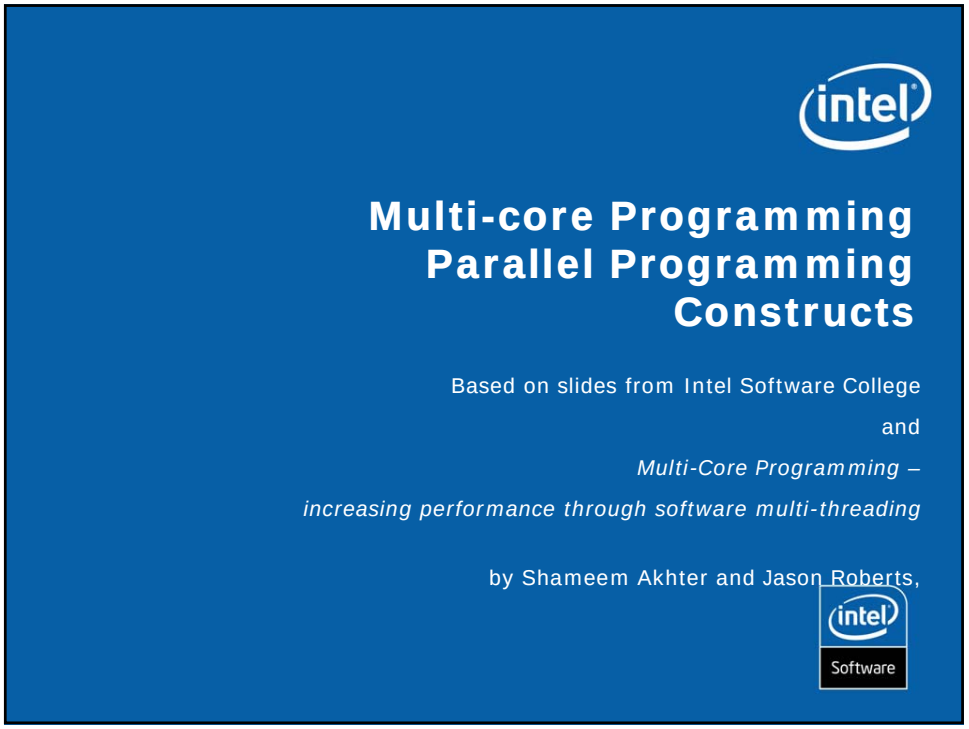
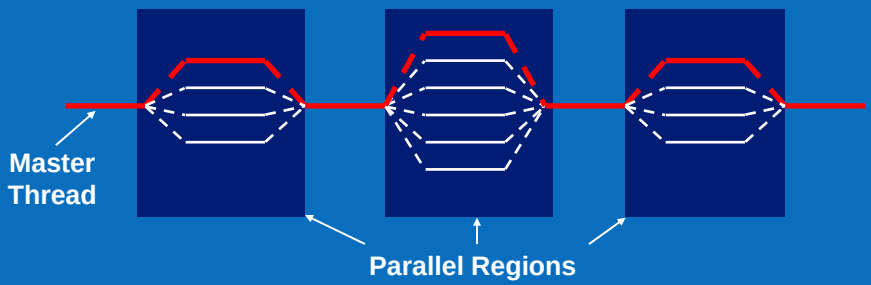


Implicit threading Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



Synchronization

- Used to coordinate thread execution and manage shared data
- Two types of primitives
  - Mutual exclusion (mutex)
    - One thread blocks a *critical section* – a section of code with shared data
    - One or more threads wait to enter
    - Mutex controlled by scheduler
  - Condition synchronization
    - Blocks thread until certain conditions on system state are met

Race Conditions

Threads "race" against each other for resources
- Execution order is assumed but cannot be guaranteed

Storage conflict is most common
- Concurrent access of same memory location by multiple threads
  - At least one thread is writing

Difficult to diagnose
- Non-deterministic
- Debugging probes can mask race conditions
- May only manifest as slight numerical deviation

Example: *Musical Chairs*

Mutual Exclusion

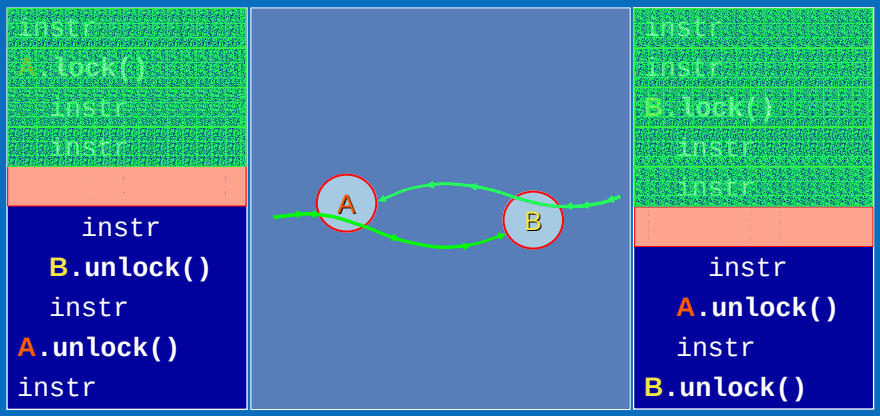
Critical Region
- Portion of code that accesses (reads & writes) shared variables

