Changelog

Changes made in this version not seen in first lecture:

- 27 September: sum example (to global): fix error where i was used in place of sum

Threads 2 / Synchronization 0

last time

Linux's completely fair scheduler
- proportional share scheduler
- lowest "virtual time"
- increment virtual time based on weight, usage

real time scheduling
- physically determined deadlines
- earliest deadline first
- provide gaurentees, maybe

multi-threaded processes
- shared memory, files, etc. between threads
- own registers
- stack allocated for each thread

pthread API — create, join, detach

quiz question (1)

The XV6 scheduler scans the proc table (after first getting a lock) looking for a ready to run process. This is both slow and non scalable to thousands of processes. An alternative that is faster is to:

if N processes, K runnable, average of N/K reads to find next process

quiz question (1)

The XV6 scheduler scans the proc table (after first getting a lock) looking for a ready to run process. This is both slow and non scalable to thousands of processes. An alternative that is faster is to:

if N processes, K runnable, average of N/K reads to find next process

Randomly select a task to run and run it.

not all tasks are runnable

Maintain a hash table or ready to run processes keyed on their priority.

need to scan high, unused priorities

arbitrarily more work depending on # of priorities

quiz question (2)

The XV6 scheduler scans the proc table (after first getting a lock) looking for a ready to run process. This is both slow and non scalable to thousands of processes. An alternative that is faster is to:

if N processes, K runnable, average of N/K reads to find next process

Maintain a stack of proc table pointers and pop process to run them and push them when they become ready to run.

1 read to find process, better than N/K

but *causes very bad starvation*

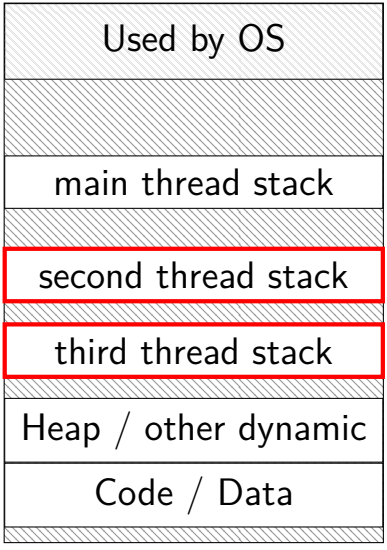
Maintain a linked list of ready to run processes and always run the first process on the list, and place newly ready processes at the end.

1 read to find process, better than N/K

what's wrong with this?

```
/* omitted: headers, using statements */
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, get_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

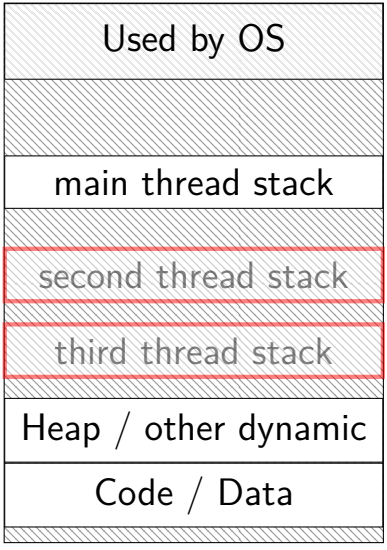
`string result` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

`string result` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

pthreads: by default need to `join` thread to deallocate everything

thread kept around to allow collecting return value

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL, show_progress, NULL)
    pthread_detach(show_progress_thread);
}
int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs, show_progress, NULL
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, NULL, show_progress,
}
```

threads: sum example (to global)

```
int array[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += array[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

threads: sum example (to global)

array, results: global variables — shared

```
int array[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += array[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =   (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =   (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

array: global variable — shared

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =   (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->
        sum += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

my_info: pointer to sum_all's stack
only okay because sum_all waits!

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =   (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (no globals)

```
struct ThreadInfo { int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->array[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *array) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (no globals)

```
struct ThreadInfo { int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->array[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *array) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (no globals)

```
struct ThreadInfo { int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->array[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *array) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (no globals)

```
struct ThreadInfo { int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->array[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *array) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *array) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

threads: sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *array) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

threads: sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *array; int start; int end; int result };
void *sum_thread(void *argument) {
    ...
}
ThreadInfo *start_sum_all(int *array) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].array = array; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}
void finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

the correctness problem

schedulers introduce non-determinism

- scheduler might run threads in any order
- scheduler can switch threads at any time

worse with threads on multiple cores

- cores not precisely synchronized (stalling for caches, etc., etc.)
- different cores happen in different order each time

makes reliable testing very difficult

solution: correctness by design

example application: ATM server

commands: withdraw, deposit

one correctness goal: don't lose money

ATM server

(pseudocode)

```
ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

a threaded server?

