work_to_do) {
    buf[wr_ptr] = produce ();
    sema_wait (&empty);
    wr_ptr = (wr_ptr + 1) % BUFSIZ;
    sema_post (&full);
}

/* Consumer thread 2 */
while (work_to_do) {
    sema_wait (&full);
    consume (buf[rd_ptr]);
    sema_post (&empty);
    rd_ptr = (rd_ptr + 1) % BUFSIZ;
}
```

Readers/writer Locks

- Allow many threads simultaneous read-only access to a protected object
  - However, only a single thread may have write access to the object while excluding any readers or other writers
- Used to protect data that is read more often than written
- Must be fully bracketed (as with mutex)
- Preference is given to writers...
Readers/writer Lock API

- `int rwlock_init (rwlock_t *rwlp, int type, void * arg);`
- `int rwlock_destroy (rwlock_t *rwlp);`
- `int rw_unrlock (rwlock_t *rwlp);`
  - Acquires a write lock, but block if any readers or a writer hold the lock
- `int rw_rdlock (rwlock_t *rwlp);`
  - Acquire a read lock, but block if a writer holds the lock

Readers/writer API (cont'd)

- `int rw_unlock (rwlock_t *rwlp);`
  - Unlock a read/write lock
- `int rw_tryrlock (rwlock_t *rwlp);`
  - Conditionally acquire read lock
- `int rw_trywrlock (rwlock_t *rwlp);`
  - Conditionally acquire write lock

Programming with Readers/writer Locks

- Concurrent bank account program, supports multiple readers, but only 1 writer...

```c
static rwlock_t account_lock; // Initialized to 0
static float checking_balance = 100.0;
static float saving_balance = 100.0;

float get_balance (void) {
    float bal;
    rw_rdlock (&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock (&account_lock);
    return bal;
}

void transfer_checking_to_savings (float amount) {
    rw_wrlock (&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock (&account_lock);
}
```

Comparison of Synchronization Primitives

- Mutex locks are the most basic and most efficient in terms of time and space
  - Based on adaptive spin-locks
- Condition variables provide a different flavor of locking than mutexes and semaphores
  - i.e., blocking themselves rather than blocking other
    - They are much less efficient than mutexes since they use sleep locks
Comparison of Synchronization Primitives (cont’d)

- Semaphores use more memory than mutexes and condition variables
  - Unlike mutexes, they do not require that the original thread is also the thread to release the semaphore
  
    ▷ They also allow more general “counting” behavior, as opposed to binary behavior

- Unlike condition variables they function only on count state, rather than complex condition state

- Readers/writer locks are the most complex synchronization mechanism
  - Use at a fairly coarse-grained level

Multi-threaded Signal Handling

- Signal handling in a single-threaded process is different than in a multi-threaded process

- For example, in a single-threaded process there is never any question as to which “thread” handles a signal

- Likewise, the use of reliable signal mechanisms enable critical sections without explicit locking

- These issues become problematic with in multi-threaded processes...

Two Categories of Signals

1. Traps (e.g., SIGSEGV, SIGPIPE)
   - Result from execution of a specific thread and are handled only by the thread that caused them
   - May be generated and handled simultaneously

2. Interrupts (e.g., SIGINT, SIGIO)
   - Are asynchronous to any thread, resulting from some external action
   - May be handled by any thread whose signal mask is enabled
   - Only one thread is chosen if several are capable of handling the signal
   - If all threads mask the signal it remains pending until some thread enables it

Advanced Topics

- The scope of setjmp and longjmp is limited to one thread
  - In particular, this means that a thread that handles a signal can only perform a longjmp if the corresponding setjmp was performed in the same thread

- The following thread-related functions are async-safe, and may be called in the context of a signal handler

1. sema_post
2. thr_sigsetmask
3. thr_kill
**Signal Masks**

- Each thread has its own signal mask
  - Therefore, a thread may block signals selectively
  - Note that all threads in a process share the same set of signal handlers...
  - Per-thread signal handlers must be programmed explicitly by developers

- Threads can send signals to other threads in their process via `thr_kill`
  - This signal behaves as a trap...
  - Note, there is no direct way to send a signal to a specific thread in a different process

**Programming with Signal Masks**

- The `thr_set_sigmask` function sets the thread's signal mask (which is initially inherited from the parent thread)
  - `int thr_set_sigmask (int how, const sigset_t *set, sigset_t *oset);`

- This example shows how to create a default thread with a new signal mask