Mutual Exclusion
- Program logic to enforce single thread access to critical region
- Enables correct programming structures for avoiding race conditions

Example: *Safe Deposit box*
- Attendants ensure mutual exclusion

- Minimize the size of critical sections when practical

Deadlock



Deadlock

Thread waits on event/object/message that will never occur

Causes:

- Incorrect locking hierarchy – lock ordering deadlock
  - T1 locks A, needs B
  - T2 locks B, needs A
- Self-deadlock : thread tries to acquire a lock it already has
- Recursive deadlock: wakeup path of one thread resides in another
- Other cases:
  - Thread terminated holding sync. object
  - Waiting at different barriers

Deadlock

Threads wait for some event or condition that will never happen

Example:

- Traffic jam at intersection
- Cars unable to turn or back up

What is Livelock?

- Threads change state in response to each other

Example:

- *Robin Hood and Little John on log bridge*

Synchronization

Synchronization primitives used to enforce mutual exclusion

- Lock, semaphore, condition variable
  - Implemented by *atomic operations* and use *memory fences (instructions that enforce an ordering constraint on memory operations issued before and after the fence instruction)*
- One thread "holds" sync. object; other threads must wait
- When done, holding thread releases object; some waiting thread given object

Example: *Library book*

- One patron has book checked out
- Others must wait for book to return

Data in Threaded Applications

Sharing Data	Protecting Data
Store values in shared memory locations • Need to synchronize access • use immutables	Within Method • Locals vars, objects • Grouping in Public Services • Alternative protocols Within Thread • Thread Local Storage Within Object • Aggregation • Delegation Adapters • Synchronized Adapters Within Domain

Synchronization Primitives

Wait-Busy	(critical section)
Semaphores	(**mutex, lock**)
Monitors	(**condition variables**, wait-signal, wait-notify)

C#.NET	Java
• Interlocked • Lock • Monitor • Event • Mutex	Specified by language • synchronized (semaphore) • Monitor. wait()/notify() • Both presented by Object Since JTSE 5.0 • java.util.concurrent • Lock, Semaphore • Condition

Semaphores

• Introduced by Dijkstra in 1968

• semaphore is integer *sem* that can only be accessed ot changed by two atomic operations *P (proberen)* and *V (verhogen)* – now often called *wait* and *signal*

• For mutex sem is initialized to 1

P(s) -> atomic {sem=sem-1; temp =sem}

if (temp<0){ block thread and place on list waiting for s}

V(s) -> atomic {sem=sem+1; temp =sem}

if (temp<=0){ release thread from list waiting for s}

Semaphores

- If *sem* is positive represents number of threads that can proceed without blocking
- If *sem* is negative represents number of blocked threads
- If *sem* is zero no thread waiting but any trying to enter will block
- If *sem* is limited to 0 or 1 called a *binary semaphore*
- Semaphores can be *strong* or *weak*
  - Strong enforce FCFS (first come first served) and avoid starvation
  - Weak do not – POSIX semaphores are weak
- Semaphores are consider a low level synchronization primitive
- In POSIX operations are sem_post() and sem_wait()

Producer-consumer using semaphores Initialize semaphore to size of buffer

```
semaphore sEmpty, sFull

void producer() {     while(1) {
        sEmpty->P()
        <produce data>
        sFull->V()  }}

void consumer() {     while(1) {
        sFull->V()
        <consume data>
        sEmpty->P()  }}
```

Locks

- Similar to semaphores
- Two basic operations
  - acquire() : atomically waits for lock state to be unlocked, then sets the lock state to lock
  - Release() : atomically changes lock state from locked to unlocked
- At most one thread acquires a lock
- Note that applications must loop (busy wait) until the lock is acquired
  - No wait built into lock implementation
- When thread wants to access shared data
  - Acquires lock
  - Exclusively performs operations on shared data
  - Releases lock
- Can be implicit locks e.g. from database to protect data
  - Safer to use explicit locks in applications

Lock Types - Mutex

Mutexes

- Simplest lock
- In POSIX
  - pthread_mutex_lock()
    - first thread that locks gets access
    - other threads trying to lock fail causing the thread to go to sleep
  - pthread_mutex_unlock()
    - Unlocks, one sleeper is awakened and given chance to obtain lock
    - Another thread may acquire first
- A timer attribute can be added to release the lock automatically after a certain time
- Can also have a try-finally clause to release mutex if exception occurs – helps to avoid deadlock

Locks – Recursive Locks

Recursive Locks

- Can be repeatedly acquired by a thread holding the load – avoids self-deadlock
- Acquires must be balanced with releases – another thread can only acquire when it has been released once for each acquire
- Most useful in recursive functions
- Slower than non-recursive locks
- Not provided in POSIX but easy to build

