Concurrent Programming’s Goals

to make our apps utilize multicore, we use multithreading.

concurrent programming is a way to manage explicit parallelism

1. Performance

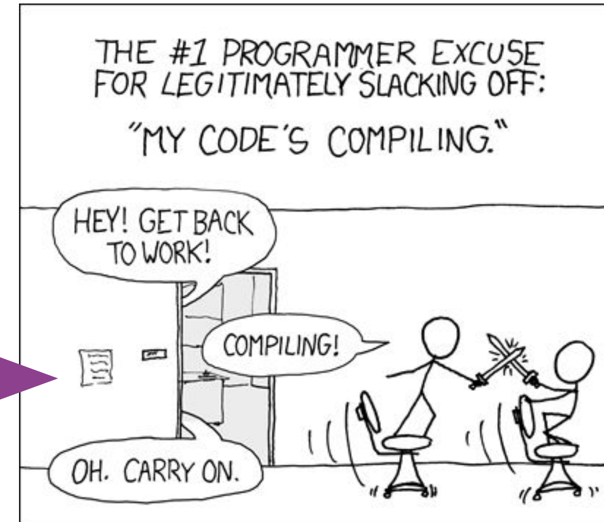
   Effective use of hardware

2. Productivity

   Effective use of Software Dev’s time

3. Generality

   To lower the cost of low-level concurrency and parallelism



https://xkcd.com/303/

cleaner in Go. in fact, Go was created because of this.

Concurrent Programming

tough, maddening, fun, $$$

caveats, gotchas, & head-scratches

today: **quick overview** of
basic synchronization primitives

to master concurrent programming, (i.e. to utilize modern HW well), you need a solid understanding of basic sync primitives offered by HW.

want more: MSc courses on this.

Practical Concurrent and Parallel Programming (Y1)
Performance of Computer Systems (Data Systems)

Outline

C-way of handling things.
concurrency / parallelism is **not** a feature of C; it's a feature of libc.
(recall: C isn't much; all interesting stuff is libraries)

- The necessity of concurrent programming
- **The problem with concurrent programming**
- Threads
- Synchronization

Concurrent vs. Parallel Programming

Parallel computing: many calculations, or execution of processes, are carried out **simultaneously**.

Concurrent computing: several processes are **in progress at the same time** (*concurrently) instead of one completing before next starts (sequentially)*

to drive home the **difference**:

- concurrent computing is the **illusion** of parallel computing; processes are actually *interleaved*.
- parallel computing **requires** HW support (multiple cores).

important to **understand the difference** (often debated, frequently asked)

why a lecture on concurrent (not parallel): parallel is an optimization of concurrent.

What Makes Concurrent Programming Hard?

1. **Identify Parallelizable Tasks:**
   Identify areas that can be divided into concurrent tasks (ideally independent).
2. **Balance:**
   Tasks should perform equal work of equal value.
3. **Data Splitting:**
   How to split data that is accessed by separate tasks?
4. **Data Dependency:**
   If data dependencies between different tasks => Synchronization needed.
5. **Testing, Debugging:**
   Many different execution paths possible, testing & debugging become more difficult.

Concurrency Anomalies

Classical problem classes of concurrent programs:

Race: outcome depends on a
elsewhere in the system

- Example: who gets the la
- Example: concurrent writ

Deadlock: improper resourc

- Example: traffic gridlock

Frederick Burr Opper *Alphonse and Gaston*. New York Journal, 1906. DEAR."

Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress

- Example: hallway dance (livelock)
- Example: people always jump in front of you in line

Outline

- The necessity of concurrent programming
- The problem with concurrent programming
- **Threads**
- Synchronization

Concurrency in C

- Processes (libc)
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing children
- Threads (libc)
  - Easy to share resources
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable
- I/O Multiplexing
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core

