Synchronization in Practice

- User program synchronization
  - for threads
  - for processes
- OS kernel synchronization

User Program Synchronization for Processes

- Processes naturally do not share the same address space
- Process communication and synchronization:
  - semaphore
  - shared memory and memory-based synchronization (spinlocks)
  - pipes

User Program Synchronization for Threads

- All threads share the same address space
- When only need to protect a short critical section (busy waiting is OK)
  - software/hardware spin locks
  - still has the risk of context switch in the middle of critical section
- For complex synchronization (busy waiting is not OK)
  - semaphore, mutex lock, condition variable, …
  - may need kernel help
- In pthreads
  - mutex lock and condition variable
  - condition variable must be used together with a mutex lock

Synchronization Primitives in Pthreads

- Mutex lock
  - pthread_mutex_init
  - pthread_mutex_destroy
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Condition variable (used in conjunction with a mutex lock)
  - pthread_cond_init
  - pthread_cond_destroy
  - pthread_cond_wait
  - pthread_cond_signal
  - pthread_cond_broadcast

Mutex Locks: Creation and Destruction

pthread_mutex_init(
pthread_mutex_t * mutex,
const pthread_mutex_attr *attr);

- Creates a new mutex lock

pthread_mutex_destroy(
pthread_mutex_t *mutex);

- Destroys the mutex specified by mutex

Mutex Locks: Lock

pthread_mutex_lock(
pthread_mutex_t *mutex)

- Tries to acquire the lock specified by mutex.
- If mutex is already locked, then calling thread blocks until mutex is unlocked.

Mutex Locks: UnLock

pthread_mutex_unlock(
pthread_mutex_t *mutex);

- If calling thread has mutex currently locked, this will unlock the mutex.
- If other threads are blocked waiting on this mutex, one will unblock and acquire mutex
- Which one is determined by the scheduler

Condition variables: Creation and Destruction

pthread_cond_init(
pthread_cond_t *cond,
pthread_cond_attr *attr)

- Creates a new condition variable cond

pthread_cond_destroy(
pthread_cond_t *cond)

- Destroys the condition variable cond.

Condition Variables: Wait

```
pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex)
```

- Blocks the calling thread, waiting on cond
- Unlocks the mutex
- Re-acquires the mutex when unblocked

Condition Variables: Signal

```
pthread_cond_signal(
    pthread_cond_t *cond)
```

- Unblocks one thread waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

Condition Variables: Broadcast

```
pthread_cond_broadcast(
    pthread_cond_t *cond)
```

- Unblocks all threads waiting on cond.
- If no thread waiting, then broadcast is a no-op.

Use of Condition Variables

- IMPORTANT NOTE: A signal is “forgotten” if there is no corresponding wait that has already occurred
- Use semaphores (or construct a semaphore) if you want the signal to be remembered

TSP (Traveling Salesman)

- Goal:
  - given a list of cities, a matrix of distances between them, and a starting city,
  - find the shortest tour in which all cities are visited exactly once.
- Example of an NP-hard search problem.
- Algorithm: branch-and-bound.

Branching

- Initialization:
  - go from starting city to each of remaining cities
  - put resulting partial path into priority queue, ordered by its current length.
- Further (repeatedly):
  - take head element out of priority queue,
  - expand by each one of remaining cities,
  - put resulting partial path into priority queue.

Finding the Solution

- Eventually, a complete path will be found.
- Remember its length as the current shortest path.
- Every time a complete path is found, check if we need to update current best path.
- When priority queue becomes empty, best path is found.

Using a Simple Bound

- Once a complete path is found, we have a lower bound on the length of shortest path
- No use in exploring partial path that is already longer than the current lower bound
- Better bounding methods exist …

Sequential TSP: Data Structures

- Priority queue of partial paths.
- Current best solution and its length.
- For simplicity, we will ignore bounding.

Sequential TSP: Code Outline

```
init_q(); init_best();
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) };
        else { en_queue(q) };
  }
}
```


Parallel TSP: Possibilities

- Have each process do one expansion
- Have each process do expansion of one partial path
- Have each process do expansion of multiple partial paths
- Issue of granularity/performance, not an issue of correctness.
- Assume: process expands one partial path.

Parallel TSP: First Cut (part 1)

process i:

```
  while( (p=de_queue()) != NULL ) {
    for each expansion by one city {
          q = add_city(p);
          if complete(q) { update_best(q) };
          else en_queue(q);
    }
  }
```

Parallel TSP: First cut (part 2)

- In de_queue: wait if q is empty
- In en_queue: signal that q is no longer empty

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```


Parallel TSP: Critical Sections

- All concurrently accessed shared data must be protected by critical section
- Update_best must be protected by a critical section
- En_queue and de_queue must be protected by the same critical section

Termination condition

- How do we know when we are done?
- All processes are waiting inside de_queue.
- Count the number of waiting processes before waiting.
- If equal to total number of processes, we are done.