```
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    StoreAccount(account);
}
```

maybe Get/StoreAccount can be slow?

- read/write disk sometimes? contact another server sometimes?

maybe lots of requests to process?

- maybe real logic has more checks than Deposit()
- ...

all reasons to handle multiple requests at once

→ many threads all running the server loop

multiple threads

```
main() {
    for (int i = 0; i < NumberOfThreads; ++i) {
        pthread_create(&server_loop_threads[i], NULL,
                       ServerLoop, NULL);
    }
    ...
}

ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
```

a side note

why am I spending time justifying this?

multiple threads for something like this make things much trickier

we'll be learning why...

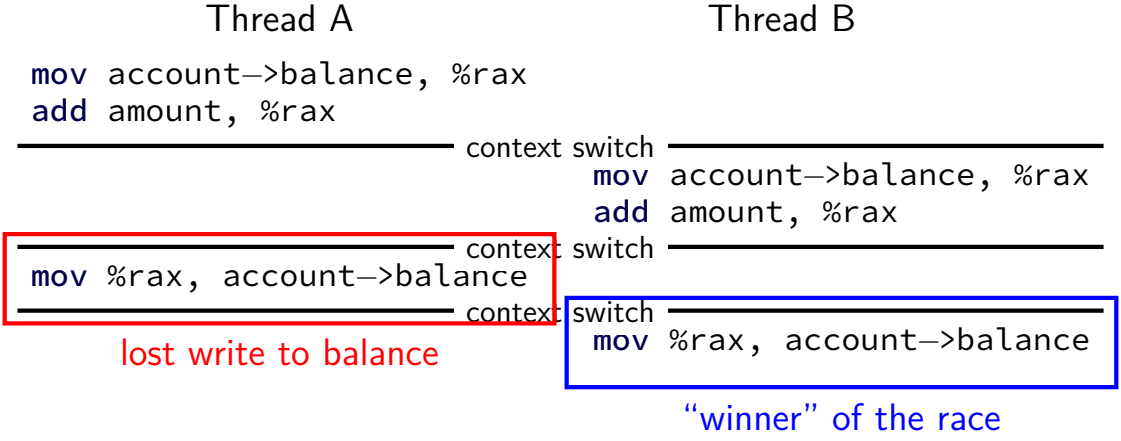
the lost write

`account->balance += amount;` (in two threads, same account)

Thread A		Thread B
`mov account->balance, %rax`		
`add amount, %rax`		
	context switch	
		`mov account->balance, %rax`
		`add amount, %rax`
	context switch	
`mov %rax, account->balance`		
	context switch	
		`mov %rax, account->balance`

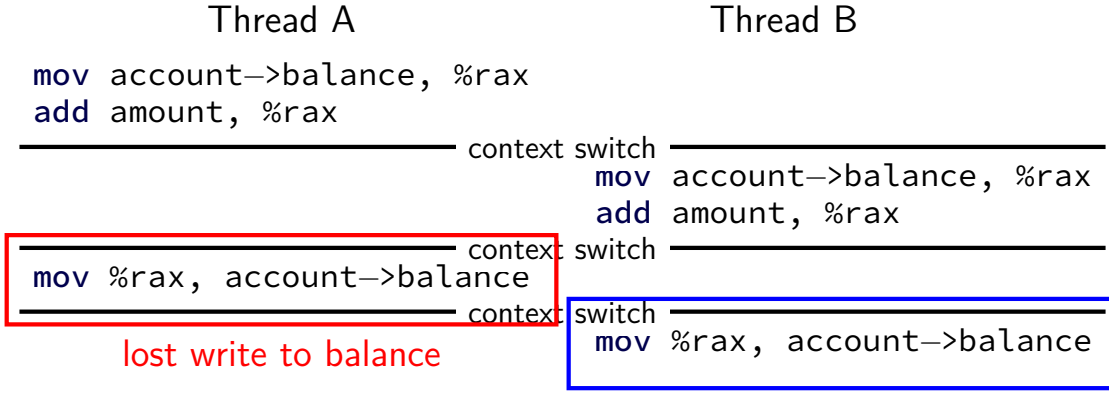
the lost write

`account->balance += amount;` (in two threads, same account)



the lost write

`account->balance += amount;` (in two threads, same account)



thinking about race conditions (1)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow 1$	$y \leftarrow 2$

thinking about race conditions (1)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow 1$	$y \leftarrow 2$

must be 1. Thread B can't do anything

thinking about race conditions (2)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

thinking about race conditions (2)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

1 or 3 or 5 (non-deterministic)

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow 1$	$x \leftarrow 2$

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	**Thread B**
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

...but why not 3? maybe each bit of x assigned seperately?

atomic operation

atomic operation = operation that runs to completion or not at all

we will use these to let threads work together

most machines: loading/storing words is atomic

- so can't get 3 from $x \leftarrow 1$ and $x \leftarrow 2$ running in parallel

but some instructions are not atomic

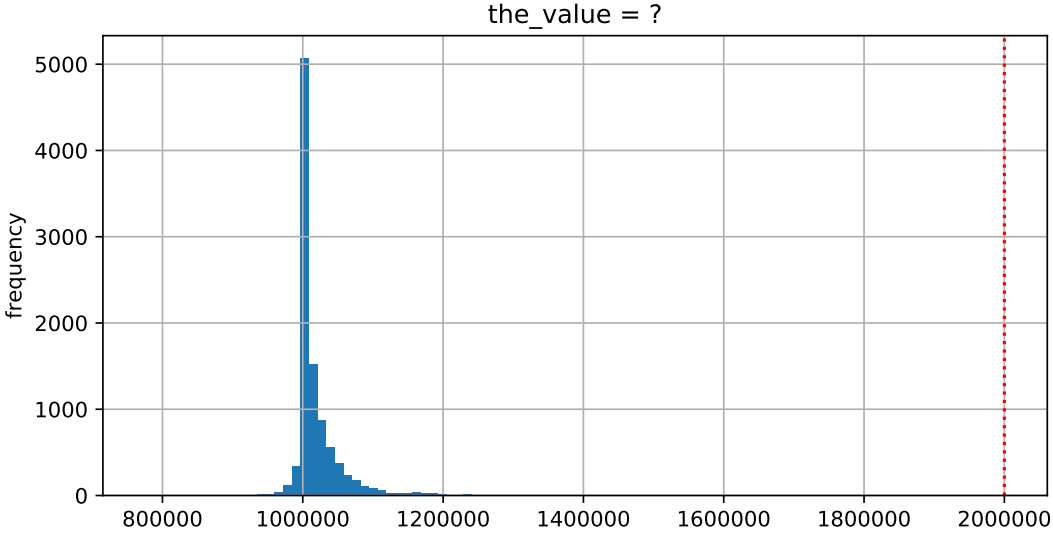
- one example: normal x86 `add` constant to memory

lost adds (program)