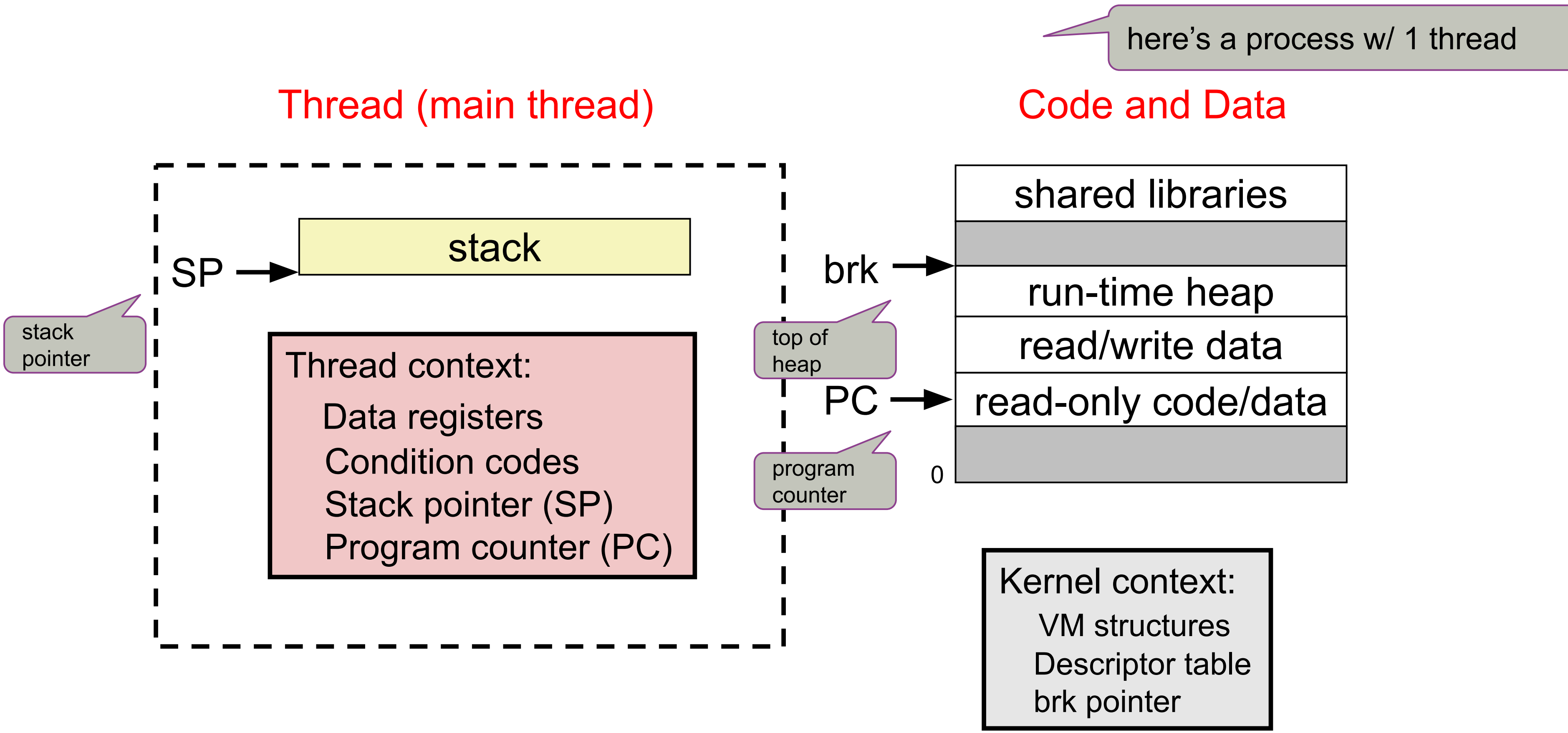
we will talk about these in a later lecture (fork, parent, children, synchronization (wait for each other), sharing across processes)

lighter form of process. instead of 2 processes w/ separate address spaces, you now have 1 process, w/ multiple threads that share address space. (separate stacks*, though)

in Linux, same data structures & mechanisms for these two

Threads

Process = thread + code, data, and kernel context



Threads

can have multiple threads in a process

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
    - Share common virtual address space (stack*)
  - Each thread has its own thread id (TID)

Thread 1 (main thread)

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Thread 2 (peer thread)

stack 2

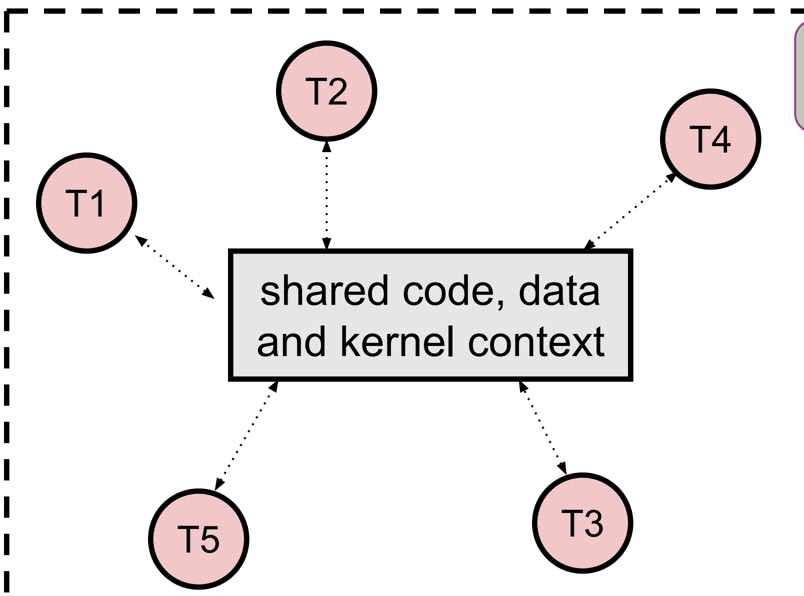
Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