```c
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

sigfillset (&new_mask);
sigdelset (&new_mask, SIGINT);
thr_set_sigmask(SIG_SETMASK, &new_mask, &orig_mask);
error = thr_create(0, 0, do_func, 0, 0, &tid);
thr_set_sigmask(SIG_SETMASK, &orig_mask, 0);
```

**Waiting and Signaling Threads**

- The `thr_kill` function sends the specified signal to a specific thread
  - `int thr_kill (thread_t target_thread, int sig);`

- The `sigwait` function waits for a pending signal from the set specified by its argument (regardless of the process signal mask)
  - `int sigwait (sigset_t *set);`
  - `sigwait` returns the number of the pending signal
  - This function is typically used to wait for signals in a separate thread, rather than using a signal handler

**Programming with `sigwait()`**

- Example illustrating the use of `sigwait`

```c
static mutex_t m; // Initialized to default
static int hup = 0;

int main (void) {
  thread_t t;
  int finishup = 0;
  sigset_t set;
  ...
  sigfillset (&set); /* block all signals */
thr_set_sigmask(SIG_BLOCK, &set, 0);
thr_create (0, 0, wait_hup, 0, THR_DETACHED, &t);
do { /* do processing */
  mutex_lock (&m);
  if (hup) {
    finishup = 1;
    mutex_unlock (&m);
  }
  while (finishup == 0);
}
```

```c
void *wait_hup (void *) {
  sigset_t set;
  sigemptyset (&set);
sigaddset (&set, SIGHUP);
sigwait (&set);
mutex_lock (&m);
hup = 1;
mutex_unlock (&m);
}
```
Process Creation and Destruction

- When a process containing multiple threads forks, it creates an exact duplicate
  - i.e., all threads are duplicated
    - However, all interruptible system calls in other threads return EINTR

- A new system call fork1() may be used to duplicate the address space, but only duplicate the invoking thread
  - Typically used to save time, especially if an exec is performed immediately following the fork1

Hazard of Using fork() and vfork()

- There are a number of hazards associated with using fork1 and vfork:
  - If the parent process had threads holding locks then the child process contains locks held by non-existent threads
  - This may lead to deadlock
  - Before calling exec, do not call library functions that use a lock held by more than one thread
  - Do not create new threads between calls to vfork and exec

Thread-Specific Data API

- int thr_keycreate (thread_key_t*, void (*)(void*value));
  - Allocates a global key value
  - The second parameter is a pointer-to-function that is called to cleanup the allocated memory when the thread exits

- int thr_setspecific (thread_key_t, void *value);
  - Binds a value to the key for the calling thread

- int thr_getspecific (thread_key_t, void **value);
  - Retrieves the current value bound to the key for the calling thread
Example of thread-specific data: Trace class

```cpp
class Trace {
    public:
        Trace (void);
        Trace (char *s, int line = 0, char *file = "") {
            "Trace (void);"
        }

    static void start_tracing (void) {
        enable_tracing_ = 1;
    }

    static void stop_tracing (void) {
        enable_tracing_ = 0;
    }

    static void set_nesting_index (int indent) {

    }

    private:
        static thread_key_t depth_key_; //
        static thread_key_t indent_key_; //
        static int once_; //
        static Trace t_; //

        static void cleanup (void) {

        }

        static int nesting_depth_;
        static int nesting_indent_; //

        static void clearsp (void *); //
        static int *nesting_index_; //
        static int *nesting_depth_; //

        static int nesting_depth_ (int *); //
        static int nesting_index_ (int *); //

        static int *nesting_depth_; //
        static int *nesting_index_; //

        char *name_; //
        int (DEFAULT_DEPTH = 0, DEFAULT_INDEX = 3, DEFAULT_TRACING = 0);
    };
}
```

Example of thread-specific data: Trace class

```cpp
void
Trace::set_nesting_index (int indent) {
    nesting_index_ = indent; // Access thread-specific data
}

Trace::Trace (char *s, int line, char *file) {
    if (Trace::enable_tracing()) {
        Log_Msg::log (LOG_INFO, "%s(%s) calling %s, file %s, line %d",
                "this">name_, s, file, line);
    }
}

Trace::Trace (void) {
    if (Trace::enable_tracing()) {
        Log_Msg::log (LOG_INFO, "%s(%s) leaving %s",
                "this">name_);
    }
}

int *
Trace::nesting_depth_ (void) {
    int *ip;
    thr_getspecific (Trace::depth_key_, (void **) &ip);
    if (ip = 0) // First time in
        { ip = new int (Trace::DEFAULT_DEPTH); } // Access TSD
        return ip;
    }

int *
Trace::nesting_index_ (void) {
    int *ip = 0;
    thr_getspecific (Trace::index_key_, (void **) &ip);
    if (ip = 0) // First time in
        { ip = new int (Trace::DEFAULT_INDEX); } // Access TSD
        return ip;
    }
```

Example of thread-specific data: Trace class

