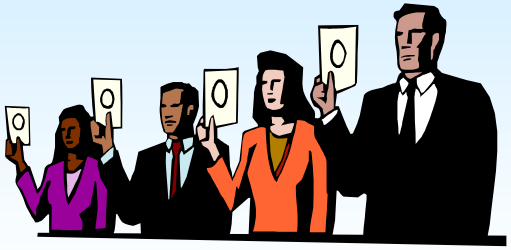
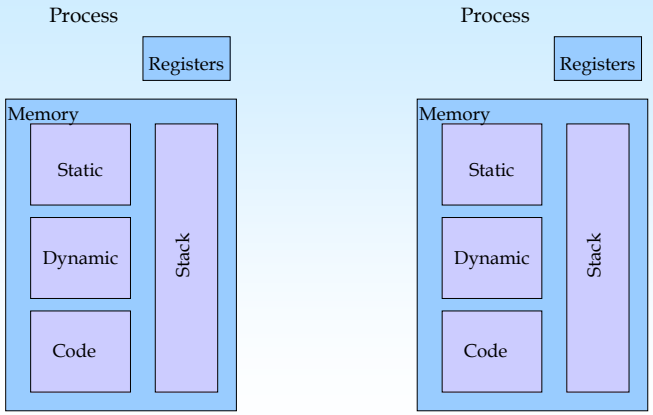
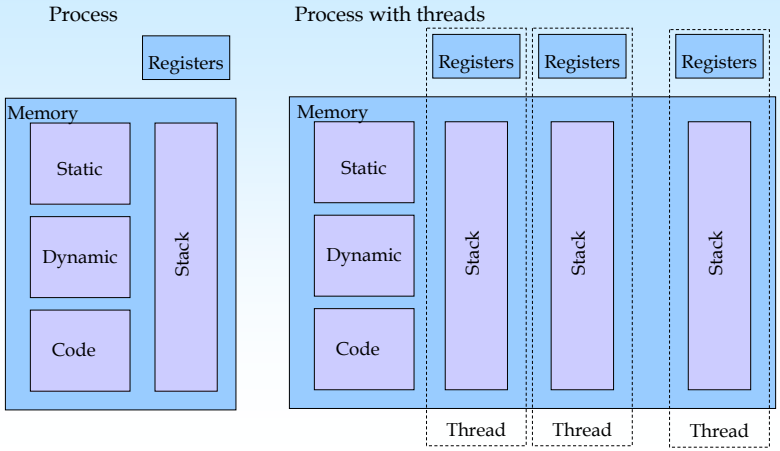
Threads and Threads Programming



Processes



Threads and Processes



Threads

- A thread is a single, sequential flow of control within a program. Within each thread, there is a single point of execution.
- Threads execute concurrently.
- Most traditional programs consist of a single thread.
- Threads execute within (and share) a single address space.
- Synchronization elements ensure proper memory access.

Advantages of using Threads

- Improve performance
  - Multiprocessors: threads may run on separate processors concurrently.
  - Uniprocessors: threads permit overlapping of slow operations (such as I/O) with computation.

Advantages of using Threads

- Use the natural parallelism in applications.
  - E.g., User interaction (slow) can be a thread.
- Multiple threads allow a server to handle clients' requests in parallel instead of artificially serializing them (e.g. Unix *login* process) or creating one server process per client, at great expense (e.g. *httpd*).

Shared Resources

- Address space, shared storage
- The process ID, parent process ID, process group ID
- User ids: real and effective.
- Group ids: real and effective.
- Supplementary group ids
- Current working directory, root directory
- File-mode creation mask
- File descriptor table
- Signal handlers
- Timers

Private Resources

- A unique thread identifier
- Resources required to support a flow of control (such as a stack)
- C/Unix `errno`
- Thread-specific key/value bindings
- Per-thread cancellation handlers
- Per-thread signal masks

Thread Operations

- Creation
- Execution
- Termination
- Cancellation
- Deletion
- Wait for termination
- Synchronization

Mutex

- A mutex is used by multiple threads to ensure the integrity of a shared object that they access by allowing only one thread to access it at a time.
- A mutex has two states, locked and unlocked.
- Each thread locks a mutex before it accesses the shared object and unlocks the mutex when it is finished accessing that object.
- If the mutex is locked by another thread, the thread requesting the lock waits for the mutex to be unlocked.