*: stack is shared. but different SP
⇒ conceptually different stacks

Threads associated with process form a pool of peers

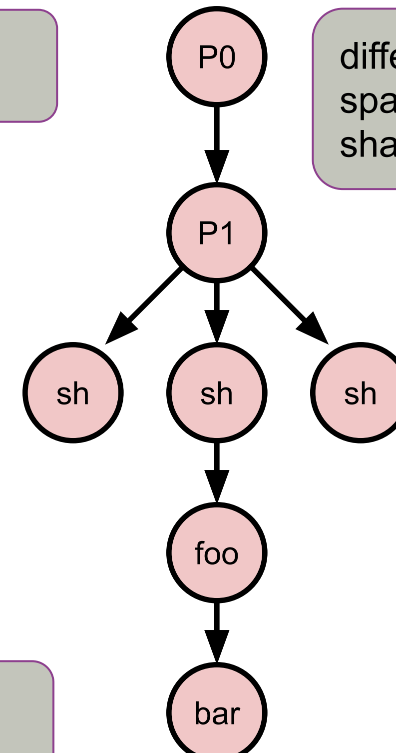
Unlike processes which form a tree hierarchy

Threads associated with process foo



same address space.

Process hierarchy



different address spaces. (can have shared memory)

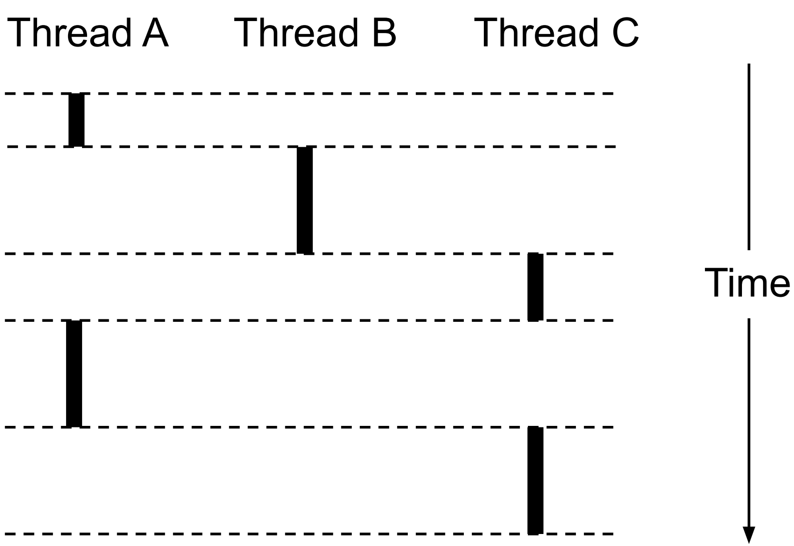
difference between process and thread is something that's easy for me to ask at the exam.

Thread Execution

illustrating the difference between concurrency and parallelism.

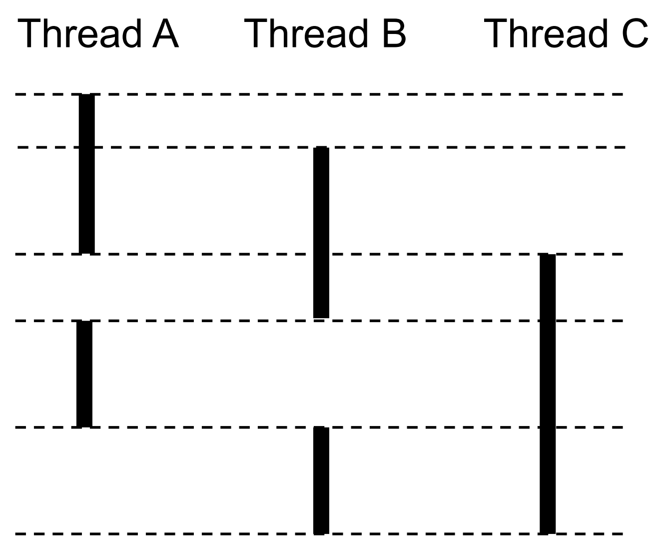
Single Core Processor

Simulate concurrency by time slicing



Multi-Core Processor

Parallel execution

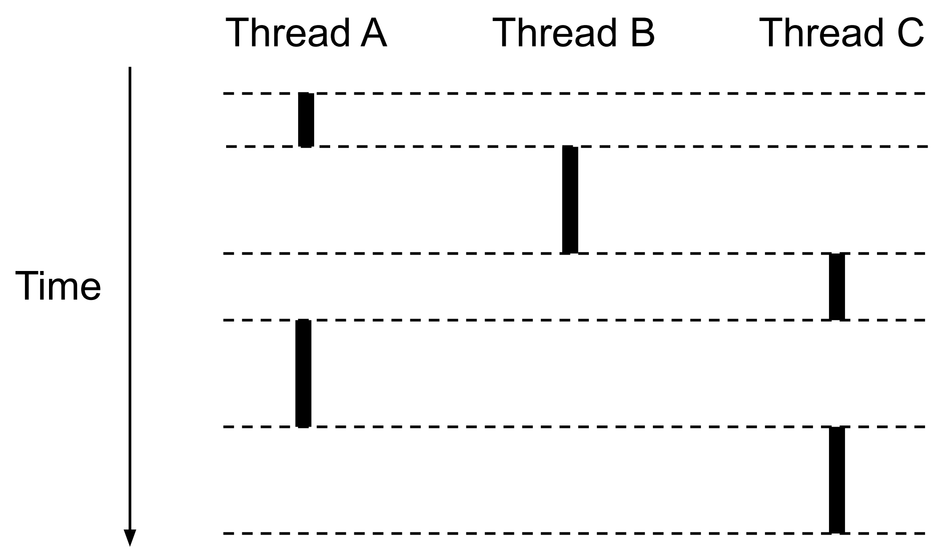


Run 3 threads on 2 cores

Logical Concurrency

- Two threads are (logically) concurrent if their flows overlap in time (otherwise, sequential)

- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



Posix Threads (Pthreads) Interface

thread interface in C given by Posix standard.

- *Pthreads* library: Standard interface of ~60 functions to manipulate threads from C.
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
    - `pthread_cond_init`
    - `pthread_cond_[timed]wait`

30s

one criticism of C: threads are not a beautiful concept, but a "fix".

https://www.gnu.org/software/libc/manual/html_mono/libc.html#POSIX-Threads

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```

declaration (see below)

creates a new thread (e.g. T5)

blocks until tid finishes

Thread attributes (usually NULL)

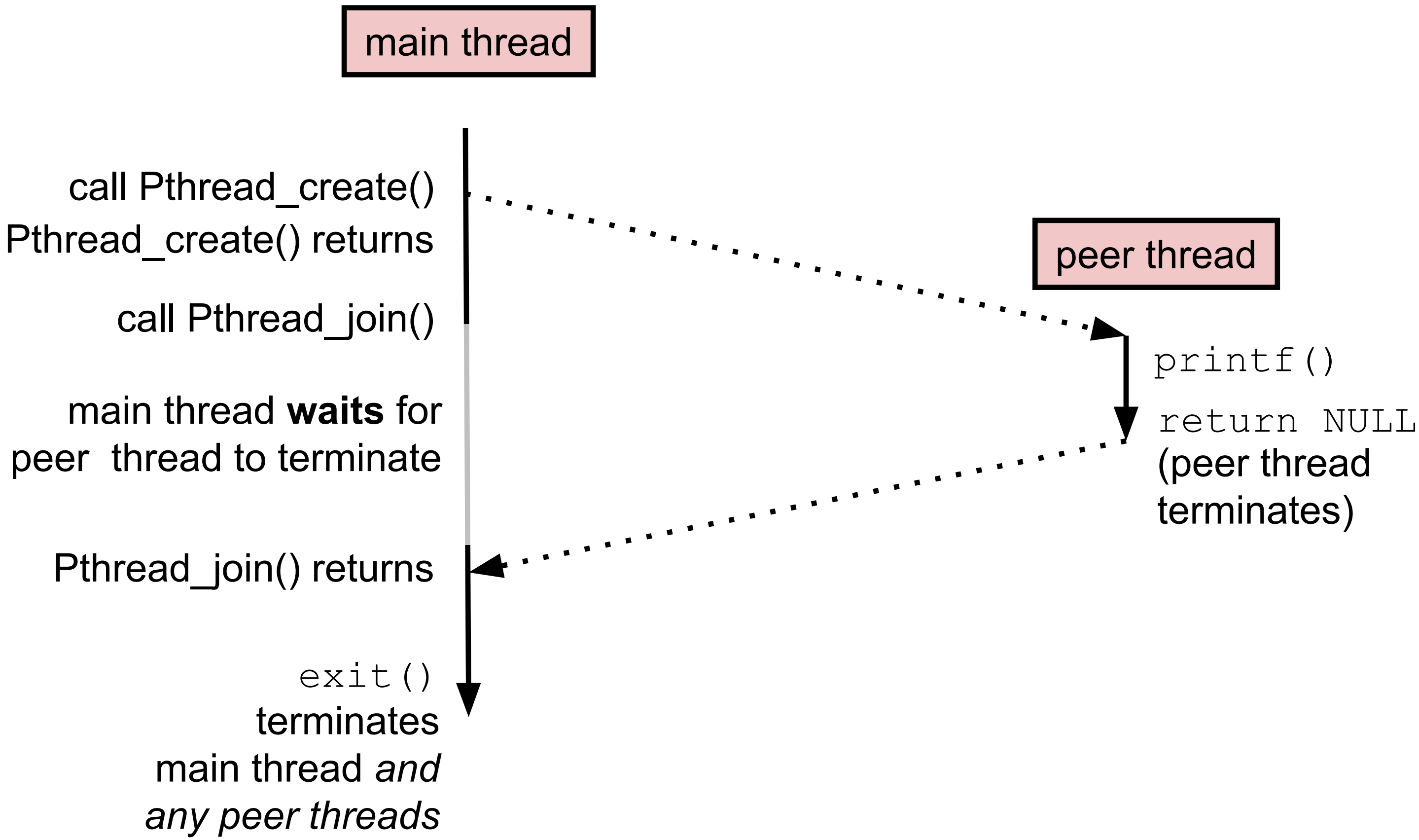
*Thread arguments (void *p)*

*return value (void **p)*

```
/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

http://csapp.cs.cmu.edu/3e/ics3/code/include/csapp.h
http://csapp.cs.cmu.edu/3e/ics3/code/src/csapp.c

Execution of Threaded "hello, world"



Shared Variables in Threaded C Programs

Threads must *synchronize* on shared data.

more on this in a bit.
for now:

Question: Which variables in a threaded C program are shared?

The answer is not as simple as
"*global variables are shared*" and
"*stack variables are private*"

Requires answers to the following questions:

What is the **memory model** for threads?

How are **instances** of variables **mapped to memory**?

How **many threads might reference** each of these instances?

Def: A variable `x` is *shared* if and only if
multiple threads reference some instance of `x`.

Threads Memory Model

Conceptual model:

Multiple threads run within the context of a single process

Each thread has its own separate thread context

- Thread ID, **stack, stack pointer, PC**, condition codes, GP registers

All threads share the remaining process context

- Code, data, heap,
  shared library segments of the process virtual address space
- Open files and installed handlers

Mapping Variable Instances to Memory

Global variables

Def: Variable declared outside of a function

Virtual memory contains exactly one instance of any global variable

Local variables

Def: Variable declared inside function without `static` attribute

Each thread stack contains one instance of each local variable

Local static variables

Def: Variable declared inside function with the `static` attribute

Virtual memory contains exactly one instance of any local static variable.

Example Program to Illustrate Sharing

what is shared?
not obvious.

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

loop index

thread id

array w/ 2 msgs

