Synchronization

Main Points

- Thread implementation
- Race conditions
- Locks and mutual exclusion

Implementing threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)
- stub(func, args): Pintos switch_entry
  - Call (*func)(args)
  - Call thread_exit()

Thread Stack

- What if a thread puts too many procedures on its stack?
  - What should happen?
  - What happens in Java?
  - What happens in Linux?
  - What happens in Pintos?

Implementing (voluntary) thread context switch

- User-level threads in a single-threaded process
  - Save registers on old stack
  - Switch to new stack, new thread
  - Restore registers from new stack
  - Return
- Kernel threads
  - Exactly the same!
  - Pintos: thread switch always between kernel threads, not between user process and kernel thread

Pintos: switch_threads (oldT, nextT) (interrupts disabled!)

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
# This stack frame must match the
     one set up by thread_create()
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
     thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
```

```
# Change stack pointer to new
     thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (Pintos)
  - End of interrupt handler calls switch_threads()
  - When resumed, return from handler resumes kernel thread or user process
- Faster version (textbook)
  - Interrupt handler returns to saved state in TCB
  - Could be kernel thread or user process

Two threads call yield

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	return thread_yield	return thread_yield
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
...	...	...

Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O

Synchronization Motivation

Thread 1

```
p = someFn();
isInitialized = true;
```

Thread 2

```
while (! isInitialized ) ;
q = aFn(p);

if q != aFn(someFn())
   panic
```

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions

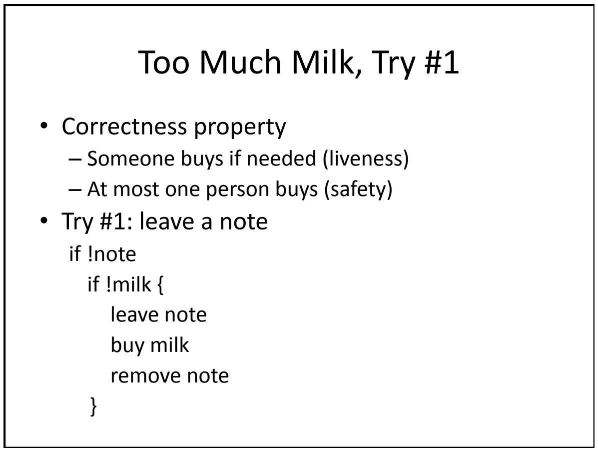
Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

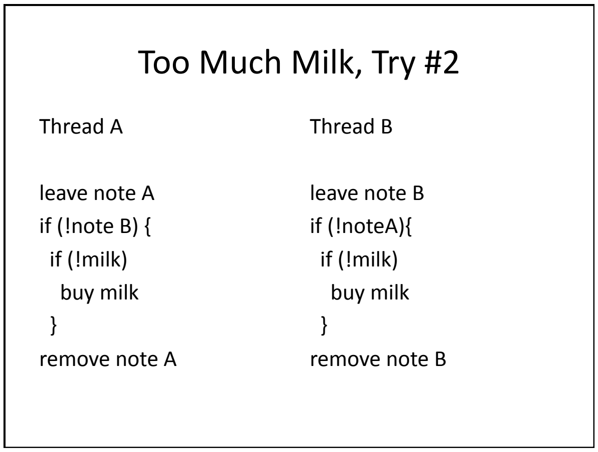
- Lock before entering critical section, before accessing shared data
- unlock when leaving, after done accessing shared data
- wait if locked (all synch involves waiting!)



Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

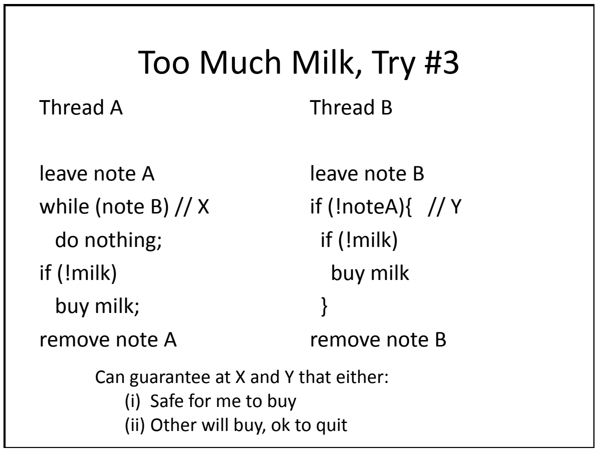
```
if !note
   if !milk {
      leave note
      buy milk
      remove note
   }
```



Too Much Milk, Try #2

```
Thread A                Thread B

leave note A            leave note B
if (!note B) {          if (!noteA){
  if (!milk)              if (!milk)
    buy milk                buy milk
  }                       }
remove note A           remove note B
```

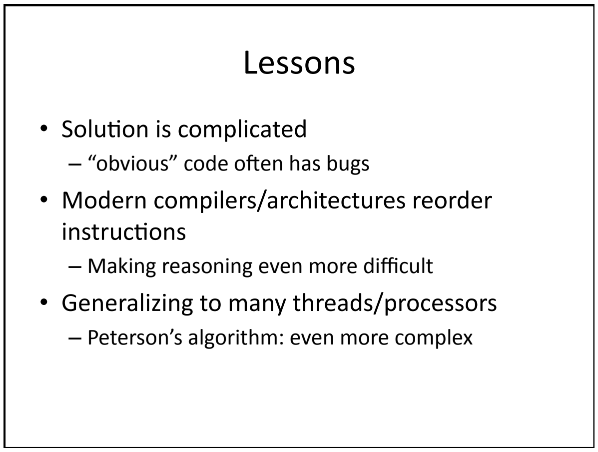


Too Much Milk, Try #3

```
Thread A                Thread B

leave note A            leave note B
while (note B) // X     if (!noteA){   // Y
   do nothing;            if (!milk)
if (!milk)                  buy milk
   buy milk;              }
remove note A           remove note B
```

Can guarantee at X and Y that either:
(i) Safe for me to buy
(ii) Other will buy, ok to quit



Lessons

- Solution is complicated
  - "obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Peterson's algorithm: even more complex

Locks

- lock_acquire
  - wait until lock is free, then take it
- lock_release
  - release lock, waking up anyone waiting for it

Allows concurrent code to be much simpler:

```
lock_acquire()
if (!milk) buy milk
lock_release()
```

- Implementation of locks
  - Hardware support for read/modify/write instructions

Lock Example: Malloc/Free

```
char *malloc (n) {
 lock_acquire(lock);
 p = allocate memory
 lock_release(lock);
 return p;
}
```

```
void free(char *p) {
  lock_acquire(lock);
  put p back on free list
  lock_release(lock);
}
```

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
  - E.g., Java "synchronized"
- What if we need to wait?
  - Ex: if no free memory, malloc could wait for free
  - Condition variables