Mutex

- For each shared object, all threads accessing that data must use the same mutex.
- Each mutex must be initialized before use.

Condition Variables

- A condition variable allows a thread to block its own execution until a condition is met (some shared data reaches a particular state).
- A condition variable is a synchronization object used in conjunction with a mutex.

Condition Variables

- A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state.
- The state is defined by a Boolean expression called a predicate.
- A predicate may be a Boolean variable in the shared data or the predicate may be indirect. E.g., testing whether a counter has reached a certain value; testing whether a queue is empty.

Condition Variables

- Each predicate should have its own unique condition variable.
- Sharing a single condition variable between more than one predicate can introduce inefficiency or errors unless you use extreme care.

POSIX Threads

- Thread library standardized by POSIX
- Portable across different processor architectures
- Applies to both multiprocessor and single processor systems
- Set of library routines to support thread operations and a set of defined types to support objects relating to threads.

Thread creation

- Thread is created using `pthread_create`.
- This routine
  - creates the thread object based on the specified or default attributes, and
  - starts execution of a specified function.

```
void * hello(void *) {  cout << "hello "; return 0; }
…
pthread_t th;
pthread_attr_t *tattr = 0;
pthread_create(&th, tattr, hello, 0);
```

Threads Example #1

```
void * hello(void *) { cout << "hello "; return 0; }
void * world(void *arg) {
  pthread_t *th = (pthread_t *)(arg);
  pthread_join(*th, 0); cout << "world";
  return 0;
}
void main() {
  pthread_t th, tw;
  pthread_attr_t *tattr = 0;
  pthread_create(&th, tattr, hello, 0);
  pthread_create(&tw, tattr, world, &th);
  pthread_join(tw, 0); cout << endl;
  pthread_exit(0);
}
```

Main thread creates two threads 'hello' and `world'. The `hello' thread prints "hello " and exits. The `world' thread waits for the `hello' thread to finish, prints "world", and then exits. The main thread, after creating these two threads, waits for the `world' thread to finish, prints end-of-line, and terminates.

Threads Example #2

```
pthread_mutex_t m;
pthread_cond_t cv;
bool helloDone = false;

void * hello(void *) {
  cout << "hello ";
  pthread_mutex_lock(&m);
  helloDone = true;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&m);
  return 0;
}
```

Similar to the previous example, but here a condition varaible and mutex is uesd to serialize the `hello' and `world' threads.

The communication between `hello' & `world' threads is via a shared variable helloDone.

Threads Example #2

```
void * world(void *) {
  pthread_mutex_lock(&m);
  while ( !helloDone ) pthread_cond_wait(&cv, &m);
  pthread_mutex_unlock(&m);
  cout << "world";
  return 0;
}
void main() {
  pthread_t th, tw;  pthread_attr_t *tattr = 0;
  pthread_mutex_init(&m, 0);
  pthread_cond_init(&cv, 0);
  pthread_create(&tw, tattr, world, 0);
  pthread_create(&th, tattr, hello, 0);
  pthread_join(tw, 0); cout << endl; pthread_exit(0);
}
```

The `hello' thread sets helloDone when it finishes and sends a signal to the `world' thread. The `world' thread waits for this signal and when it receives it asserts helloDone is set and prints "world". The main thread, as before, waits for the `world' thread to finish, prints end-of-line, and terminates.

Waiting for Thread Termination

- A thread waits for the termination of another thread by calling the `pthread_join` routine.
- Execution in the current thread is suspended until the specified thread terminates.
- Behavior is undefined if multiple threads call `pthread_join` and specify the same thread. This is because completion of the first join will detach the target thread.
- Specifying the current thread with the `pthread_join` routine, results in a deadlock.

Mutexes

- Define mutexes as global variables since they are generally required to be visible to all the threads that contend.
- Initialize the mutex by calling `pthread_mutex_init`. You can also use the static initializer PTHREAD_MUTEX_INITIALIZER.