Parallel TSP: Mutual Exclusion

```
en_queue() / de_queue() {
   pthread_mutex_lock(&queue);
   …;
   pthread_mutex_unlock(&queue);
}
update_best() {
   pthread_mutex_lock(&best);
   …;
   pthread_mutex_unlock(&best);
}
```


Parallel TSP: Condition Synchronization

```
de_queue() {
   pthread_mutex_lock(&queue);
   while( (q is empty) and (not done) ) {
         waiting++;
         if( waiting == p ) {
               done = true;
               pthread_cond_broadcast(&empty);
         }
         else {
               pthread_cond_wait(&empty, &queue);
               waiting--;
         }
   }
   if( done )
         return null;
   else
         remove and return head of the queue;
   pthread_mutex_unlock(&queue);
}
```


SYNCHRONIZATION IN THE LINUX KERNEL

Examples of OS Kernel Synchronization

- Two processes making system calls to read/write on the same file, leading to possible race condition on the file system data structures in OS
- Interrupt handlers put I/O data into a buffer queue that might be retrieved by application-initiated I/O system calls

OS Kernel Structure for Synchronization

- OS is divided into two parts:
  - upper part (serving application requests): system call, exception
  - lower part (serving hardware device requests): interrupt handling
- Upper part runs in process/thread context
  - resource accounting to corresponding process/thread
  - running on a kernel stack usually associated with the corresponding process/thread control block
- Lower part runs in a separate interrupt context
  - resource accounting to who?
  - running in a separate (often dedicated) kernel interrupt stack
- Blocking behaviors:
  - Upper part may block (yield CPU), interleave with others
  - Lower part does not block, must run atomically (one by one) – interrupt handlers typically run with other interrupts disabled
- Preemption/priority:
  - A lower part interrupt handler may preempt an upper part system call processing, but not vice versa

OS Kernel Synchronization

- Available mechanisms:
  - disabling interrupts
  - spin_lock (busy waiting lock)
  - blocking synchronization (mutex lock, semaphore, …)
- Synchronization between upper part kernel "threads"
  - typically blocking synchronization
  - Spin lock if critical section short (only useful on multiprocessor)
- Synchronization between an upper part kernel "thread" and a lower part interrupt handler:
  - if blocking synchronization: block only at upper part, never lower part (possible in semaphore)
  - Spin lock may be used (only useful on multiprocessor)
  - the upper part should disable interrupt before entering critical section

A Little More on OS Kernel Structure

- Lower part interrupt handlers do not block
  - interrupt handlers typically run with other interrupts disabled
- This can be a problem when interrupt handlers do more and more work
- In modern OSes, interrupt handlers typically defer some work to later (interruptible contexts)
  - soft irqs in Linux

Preemptible Kernel

- Preemptible kernel
  - One in which a process switch may occur at any point when a process is executing in kernel mode
  - Requires the re-entrant property
    - Several processes may be executing in kernel mode at the same time
    - Use either re-entrant functions (ones that don't modify glabal variables) or thread-safe functions

Synchronization in Linux

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local / All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Per-CPU Variables

- An array of data structures, one element per CPU
- Ensure that element falls on a unique cache line
- Caveats?
  - Still require disabling preemption on a single CPU
  - Prone to race conditions because of the above

Atomic Operations

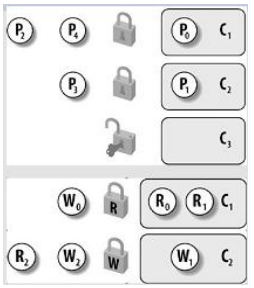
- Read-modify-write
  - Use specific atomic instructions or lock prefix on x86
- Ensures that operations are not interleaved with those by other threads
  - Locally ensures that process will not get context switched between read and write in read-modify-write (single opcode)

Memory Barriers

- Prevent compiler reordering (e.g., reordering for optimized register use)
  - Optimization barrier
- Prevent hardware reordering
  - Memory barrier
  - Instructions that operate on I/O, or are “locked” (on x86 machines)
  - Writes to control registers
  - … a few others

Spin Locks (+ R-W spinlocks)

- Ensure that code executing spin lock is non-blocking
- spinlock_t (uses xchgb on x86)
- R-W spin locks
  - Uses an atomic decrement and subtract with multiple values for the lock



Sequence locks

- Higher priority to writers
- Seqlock_t consists of two fields – spinlock and an integer sequence
- Reads – read sequence before and after

```
do {
    seq = read_seqbegin(&seqlock);
    … critical section …
} while (read_seqretry(&seqlock, seq);
```

- Writes – acquire spinlock, increase sequence by 1 on entry; increase sequence by 1 and release spinlock on exit
  - write_seqlock()
  - write_sequnlock()

Read-Copy Update

- Non-blocking form of synchronization
- Reader reads in place
  - Data structure must be dynamically allocated and referenced using pointers
  - Disable preemption
- Copy data structure to write; switch pointers (requires memory barriers to ensure that data structure modification precedes pointer modification)
- Old copy can be freed only after all potential readers have unlocked – special function to free old copy

Semaphores

- struct semaphore
  - atomic_t count
  - wait (wait queue list address)
  - Sleepers (count to indicate processes are sleeping)
- To be used only by functions allowed to block
- Read/write semaphores
- Mutexes (binary semaphores)

Synchronization in Linux

Goal: Maximize concurrency

- Per-CPU variables to avoid synchronization
- Atomic variables (non-blocking)
- Read-copy-update (non-blocking)
- Sequence locks (writer-prioritized reader/writer locks)
- Spin-locks – basic, r/w (blocking)
- Semaphores (sleeping)
- Local interrupt disabling

Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, and Kai Shen. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).