```
.global update_loop
update_loop:
    addl $1, the_value // the_value (global variable) += 1
    dec %rdi           // argument 1 -= 1
    jg update_loop     // if argument 1 >= 0 repeat
    ret
```

```
int the_value;
extern void *update_loop(void *);
int main(void) {
    the_value = 0;
    pthread_t A, B;
    pthread_create(&A, NULL, update_loop, (void*) 1000000);
    pthread_create(&B, NULL, update_loop, (void*) 1000000);
    pthread_join(A, NULL);
    pthread_join(B, NULL);
    // expected result: 1000000 + 1000000 = 2000000
    printf("the_value = %d\n", the_value);
}
```

lost adds (results)



but how?

probably not possible on single core

- exceptions can't occur in the middle of `add` instruction

...but 'add to memory' implemented with multiple steps

- still needs to load, add, store internally
- can be interleaved with what other cores do

but how?

probably not possible on single core

- exceptions can't occur in the middle of `add` instruction

...but 'add to memory' implemented with multiple steps

- still needs to load, add, store internally
- can be interleaved with what other cores do

(and actually it's more complicated than that — we'll talk later)

so, what is actually atomic

for now we'll assume: load/stores of 'words'
(64-bit machine = 64-bits words)

in general: processor designer will tell you

their job to design caches, etc. to work as documented

too much milk

roommates Alice and Bob want to keep fridge stocked with milk:

time	Alice	Bob
3:00	look in fridge. no milk	
3:05	leave for store	
3:10	arrive at store	look in fridge. no milk
3:15	buy milk	leave for store
3:20	return home, put milk in fridge	arrive at store
3:25		buy milk
3:30		return home, put milk in fridge

how can Alice and Bob coordinate better?

too much milk "solution" 1 (algorithm)

leave a note: "I am buying milk"

- place before buying
- remove after buying
- don't try buying if there's a note

≈ setting/checking a variable (e.g. "`note = 1`")

- with atomic load/store of variable

```
if (no milk) {
    if (no note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

too much milk “solution” 1 (timeline)