Mutexes

- Initialize a mutex only once.
- After deciding that a mutex is no longer used or needed, use `pthread_mutex_destroy` to release the resources associated with it.
- Note that the second argument to `pthread_mutex_init` specifies the attributes of the mutex; a 0 value indicates the default mutex attribute.

Condition Variables

- Use `pthread_cond_init` routine to initialize a condition variable. You can also use the static initializer `PTHREAD_COND_INITIALIZER`.
- Use the `pthread_cond_wait` routine to cause a thread to wait until the condition is signaled or broadcasted.
- Use `pthread_cond_signal` to wake one thread that is waiting on the condition variable.
- Use `pthread_cond_broadcast` to wake all threads that are waiting on a condition variable.

Condition Variables

- Define condition variables as global variables since they are usually required to be visible to all the threads that use them.
- Associate a boolean with a condition variable.
- Initialize the condition variable by calling `pthread_cond_init` before use.
- Initialize a condition variable only once.
- Destroy a condition variable and reclaim its storage by calling the `pthread_cond_destroy`.

Condition Variables

```
pthread_mutex_t m;
pthread_cond_t cv;
bool condition_met = false;

// .....

pthread_mutex_lock(&m);
while (! condition_met ) {
        pthread_cond_wait(&cv, &m);
}
pthread_mutex_unlock(&m);
```

Java Threads

- Every object inherited from the class `Thread` is a thread.
- Every object that implements the interface `Runnable` is runnable under a thread.
- The method `run` contains the code that should be run by the thread.
- There are methods to manipulate the threads.

Monitors

- Monitors provide mutual exclusion. Every object that has a `synchronized` method is a monitor.
- Only one thread can be executing any of the `synchronized` methods of a monitor.

Monitors

- When a thread executing a `synchronized` method cannot proceed due to some condition, it may voluntarily `wait` for the condition to be satisfied.
- Another thread that executes code that will change this condition may `notify` the original thread that the condition has been changed.

Monitors

- `wait` makes the thread wait until notified by another thread of a change in this object. The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the `notify` method or the `notifyAll` method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

Monitors

- `notify` wakes up a single thread that is waiting on this object's monitor.
- `notifyAll` wakes up all threads that are waiting on this object's monitor.

C# Threads

- Similar to Java in many respects
  - Thread creation similar to POSIX threads
- A *ThreadStart* delegate points to the code that should be run by the thread

```
public void DoSomething() { ... }

ThreadStart ts = new ThreadStart(DoSomething);
Thread t = new Thread(ts);
t.Start();

Thread t2 = new Thread(new ThreadStart(DoSomething));
t2.Start();
```

C# Threads

- *Monitor.Enter* for entry to critical section, and *Monitor.Exit* to exit out of the critical section.
- *Monitor.Wait* releases the lock on an object and blocks the current thread until it reacquires the lock. Same as the Java wait.
- *Monitor.Pulse* notifies a thread in the waiting queue of a change in the locked object's state.
- *Monitor.PulseAll* notifies all waiting threads of a change in the object's state.

Example: Readers/Writers

In general, the number of reads to shared data exceeds the number of writes. Reading shared data concurrently can be permitted, but writing concurrently cannot be. Reading and writing concurrently cannot be permitted either.

Using a simple mutex to protect shared data restricts concurrent read accesses as well. Design and implement (using pseudo-code) a ReadWriteLock object that allows concurrent read accesses, but disallows concurrent write or read/write accesses. You may use simple mutexes and/or condition variables in your code.

Hint. This object may have public methods readLock, readUnlock, writeLock, and writeUnlock to permit locking and locking under read and write accesses.

Example: Readers/Writers

```
class ReadWriteLock {
private:
  int noOfReaders;
  mutex m;
  condition_variable cv;
public:
  ReadWriteLock();
  void readLock();
  void readUnlock();
  void writeLock();
  void writeUnlock();
};
```

Constructor: initializes *noOfReaders* to 0, and initializes the mutex *m*, and condition variable *cv*.