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int) vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++cnt);
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m, msgs.m`)

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++cnt);
}
```

Local static var: 1 instance (`cnt` [data])

Shared Variable Analysis

Which variables are shared?

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

Thread Local Storage (TLS)

https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Thread_002dLocal.html#Thread_002dLocal

modern version of libc, new keyword

New storage class keyword: __thread

One instance of the variable per thread

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

recommendation: to make clear *what should be shared* and *what should not*, use thread-local storage.

Crucial concept: Thread Safety

Functions called from a thread must be *thread-safe*

Def: A function is *thread-safe* iff it always produce correct results when called repeatedly from multiple concurrent threads.

Classes of **thread-unsafe** functions:

(despite accessing shared variables; fluke)

Class 1: Functions that do not protect shared variables.

Class 2: Functions that keep state across multiple invocations.

Class 3: Functions that return a pointer to a static variable.

Class 4: Functions that call thread-unsafe functions.

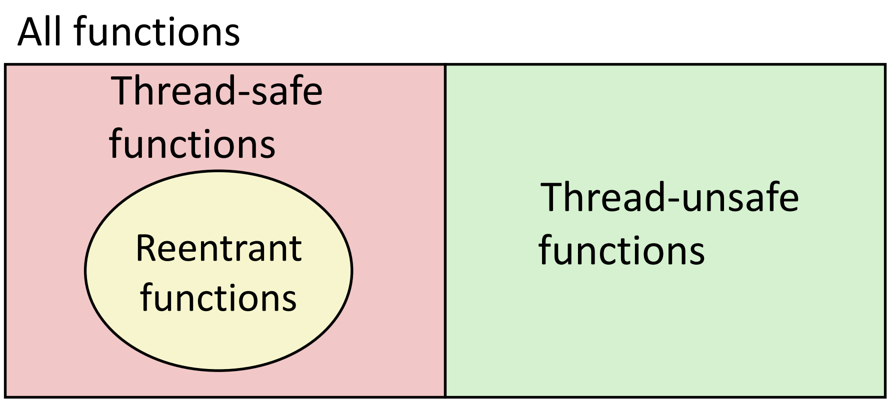
Reentrant Functions

cf. **referential transparency**: output of function always the same for a given input.

sanity

Def: A function is *reentrant* iff
it accesses no shared variables when called by multiple threads.

- Important subset of thread-safe functions.
- Require no synchronization operations.



there is another definition of reentrant which makes it a subset of both thread-safe and thread-unsafe. we will be using the above definition.

Outline

- The necessity of concurrent programming
- The problem with concurrent programming
- Threads
- **Synchronization**
  - Doorbell, Mutex, Conditional variable, Semaphore

Def: special shared variable that guarantees that a data structure can only be accessed atomically

HW synchronization

thread synchronization primitives

Synchronization Issues

Ancient Greek “ἄτομος” (atomos, “indivisible”)

Thread 1:

this is just a few assignments to two variables! think about full programs.

```
func foo() {
    x++;
    y = x;
}
```

Thread 2:

```
func bar() {
    y++;
    x+=3;
}
```

If the initial state is x = 6, y = 0,
what happens after these threads finish running?

Q: what are possible final values of x and y?

example of **data race**
aka. **race condition**
(notoriously hard to debug!)

