Systèmes d'Exploitation Avancés

Instructor: Pablo Oliveira

ISTY

Review : Thread package API

- `tid thread_create (void (*fn) (void *), void *arg);`
  - Create a new thread that calls `fn` with `arg`
- `void thread_exit ();`
- `void thread_join (tid thread);`
- The execution of multiple threads is interleaved
- Can have *non-preemptive threads* :
  - One thread executes exclusively until it makes a blocking call.
- Or *preemptive threads* :
  - May switch to another thread between any two instructions.
- Using multiple CPUs is inherently preemptive
  - Even if you don't take CPU_0 away from thread T, another thread on CPU_1 can execute between any two instructions of T.

Program A

```
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main () {
  tid id = thread_create (p1, NULL);
  p2 (); thread_join (id);
}
```

- Can both critical sections run ?

Program B

```
int data = 0, ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

- Can use be called with value 0 ?

Correct answers

- Program A : I don't know
- Program B : I don't know
- Why ?
  - It depends on your hardware
  - If it provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency

Sequential Consistency

- *Sequential consistency* : The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. [Lamport]
- Boils down to two requirements :
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*
- Without SC, multiple CPUs can be “worse” than preemptive threads
  - May see results that cannot occur with any interleaving on 1 CPU
- Why doesn't all hardware support sequential consistency ?

SC thwarts hardware optimizations

- Can't re-order overlapping write operations
  - Coalescing writes to same cache line
- Complicates non-blocking reads
  - E.g., speculatively prefetch `data`
- Makes cache coherence more expensive

SC thwarts compiler optimizations

- Code motion
- Caching value in register
  - Collapse multiple loads/stores of same address into one operation
- Common subexpression elimination
  - Could cause memory location to be read fewer times
- Loop blocking
  - Re-arrange loops for better cache performance
- Software pipelining
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency [intel 3a, §8.2]

- x86 supports multiple consistency/caching models
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- Choices include :
  - **WB** : Write-back caching (the default)
  - **WT** : Write-through caching (all writes go to memory)
  - **UC** : Uncacheable (for device memory)
  - **WC** : Write-combining – weak consistency & no caching
    (used for frame buffers, when sending a lot of data to GPU)

x86 WB consistency

- Old x86s (e.g, 486, Pentium 1) had almost SC
  - Exception : A read could finish before an earlier write to a different location
  - Which of Programs A, B, might be affected ?
- Newer x86s also let a CPU read its own writes early

x86 WB consistency

- Old x86s (e.g, 486, Pentium 1) had almost SC
  - Exception : A read could finish before an earlier write to a different location
  - Which of Programs A, B, might be affected ? *Just A*
- Newer x86s also let a CPU read its own writes early

x86 atomicity

- `lock` prefix makes a memory instruction atomic
  - Usually locks bus for duration of instruction (expensive !)
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered w. locked ones
- `xchg` instruction is always locked (even w/o prefix)
- Special fence instructions can prevent re-ordering
  - `mfence` – can't be reordered w. reads or writes

Assuming sequential consistency

- Important point : Know your memory model
  - Particularly as OSes typically have their own synchronization
- Most application code should avoid depending on memory model
  - Obey certain rules, and behavior should be identical to S.C.
- Let's for now say we have sequential consistency
- Example concurrent code : Producer/Consumer
  - `buffer` stores `BUFFER_SIZE` items
  - `count` is number of used slots
  - `out` is next empty buffer slot to fill (if any)
  - `in` is oldest filled slot to consume (if any)

```c
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        while (count == BUFFER_SIZE)
            /* do nothing */;
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            /* do nothing */;
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        consume_item (nextConsumed);
    }
}
```

- What can go wrong here ?

Data races

- `count` may have wrong value
- Possible implementation of `count++` and `count--`

register←count	register←count
register←register $+ 1$	register←register $- 1$
count←register	count←register

- Possible execution (count one less than correct) :

register←count
register←register $+ 1$
register←count
register←register $- 1$
count←register
count←register

Data races (continued)

- What about a single-instruction add ?
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe ?

Data races (continued)

- What about a single-instruction add ?
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe ?
- Not atomic on multiprocessor !
  - Will experience exact same race condition
  - Can potentially make atomic with `lock` prefix
  - But `lock` very expensive
  - Compiler won't generate it, assumes you don't want penalty
- Need solution to *critical section* problem
  - Place `count++` and `count--` in critical section
  - Protect critical sections from concurrent execution

Desired properties of solution

- *Mutual Exclusion*
  - Only one thread can be in critical section at a time
- *Progress*
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- *Bounded waiting*
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in
- Note progress vs. bounded waiting
  - If no thread can enter C.S., don't have progress
  - If thread A waiting to enter C.S. while B repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

Mutexes

- Must adapt to machine memory model if not S.C.
  - Ideally want your code to run everywhere
- Want to insulate programmer from implementing synchronization primitives
- Thread packages typically provide *mutexes* :

```
void mutex_init (mutex_t *m, ...);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acquires `m` at a time, others wait

Thread API contract

- All global data should be protected by a mutex !
  - Global = accessed by more than one thread, at least one write
  - Exception is initialization, before exposed to other threads
  - This is the responsibility of the application writer
- If you use mutexes properly, behavior should be indistinguishable from Sequential Consistency
  - This is the responsibility of the threads package (& compiler)
  - Mutex is broken if you use properly and don't see S.C.
- OS kernels also need synchronization
  - May or may not look like mutexes

Same concept, many names

- Most popular application-level thread API : *pthreads*
  - Function names in this lecture all based on *pthreads*
  - Just add `pthread_` prefix
  - E.g., `pthread_mutex_t`, `pthread_mutex_lock`, ...