Example: Readers/Writers

```
void ReadWriteLock::readLock()
{
  m.lock();
  ++noOfReaders;
  m.unlock();
}

void ReadWriteLock::readUnlock()
{
  m.lock();
  --noOfReaders;
  if ( noOfReaders == 0 ) cv.signal(); // wake up a writer
  m.unlock();
}
```

Example: Readers/Writers

```
void ReadWriteLock::writeLock()
{
  m.lock();
  while ( noOfReaders > 0 ) cv.wait(m);
}

void ReadWriteLock::writeUnlock()
{
  cv.signal(); // wake up a writer
  m.unlock();
}
```

Example: Monitors

- A monitor provides high-level synchronization for multiple threads.
  - It has mechanisms for *Lock*, *Unlock*, *Wait*, *SignalOne*, and *SignalAll*.
  - In OO terms, a monitor is an object that implicitly has a mutex and a condition variable (*cv*). The mutex & *cv* are automatically added by the compiler.

Example: Monitors

- Typically, *Lock* and *Unlock* are implicit. A monitor may provide a way to specify critical sections. For example "*lock* { ... }" or "*synchronized* { ... }". These translate to "{ mutex.lock(); ...; mutex.unlock(); }" where the mutex operations are implicitly added by the compiler.
- The *Wait* and *Signal* operations can only be used from within critical sections. *Wait* releases the *mutex* on entry and re-acquires it prior to exit.

Example: Monitors

```
abstract class Monitor {
private:
  mutex m;
  condition_variable cv;
public:
  void Lock()       { m.lock(); }
  void Unlock()     { m.unlock(); }
  void Wait()       { cv.wait(m); }
  void SignalOne()  { cv.signal(); }
  void SignalAll()  { cv.broadcast(); }
}
```

Example: Monitors

- A monitor class may derive from *Monitor* and use the *Monitor* operations.
- A monitor class could simply have a *Monitor* instance in the class and use the *Monitor* operations through this instance.
  - Having more than one *Monitor* instance may lead to incorrect usage, so derivation is a better approach than using an instance.

Thread Safety

- A function is thread-safe if simultaneous execution of the function by multiple threads produces logically correct results.
- Such a function is also known as a re-entrant function.
- Functions that rely on global states are generally not thread-safe.
- Thread safety is the avoidance of data races - situations in which data are set to either correct or incorrect values depending on the order in which multiple threads access and modify the data.

Unsafe functions

An unsafe function has data races. It does not produce logically correct results.

```
fputs(const char *s, FILE *stream)
{
  char *p;
  for ( p = s; *p; ++p )
    putc((int)*p, stream);
}
```

Thread-Safe Functions

- A thread-safe function would produce logically correct results even if it is executed simultaneously by several threads.
- An unsafe function can be made thread-safe by enclosing it within a lock/unlock.
- A thread-safe function simply serializes simultaneous accesses.

Thread-Safe functions: An Example

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
fputs(const char *s, FILE *stream)
{
  char *p;
  pthread_mutex_lock(&m);
  for ( p = s; *p; ++p )
    putc((int)*p, stream);
  pthread_mutex_unlock(&m);
}
```

MT-Safe Functions

- An MT-safe function is not only thread-safe but also uses the threads that access the function simultaneously to exploit the parallelism within the function.
- That is, several threads may be accessing the function simultaneously, operating on distinct data.
- In contrast, a thread-safe function simply serializes simultaneous accesses.

MT-Safe Functions: An Example

- When two threads are calling it to print to different files, one need not wait for the other; both can safely print at the same time.
- An MT-safe version of `fputs` may use one lock for each file, allowing two threads to print to different files at the same time.

MT-Safe functions : An Example

```
pthread_mutex_t m[FOPEN_MAX];
fputs(const char *s, FILE *stream)
{
  char *p;
  pthread_mutex_lock(&m[fileno(stream)]);
  for ( p = s; *p; ++p )
    putc((int)*p, stream);
  pthread_mutex_unlock(&m[fileno(stream)]);
}
```

Exercises

- Implement a Java monitor that solves the Readers/Writers problem.
- Develop a solution for a *one lane bridge* problem using mutexes and condition variables.