Alice

```
if (no milk) {
    if (no note) {

        leave note;
        buy milk;
        remove note;
    }
}
```

Bob

```
if (no milk) {
    if (no note) {

        leave note;
        buy milk;
        remove note;
    }
}
```

too much milk "solution" 2 (algorithm)

intuition: leave note when buying or checking if need to buy

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

⟵ but there's always a note

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

← but there's always a note

…will never buy milk (twice *or* once)

“solution” 3: algorithm

intuition: label notes so Alice knows which is hers (and vice-versa)

computer equivalent: separate noteFromAlice and noteFromBob variables

Alice

```
leave note from Alice;
if (no milk) {
    if (no note from Bob) {
        buy milk
    }
}
remove note from Alice;
```

Bob

```
leave note from Bob;
if (no milk) {
    if (no note from Alice)
        buy milk
    }
}
remove note from Bob;
```

too much milk: “solution” 3 (timeline)

Alice

```
leave note from Alice
if (no milk) {
    if (no note from Bob) {
        buy milk
    }
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no milk) {
    if (no note from Alice) {
        buy milk
    }
}
remove note from Bob
```

too much milk: is it possible

is there a solutions with writing/reading notes?

$\approx$ loading/storing from shared memory

yes, but it's not very elegant

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

exercise (hard): extend to three people

Peterson's algorithm

general version of solution

see, e.g., Wikipedia

we'll use special hardware support instead

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

critical section: code that exactly one thread can execute at a time

- result of critical section

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

> like checking for and, if needed, buying milk

critical section: code that exactly one thread can execute at a time

> result of critical section

lock: object only one thread can hold at a time

> interface for creating critical sections

the lock primitive

locks: an object with (at least) two operations:

acquire or *lock* — wait until lock is free, then "grab" it

release or *unlock* — let others use lock, wakeup waiters

```
Lock(MilkLock);
if (no milk) {
    buy milk
}
Unlock(MilkLock);
```

pthread mutex

```
#include <pthread.h>

pthread_mutex_t MilkLock;
pthread_mutex_init(&MilkLock, NULL);
...
pthread_mutex_lock(&MilkLock);
if (no milk) {
    buy milk
}
pthread_mutex_unlock(&MilkLock);
```

xv6 spinlocks

```
#include "spinlock.h"
...
struct spinlock MilkLock;
initlock(&MilkLock, "name_for_debugging");
...
acquire(&MilkLock);
if (no milk) {
    buy milk
}
release(&MilkLock);
```

C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

can access from multiple threads ...as long as not being resized?

C++ standard rules for containers

multiple threads can read anything at the same time

can only read element if no other thread is modifying it

can only add/remove elements if no other threads are accessing container

- some exceptions, read documentation really carefully

implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

x86 instructions:

- `cli` — disable interrupts
- `sti` — enable interrupts

naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: user can hang the system:

```
Lock(some_lock);
while (true) {}
```

naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: user can hang the system:

```
Lock(some_lock);
while (true) {}
```

problem: can't do I/O within lock

```
Lock(some_lock);
read from disk
    /* waits forever for (disabled) interrupt
       from disk IO finishing */
```

naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

naive interrupt enable/disable (2)

```
Lock() {
    disable interrupts
}
```

```
Unlock() {
    enable interrupts
}
```

problem: nested locks

```
Lock(milk_lock);
if (no milk) {
    Lock(store_lock);
    buy milk
    Unlock(store_lock);
    /* interrupts enabled here?? */
}
Unlock(milk_lock);
```

xv6 interrupt disabling (1)

```
...
acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock
  ... /* this part basically just for multicore */
}
release(struct spinlock *lk)
{
  ... /* this part basically just for multicore */
  popcli();
}
```

xv6 push/popcli

pushcli / popcli — need to be in pairs

pushcli — disable interrupts if not already

popcli — enable interrupts if corresponding pushcli disabled them

- don't enable them if they were already disabled

a simple race

```
thread_A:
    movl $1, x    /* x ← 1 */
    movl y, %eax /* return y */
    ret
```

```
thread_B:
    movl $1, y    /* y ← 1 */
    movl x, %eax /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

a simple race

```
thread_A:
    movl $1, x    /* x ← 1 */
    movl y, %eax  /* return y */
    ret
```

```
thread_B:
    movl $1, y    /* y ← 1 */
    movl x, %eax  /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

if loads/stores atomic, then possible results:

A:1 B:1 — moves into x/y execute, then moves into eax

A:0 B:1 — thread A executes before thread B

A:1 B:0 — thread B executes before thread A

a simple race: results

```
thread_A:
    movl $1, x    /* x ← 1 */
    movl y, %eax  /* return y */
    ret
```

```
thread_B:
    movl $1, y    /* y ← 1 */
    movl x, %eax  /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	(‘A executes before B’)
171 161	A:1 B:0	(‘B executes before A’)
4 706	A:1 B:1	(‘execute moves into x/y first’)
394	A:0 B:0	???

a simple race: results

```
thread_A:
    movl $1, x    /* x ← 1 */
    movl y, %eax  /* return y */
    ret
```

```
thread_B:
    movl $1, y    /* y ← 1 */
    movl x, %eax  /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	(‘A executes before B’)
171 161	A:1 B:0	(‘B executes before A’)
4 706	A:1 B:1	(‘execute moves into x/y first’)
394	A:0 B:0	???

load/store reordering

recall: out-of-order processors

processors execute instructons in different order

- hide delays from slow caches, variable computation rates, etc.

convenient optimization: execute loads/stores in different order

why load/store reordering?

prior example: load of x executing before store of y

why do this? otherwise delay the load

- if x and y unrelated — no benefit to waiting

some x86 reordering restrictions

each core sees its own loads/stores in order

(if a core store something, it can always load it back)

stores *from other cores* appear in a consistent order

(but a core might observe its own stores "too early")

causality: if a core reads X, then writes Y, no core can observe the read of Y before the read X

Source: Intel 64 and IA-32 Software Developer's Manual, Volume 3A, Chapter 8

how do you do anything with this?

special instructions with stronger ordering rules

special instructions that restirct ordering of instructions around them ("fences")

- loads/stores can't cross the fence

pthreads and reordering

synchronizing pthreads functions prevent reordering

- everything before function call actually happens before everything after

includes preventing some optimizations

- e.g. keeping global variable in register for too long

not just pthread_mutex_lock/unlock!

includes pthread_create, pthread_join, ...

GCC: preventing reordering

intended to help implementing things like pthread_mutex_lock

builtin functions starting with `__sync` and `__atomic`

prevent CPU reordering *and* prevent compiler reordering

also provide other tools for implementing locks (more later)

could also hand-write assembly code

- compiler can't know what assembly code is doing

GCC: preventing reordering example (1)

```
void Alice() {
    note_from_alice = 1;
    do {
        __atomic_thread_fence(__ATOMIC_SEQ_CST);
    } while (note_from_bob);
    if (no_milk) {++milk;}
}
```

```
Alice:
  movl $1, note_from_alice  // note_from_alice ← 1
.L3:
  mfence  // make sure store is visible to other cores before
          // not needed on second+ iteration of loop
  cmpl $0, note_from_bob  // if (note_from_bob == 0) repeat fe
  jne .L3
  cmpl $0, no_milk
  ...
```

mfence

x86 instruction `mfence`

make sure all loads/stores in progress finish

...and make sure no loads/stores were started early

fairly expensive

- Intel 'Skylake': order 33 cycles + time waiting for pending stores/loads

GCC: preventing reordering example (2)

```
void Alice() {
    int one = 1;
    __atomic_store(&note_from_alice, &one, __ATOMIC_SEQ_CST);
    do {
    } while (__atomic_load_n(&note_from_bob, __ATOMIC_SEQ_CST));
    if (no_milk) {++milk;}
}
```

```
Alice:
  movl $1, note_from_alice
  mfence
.L2:
  movl note_from_bob, %eax
  testl %eax, %eax
  jne .L2
  ...
```

backup slides

passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

passing thread IDs (1)

```
DataType items[1000];
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = 500 * thread_id;
    int end = start + 500;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(2);
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
}
```

passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads - 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
```

passing thread IDs (2)

```
DataType items[1000];
int num_threads;
void *thread_function(void *argument) {
    int thread_id = (int) argument;
    int start = thread_id * (1000 / num_threads);
    int end = start + (1000 / num_threads);
    if (thread_id == num_threads - 1) end = 1000;
    for (int i = start; i < end; ++i) {
        DoSomethingWith(items[i]);
    }
    ...
}
void run_threads() {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void*) i);
    }
    ...
```

passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```

passing data structures

```
class ThreadInfo {
public:
    ...
};

void *thread_function(void *argument) {
    ThreadInfo *info = (ThreadInfo *) argument;
    ...
    delete info;
}

void run_threads(int N) {
    vector<pthread_t> threads(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL,
            thread_function, (void *) new ThreadInfo(...));
    }
    ...
}
```