Synchronization Issues

30s

Many things that look like "one step" operations take several steps under the hood:

```
func foo() {
    eax = mem[x];
    inc eax;
    mem[x] = eax;
    ebx = mem[x];
    mem[y] = ebx;
}


func bar() {
    eax = mem[y];
    inc eax;
    mem[y] = eax;
    eax = mem[x];
    add eax, 3;
    mem[x] = eax;
}
```

to update `mem[x]`: (RMW)
1. read `mem[x]` into register,
2. op on register,
3. write from register to `mem[x]`.

this is **multi-step** (non-atomic).
`foo` can be in midst, while `bar` completes the three steps.
⇒ `foo` has stale `mem[x]` in its register.

(cache coherence (across cores) won't help)

to understand synchronization issues, must know how code is mapped to assembly.

When we run a multithreaded program, we don't know what order threads run in, nor do we know when they will be interrupted.

Synchronization

10s

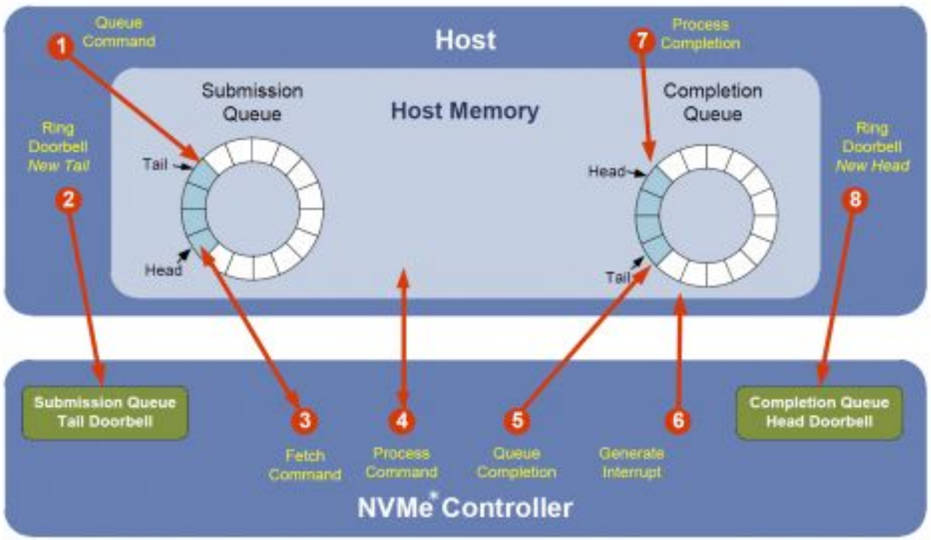
Synchronization is needed
when data structures are shared

HW synchronization: PCIe/NVMe Doorbell

30s

side note (low level sync)

doorbell is a boolean register



host software notifies storage device that data is ready

- SQ doorbell
- CQ doorbell

submission queue

completion queue

now, on to **thread synchronization primitives**

Thread Synchronization

read:
problems

how *do* threads
even synchronize at the
lowest level?

main reference: →

solutions are opaque,
solutions vary
between processors.

I'll give the gist of
common cases.

section 8.1
(quite opaque)

intel®

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 3A:
System Programming Guide, Part 1

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers*, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

gathered from scraps of info
from here and there.
why important: to be able to
debug performance problems

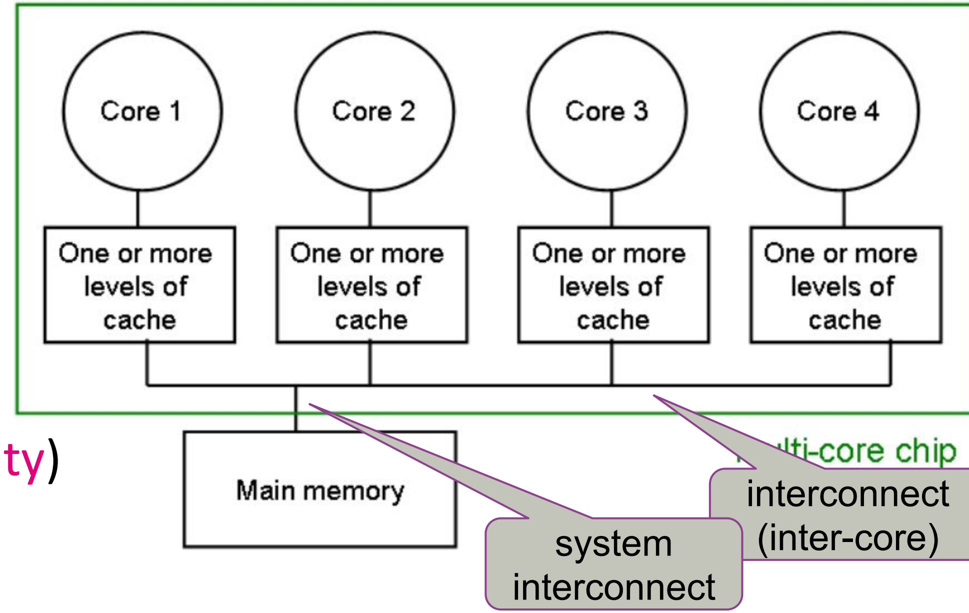
central concepts:

- **bus locking**
- **memory consistency**
- **cache coherence (next lecture)**

Bus Locking & Atomicity

cores share buses.