```cpp
Trace::Trace (void) {
    if (Trace::once_ == 0) {
        this->name_ = "static dummy";
        Trace::once_ = 1;
        thr_keycreate (&Trace::depth_key_, Trace::clearsp);
        thr_keycreate (&Trace::index_key_, Trace::clearsp);
    }
}

void
Trace::clearsp (void *ptr) {
    Trace::stop_tracing ();
    delete ptr;
}
```

```cpp
```
Example: File Copy

- Perform simultaneous I/O on two different devices

```c
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <sysch.h>

sema_t emptybuf_sem, fullbuf_sem;
struct {
  char data[BUFFSZ]; int size;
} buf[2];

void *producer (void *x) {
  int i = 0;
  for (;;) {
    sema_wait (&emptybuf_sem);
    if (sema_init (emptybuf_sem, 2, 0, 0) != 0 ||
       sema_init (&fullbuf_sem, 0, 0, 0) != 0)
      return 1;
    if (thr_create (0, 0, producer, 0, THR_BEK_LWP, &r_id) == 0
       && thr_create (0, 0, consumer, 0, THR_BEK_LWP, &w_id) == 0)
        i = 1 - i;
    return 1;
  }
}
```

Example: File Copy (cont’d)

- Producer thread

```c
void *producer (void *x) {
  int i = 0;
  for (;;) {
    sema_wait (emptybuf_sem);
    if (buf[i].size <= 0)
      return (void *) 0;
    if (write (1, buf[i].data, buf[i].size) != buf[i].size) {
      fprintf (stderr, "write failed\n");
      return (void *) -1;
    }
    sema_post (emptybuf_sem);
    i = 1 - i;
  }
}
```

Example: File Copy (cont’d)

- Consumer thread

```c
void *consumer (void *x) {
  int i = 0;
  for (;;) {
    sema_wait (&fullbuf_sem);
    if (buf[i].size <= 0)
      return (void *) 0;
    if (write (1, buf[i].data, buf[i].size) != buf[i].size) {
      fprintf (stderr, "write failed\n");
      return (void *) -1;
    }
    sema_post (emptybuf_sem);
    i = 1 - i;
  }
}
```

Example: Matrix Multiplication

- This example illustrates conditional variables and mutexes in the context of multiplication of two-dimensional matrices

```c
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <sysch.h>

#define SZ 10
#define CPUs 4
int number_of_cpu = CPUs;
typedef int (*MATRIX)[SZ];
typedef int MATRIX[SZ][SZ];

static MATRIX m1 = {
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
};
```
static MATRIX m2 =
{
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
};

static MATRIX m3 =
{
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
};

struct
{
    /* Matrix data */
    MATRIX_P m1;
    MATRIX_P m2;
    MATRIX_P m3;
    int row;
    int col;

    /* Multi-processing control variables */
    mutex_t lock;
    cmd_t start_cond;
    cmd_t done_cond;

    /* More control variables */
    int todo;
    int notdone;
    int workers;
}

work;

mutex_t m1_lock;

if (++work.col == SZ)
{
    work.col = 0;
    if (++work.row == SZ)
        work.row = 0;
}

mutex_unlock (&work.lock);

result = 0;

for (i = 0; i < SE; i++)
    result += m1[row][i] * m2[i][col];

m3[row][col] = result;

mutex_lock (&work.lock);

work.notdone--;

if (work.notdone == 0)
    cond_signal (&work.done_cond);

mutex_unlock (&work.lock);

return 0;
}

static void
matrix_multiply (MATRIX mi, MATRIX m2, MATRIX m3)
{
    int i;

    mutex_lock (&m1_lock);
    mutex_lock (&work.lock);

    if (work.workers == 0)
    {

    }

main (argc, char *argv)
{
    int i;

    print (m3);

    for (i = 0; i < 10; i++)
        matrix_multiply (mi, m2, m3);

    print (m3);
}
Conclusions and Caveats

- Some applications do not benefit directly from threads
  - *e.g.*, CPU-bound programs on a uni-processor

- Threads should be created for processing that lasts at least several thousand machine instructions

- Synchronization may be expensive
  - Therefore, choose primitives carefully

- Developer intuition is often underdeveloped...

- Debugging is more complicated
  - *e.g.*, lack of tools