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex); /* <--- Why? */
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
          mutex_unlock (&mutex);
          thread_yield ();
          mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Condition variables

- Busy-waiting in application is a bad idea
  - Thread consumes CPU even when can't make progress
  - Unnecessarily slows other threads and processes
- Better to inform scheduler of which threads can run
- Typically done with *condition variables*
- `void cond_init (cond_t *, ...);`
  - Initialize
- `void cond_wait (cond_t *c, mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `void cond_signal (cond_t *c);`
  `void cond_broadcast (cond_t *c);`
  - Wake one/all threads waiting on `c`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
          cond_wait (&nonempty, &mutex);

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Re-check conditions

- Always re-check condition on wake-up :

```
while (count == 0) /* not if */
  cond_wait (&nonempty, &mutex);
```

- Otherwise, breaks with two consumers
  - Start with empty buffer, then :

C_1	C_2	P
`cond_wait (...);`		`mutex_lock (...);`
		⋮
		`count++;`
		`cond_signal (...);`
	`mutex_lock (...);`	`mutex_unlock (...);`
	`if (count == 0)`	
	⋮	
	use `buffer[out] ...`	
	`count--;`	
	`mutex_unlock (...);`	
use `buffer[out] ...` ⟵ No items in buffer		

Condition variables (continued)

- Why must `cond_wait` both release mutex & sleep ?
- Why not separate mutexes and condition variables ?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

Condition variables (continued)

- Why must cond_wait both release mutex & sleep ?
- Why not separate mutexes and condition variables ?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

- Can end up stuck waiting when bad interleaving

PRODUCER	CONSUMER
`while (count == BUFFER_SIZE)`	
`  mutex_unlock (&mutex);`	
	`mutex_lock (&mutex);`
	`...`
	`count--;`
	`cond_signal (&nonfull);`
`  cond_wait (&nonfull);`	

Semaphores [Dijkstra]

- A *Semaphore* is initialized with an integer N
- Provides two functions :
  - `sem_wait (S)` (originally called P)
  - `sem_signal (S)` (originally called V)
- Guarantees `sem_wait` will return only N more times than `sem_signal` called
  - Example : If $N == 1$, then semaphore is a mutex with `sem_wait` as lock and `sem_signal` as unlock
- Semaphores give elegant solutions to some problems
- Linux primarily uses semaphores for sleeping locks
  - `sema_init`, `down_interruptible`, `up`, ...
  - But evidence might favor mutexes [Molnar]

Semaphore producer/consumer

- Initialize `nonempty` to 0 (block consumer when buffer empty)
- Initialize `nonfull` to N (block producer when queue full)

```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait (&nonfull);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&nonempty);
    }
}
void consumer (void *ignored) {
    for (;;) {
        sem_wait (&nonempty);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&nonfull);
        consume_item (nextConsumed);
    }
}
```

Other thread package features

- Alerts – cause exception in a thread
- Timedwait – timeout on condition variable
- Shared locks – concurrent read accesses to data
- Thread priorities – control scheduling policy
  - Mutex attributes allow various forms of *priority donation* (will be familiar concept after lab 1)
- Thread Local Storage

Implementing synchronization

- User-visible mutex is straight-forward data structure

```
typedef struct mutex {
  bool is_locked;          /* true if locked */
  thread_id_t owner;       /* thread holding lock, if locked */
  thread_list_t waiters;   /* threads waiting for lock */

  lower_level_lock_t lk;   /* Protect above fields */
};
```

- Need lower-level lock `lk` for mutual exclusion
  - Internally, `mutex_*` functions bracket code with `lock(mutex->lk)` ... `unlock(mutex->lk)`
  - Otherwise, data races ! (E.g., two threads manipulating `waiters`)
- How to implement `lower_level_lock_t` ?
  - Use hardware support for synchronization

Approach #1 : Disable interrupts

- Only for apps with $n:1$ threads (1 kthread)
  - Cannot take advantage of multiprocessors
  - But sometimes most efficient solution for uniprocessors
- Have per-thread "do not interrupt" (DNI) bit
- `lock (lk)` : sets thread's DNI bit
- If timer interrupt arrives
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- `unlock (lk)` : clears DNI bit *and* checks I bit
  - If I bit is set, immediately yields the CPU

Approach #2 : Spinlocks

- Most CPUs support atomic read-[modify-]write
- Example : `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value
  - Special instruction – can't be implemented in portable C
- Use this instruction to implement *spinlocks* :

```
#define lock(lockp)  while (test_and_set (lockp))
#define trylock(lockp) (test_and_set (lockp) == 0)
#define unlock(lockp) *lockp = 0
```

- Spinlocks implement mutex's `lower_level_lock_t`
- Can you use spinlocks instead of mutexes ?
  - Wastes CPU, especially if thread holding lock not running
  - Mutex functions have short C.S., less likely to be preempted
  - On multiprocessor, sometimes good to spin for a bit, then yield