Q: core 1, 2 do an op simultaneously;
what happens? (need: atomicity)



bus locking prevents this.

"While [LOCK#] signal is asserted, requests from other processors or bus agents for control of the bus are blocked."

(many cores assert lock simultaneously ⇒ bus arbiter decides priority)

guaranteed atomic: read, write. (more later)

can otherwise state desire for instr. to be atomic.

note on **alignment**: data unaligned ⇒ read might fetch two cache lines. ⇒ lock asserted for longer (**slow**)

LOCK instruction prefix, example

88

I saw some x86 assembly in Qt's source:

```
q_atomic_increment:
    movl 4(%esp), %ecx
    lock
    incl (%ecx)
    mov $0,%eax
    setne %al
    ret

    .align 4,0x90
    .type q_atomic_increment,@function
    .size   q_atomic_increment,.-q_atomic_increment
```

1. From Googling, I knew `lock` instruction will cause CPU to lock the bus, but I don't know when CPU frees the bus?
2. About the whole above code, I don't understand how this code implements the `Add`?

129

1. `LOCK` is not an instruction itself: it is an instruction prefix, which applies to the following instruction. That instruction must be something that does a read-modify-write on memory (`INC`, `XCHG`, `CMPXCHG` etc.) --- in this case it is the `incl (%ecx)` instruction which `inc`rements the `l`ong word at the address held in the `ecx` register.

   The `LOCK` prefix ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation, and provides certain additional ordering guarantees. This may be achieved by asserting a bus lock, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.
2. This code copies the address of the variable to be incremented off the stack into the `ecx` register, then it does `lock incl (%ecx)` to atomically increment that variable by 1. The next two instructions set the `eax` register (which holds the return value from the function) to 0 if the new value of the variable is 0, and 1 otherwise. The operation is an **increment**, not an add (hence the name).

Instruction reordering due to bus locks

out-of-order execution

ex: access to memory is 100x more expensive than L1 cache.

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes line	On-Chip L1	1	Hardware
L2 cache	64-bytes line	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

core 1 asserts bus lock to access mem, core 2 wants access to mem
⇒ core 2 must wait for a **really** long time. *next instructions need mem*?

Memory [Consistency] Models

other models: sequential consistency, acquire/release, relaxed.
x86 has a strong memory model, w/ a wee bit of reordering.

which instructions reorders can take place?

weak memory model: R/Ws can be reordered arbitrarily as long as behavior of isolated thread unaffected.

- compiler, CPU core (← weak HW memory model)

sometimes order matters.

Thread #1 Core #1:
```
while (f == 0);
// Memory fence required here
print x;
```
Thread #2 Core #2:
```
x = 42;
// Memory fence required here
f = 1;
```

ex: NVMe I/O
ex (silly): DMA to robotic surgeon

to prevent reordering (when important):
memory barriers. (sync)

volatile keyword in C prevents statement from being reordered / skipped.
(anecdote: password in Windows)

Atomic CPU Operations

& Posix

synchronization mechanisms are based on **shared variables** and **atomic instructions**.

- Atomic CPU instructions:
  - Fetch and Add
  - Compare and Swap
  - Test and Set
  - Memory Barrier: operations placed before the barrier are guaranteed to execute before operations placed after the barrier. (aka. “fence”)

you have abstractions for the above

- In GCC:

  https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html
  - `__sync_`**`fetch_and_{`**`sub, or, and, xor, nand`**`}`**`()`
  - `__sync_`**`{`**`bool, val`**`}_compare_and_swap`**`()`
  - `__sync_lock_`**`test_and`**`_set, __sync_lock_release`
  - `__sync_synchronize()`

Mutex, Implementation?

API for providing
mutually **ex**clusive access to a resource

Thread 1:

critical section

```
void foo() {
    mutex.lock();
    x++;
    y = x;
    mutex.unlock();
}
```

Thread 2:

critical section

```
void bar() {
    mutex.lock();
    y++;
    x+=3;
    mutex.unlock();
}
```

Global mutex guards access to x & y.

that's nice. how to implement?

Can we do something like this? (easy?)

```
static unsigned int lockvar = 0;
static void lock() {          // acquire
    while ( lockvar ) {}
    lockvar = 1;
}
static void unlock() {        // release
    lockvar = 0;
}
```

Mutex

Thread 1:

```
void foo() {
    mutex.lock();
    x++;
    y = x;
    mutex.unlock();
}
```

In C:
pthread_mutex_t lock;

pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);

lock variable

Thread 2:

```
void bar() {
    mutex.lock();
    y++;
    x+=3;
    mutex.unlock();
}
```

Global mutex guards access to x & y.