Lock Types

Recursive Lock Use in Recursive Function

```
Recursive_Lock L
void recursiveFunction (int count) {
        L->acquire()
        if (count > 0) {
                count=count-1;
                 recursiveFunctin (count);
        }
        L->release();
}
```

Lock Types - Read-Write Locks

Read-Write Locks

- Shared-exclusive or multiple-read/single-write or non-mutual exclusion semaphores
- Allow simultaneous read access to multiple threads but limit write access to only one thread
- Sometimes better to break lengthy data into smaller blocks each guarded by own read-write locks
- Not provided in POSIX but easy to build

Lock Types - Spin Locks

Spin Locks

- Non-blocking locks owned by a thread
- Waiting threads “spin” that is poll the state of the lock rather than get blocked
- Used mostly on multi-processor/core systems
  - On single-core system no resources to run thread that will release lock
- Use spin-locks when average hold time of lock less than time to block and wakeup a thread
  - Say hold time is 50% to 100% of thread context switch time
  - Don’t hold during calls to subsystems
- Spin locks used incorrectly can lead to starvation
  - Can be alleviated by using queueing - every waiting thread spins on different flag using FIFO queue
- Not provided in POSIX but easy to build

Condition Variables

- Unlike semaphores, no stored value is associated with it
- A condition variable creates a safe environment for testing a condition, sleep on it when false, and be awakened when it might be true
- A shared data state is used instead to maintain condition
  - Should always be accessed, tested or changed inside a mutex
- If condition is true, thread completes task and releases mutex
- If condition is false, the mutex is released *for you by system,* and thread goes to sleep on condition variable
- When other thread changes some aspect of condition, calls *cond_signal* waking up one sleeping thread
- Preferable to locks when specific scheduling behavior needed between threads

Condition Variables

To operate on shared data, condition variable C uses a lock L

Three basic atomic operations

- cond_wait(L): atomically releases lock and blocks thread on cond, when returns lockis reacquired
- cond_signal(L) : enables one of waiting threads to run, reacquire mutex and retest condition
- cond_broadcast(L) : enables all waiting threads to run, and reacquire lock, and retest condition
- To control pool of threads, use of signal function is recommended
- Using broadcast-based signaling could be expensive
- Sometimes effective e.g. in readers-writers problems

Condition Variables for Producer Consumer

Variable LC used to associate C with L

Condition C;

Lock L;

Bool LC=false;

Condition Variables for Producer Consumer Producer

```
void producer() {
        while (1) {
                L->acquire();
                //start critical section
                while (LC==true) { C->cond_wait(L);}
                // produce the next data
                LC = true;
                C->cond_signal(L);
                // end critical section
                L->release();
        }
}
```

Condition Variables for Producer Consumer Consumer

```
void consumer() {
        while (1) {
                L->acquire();
                //start critical section
                while (LC==false) { C->cond_wait(L);}
                // produce the next data
                LC = false;
                C->cond_signal(L);
                // end critical section
                L->release();
        }
}
```


Monitors

- A higher level construct to simplify the use of condition variables and locks
- If language supports monitors, compiler automatically inserts lock operations at the beginning and end of each synchronization-aware routine
- Java supports (and Algol did) monitors and synchronized blocks inside a method
  - Used to perform resource management in JMX (Java Management Extension)
- Most modern languages do not implement monitors

Flow control based concepts

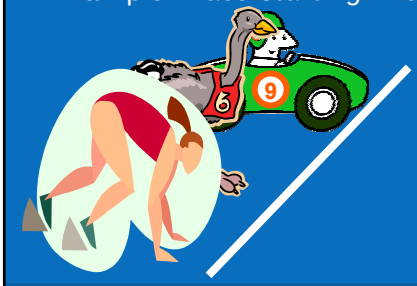
- Fence
  - On shared memory multiprocessor or multi-core system, fence instructions ensures consistent memory operations
  - Fence guarantees the completion of all pre-fence memory operations and halts all post-fence memory operations until after fence instruction
  - Ensures proper memory mapping from software to hardware memory
  - Using fence instructions explicitly could be error-prone
  - Better to reply on compiler technology to implement implicitly

Barrier Synchronization

All threads pause at barrier
- Threads waiting there are idle; overhead

When all threads arrive, all are released

Example: *Race starting line*



Implementation-Dependent Threading Features

- Concept of threads independent of operating systems
- Implementation and semantics different e.g. in Win32, Win64 and POSIX
- Windows API defined by Microsoft
- Pthread API defined by IEEE
  - Implementation up to OS developers
  - Not all features in all implementations
  - Pthreads can be used as wrapper for native threads
  - Native Linux Pthreads library called Native POSIX Thread Library (NPTL)

POSIX-Windows differences

Examples:

Mechanism to Signal Threads

- Windows uses an event model
- POSIX uses condition variables

Multiple mechanism within OS Win32

Two versions of mutex with different API

1. Mutex – kernel mechanism
   - Need user-mode to kernel-mode transition
   - Expensive but can be used across process boundaries
2. CriticalSection – user level mechanism