(implementation on next slide)

Mutex, sample implementation (w/ spinlock)

lock is a datastructure (unsigned int) which is set to 0 or 1 w/ compare and swap (atomic).

area of memory

```
static inline void _lock(unsigned int *lock) {
    while (1) {
        int i;
        for (i=0; i < 10000; i++) {
            if (__sync_bool_compare_and_swap(lock, 0, 1)) {
                return;
            }
        }
        sched_yield();
    }
}

static inline void _unlock(unsigned int *lock) {
        __sync_bool_compare_and_swap(lock, 1, 0);
}
```

now you see why it's called a spinlock

lock is a datastructure (unsigned int) which is set to 0 or 1 w/ compare and swap (atomic).

usage: you take the lock before accessing the shared variable. when done, you unlock.
guarantee: only 1 thread gets the lock. (guaranteed by CPU instr.)

thread yields the core; someone else takes over (hopefully the thread that held the lock)

important if you only have a single core!

https://idea.popcount.org/2012-09-12-reinventing-spinlocks/

Condition Variable

signal threads that a condition is true.
(implemented using mutex)

```
// safely examine the condition, prevent other threads from
// altering it
pthread_mutex_lock (&lock);
while ( SOME-CONDITION is false)
    pthread_cond_wait (&cond, &lock);

// Do whatever you need to do when condition becomes true
do_stuff();
pthread_mutex_unlock (&lock);
```

block the thread. will unblock when
1) signal received on cond, 2) mutex unlocked
after which, the thread will hold the mutex.

```
// ensure we have exclusive access to whathever comprises the condition
pthread_mutex_lock (&lock);

ALTER-CONDITION

// Wakeup at least one of the threads that are waiting on the condition (if any)
pthread_cond_signal (&cond);

// allow others to proceed
pthread_mutex_unlock (&lock)
```

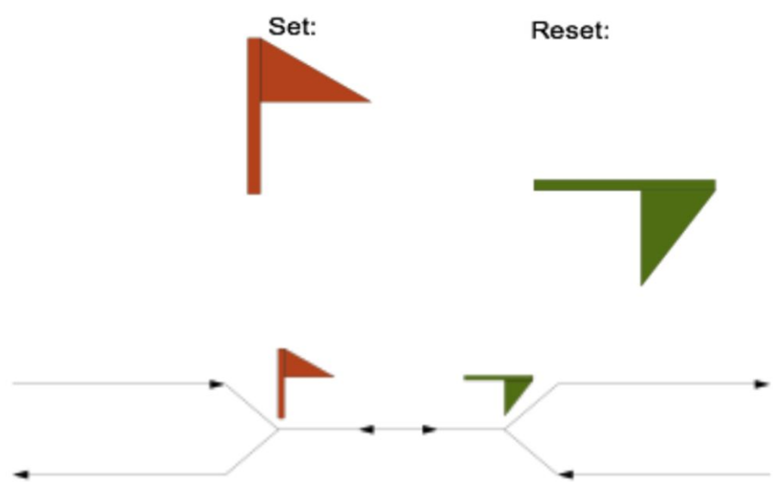
pthread_cond_broadcast() function shall unblock all threads currently blocked on the specified condition variable *cond*.

https://stackoverflow.com/questions/20772476/when-to-use-pthread-conditional-variables

Semaphore

limit how many threads can access resource at a time: **semaphore**

- A semaphore is a flag that can be raised or lowered in one step.
- Semaphores were flags that railroad engineers would use when entering a shared track.



For more see Edsger W. Dijkstra: Cooperating sequential processes.

Semaphore

- Semaphore restricts the **number** of simultaneous threads accessing a shared resource.
    - Semaphore = counter + mutex + wait_queue
- For a binary semaphore (= mutex + conditional variable)
    - **wait**() and **signal**() can be thought of as lock() and unlock()
    - Calls to lock() when the semaphore is already locked cause the thread to block.
- Pitfalls:
    - Must "bind" semaphores to particular objects; must remember to unlock correctly
    - Mutex can only be unlocked by thread that locked it, semaphore can be signaled from any thread => used for synchronization.

Take-Aways

concurrency: **illusion** of parallel

Concurrent Programming is a **necessity** on today's hardware.

Concurrency is **not** a first-class citizen in C; as opposed to languages based on communicating sequential processes (e.g., golang), actor languages (e.g., erlang).

Concurrency in C is based on **multi-threading**.

Communication necessary across threads:

- message passing, shared memory.

Classical problems of concurrent programs:

- races, deadlocks, starvation.

Synchronization primitives needed to avoid problems in concurrent programs:

- mutex, semaphore, conditional variable.

Synchronization primitives require hardware support:

- fetch-and-add, compare-and-swap, test-and-set, memory-barrier.

Other important concepts:

- reentrant, memory model, cache coherence, bus locking, thrashing, critical section

Further reading

section 8.1
(quite opaque)

intel®

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 3A: System Programming Guide, Part 1

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers*, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Is Parallel Programming Hard, and, if so, What Can You Do About It?

Edited by Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com