6.1800 Spring 2024

Lecture #5: Threads

understanding the “most mysterious code” in an OS

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

6.1800 in the news

how do we decide what features to add to a system?

can we correctly predict how features will be used? who/what they'll impact?

who gets to make these decisions?

Examining Emoji Color Spaces: A Strategy for Improving the Coverage of Heart Emoji

Author: Jennifer Daniel on behalf of the Unicode Emoji Subcommittee
To: Unicode Technical Committee
Date: November 15, 2020
Last Updated: April 11, 2021

🩷❤️🧡💛💚🩵💙💜🤎🖤🩶🤍

I. Background

Hearts are among the most frequently used emoji. Users of emoji often juxtapose the existing nine colored-heart emoji next to each other to denote markers of emotion, identity or affiliation that are not represented with atomic emoji in the Unicode Standard (ex. Support of Belarus: 🤍❤️🤍, Bisexuality: 💖💙💜, etc.).[1] It has also recently come to our attention that colored hearts are the most common emoji used in non-messaging spaces like Twitter bios.

Identifying emoji additions that have multiple-uses is increasingly important as emoji gain in popularity. The more emoji can operate as building blocks instead of specific images the more versatile, fluid, and useful they become.

https://www.unicode.org/L2/L2021/21075-heart-emoji-coverage.pdf

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** with each other → **bounded buffers** (virtualize communication links)
3. programs should be able to **share a CPU** without one program halting the progress of the others → **threads** (virtualize processors)

today's goal: implement **threads**, which allow multiple programs to share a CPU

a **thread** is a virtual processor
can *suspend* and *resume* a thread

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  // spin until it's safe to send
  while bb.in - bb.out >= N:
      release(bb.lock)
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  return
```

question: what are we hoping will happen in between `release(bb.lock)` and `acquire(bb.lock)`?

a **thread** is a virtual processor
can *suspend* and *resume* a thread

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  // spin until it's safe to send
  while bb.in - bb.out >= N:
      release(bb.lock)
      yield()
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  return
```

`yield()`'s job is to suspend the current thread and resume another* thread; our first job today is to understand what that means

*there are cases where `yield()` might suspend the current thread and end up resuming the same thread; that's okay

yield() suspends the running thread, chooses a new thread to run, and resumes the new thread

t_lock makes **yield()** an atomic action

threads is a table that contains information about each of the current threads

for each thread it stores the thread's
- state: RUNNABLE, RUNNING
- stack pointer (sp)
- page table register (ptr)

cpus is a table that keeps track of the id of the thread currently running on each cpu

SP = current stack pointer
PTR = current page table register
CPU = current cpu

```
yield():
  acquire(t_lock)

  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].state = RUNNABLE
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  release(t_lock)
```

yield() suspends the running thread, chooses a new thread to run, and resumes the new thread

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  while bb.in - bb.out >= N:
      release(bb.lock)
      yield()
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  return
```

performance concern: if the processor resumes the sending thread before any thread has called `receive()`, the buffer will still be full, and the sending thread will resume, but immediately `yield` again

```
yield():
  acquire(t_lock)

  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].state = RUNNABLE
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  release(t_lock)
```

`yield()` suspends the running thread, chooses a new thread to run, and resumes the new thread

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  while bb.in - bb.out >= N:
      release(bb.lock)
      yield()
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  return
```

it would be nice if `send()` could indicate "`yield`, and don't resume this thread until there's room in the buffer"

condition variables let threads wait for events ("conditions"), and get notified when they occur.
can **wait** on a condition, and be **notified** of it occurring

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  while bb.in - bb.out >= N:
      release(bb.lock)
      wait(bb.has_space)
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  notify(bb.has_message)
  return
```

```
// receive a message from bb
receive(bb):
  acquire(bb.lock)
  while bb.out >= bb.in:
      release(bb.lock)
      wait( ? )
      acquire(bb.lock)
  message <- bb.buf[bb.out mod N]
  bb.out <- bb.out + 1
  release(bb.lock)
  notify( ? )
  return message
```

new variables in use

bb.has_space = indicates that the buffer is not full (and so has space for at least one message)

bb.has_message = indicates that the buffer has at least one message in it

condition variables let threads wait for events ("conditions"), and get notified when they occur.
can **`wait`** on a condition, and be **`notified`** of it occurring

```
// send a message by placing it in bb
1:  send(bb, message):
2:    acquire(bb.lock)
3:    while bb.in - bb.out >= N:
4:        release(bb.lock)
5:        wait(bb.has_space)
6:        acquire(bb.lock)
7:    bb.buf[bb.in mod N] <- message
8:    bb.in <- bb.in + 1
9:    release(bb.lock)
10:   notify(bb.has_message)
11:   return
```

```
// receive a message from bb
receive(bb):
  acquire(bb.lock)
  while bb.out >= bb.in:
      release(bb.lock)
      wait(bb.has_message)
      acquire(bb.lock)
  message <- bb.buf[bb.out mod N]
  bb.out <- bb.out + 1
  release(bb.lock)
  notify(bb.has_space)
  return message
```

question: what happens if `send()` is interrupted between lines 4 and 5?

`bb.has_space = indicates that the buffer is not full (and so has space for at least one message)`
`bb.has_message = indicates that the buffer has at least one message in it`

condition variables let threads wait for events ("conditions"), and get notified when they occur.
can **wait** on a condition, and be **notified** of it occurring

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  while bb.in - bb.out >= N:
      release(bb.lock)
      wait(bb.has_space)
      acquire(bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  notify(bb.has_message)
  return
```

```
// receive a message from bb
receive(bb):
  acquire(bb.lock)
  while bb.out >= bb.in:
      release(bb.lock)
      wait(bb.has_message)
      acquire(bb.lock)
  message <- bb.buf[bb.out mod N]
  bb.out <- bb.out + 1
  release(bb.lock)
  notify(bb.has_space)
  return message
```

problem: lost notify

```
bb.has_space = indicates that the buffer is not full (and so has space for at least one message)
bb.has_message = indicates that the buffer has at least one message in it
```

condition variables let threads wait for events ("conditions"), and get notified when they occur.
can **wait** on a condition, and be **notified** of it occurring

```
// send a message by placing it in bb
send(bb, message):
  acquire(bb.lock)
  while bb.in - bb.out >= N:
      wait(bb.has_space, bb.lock)
  bb.buf[bb.in mod N] <- message
  bb.in <- bb.in + 1
  release(bb.lock)
  notify(bb.has_message)
  return
```

condition variable API:

`wait(cv,lock)`: yield processor, release `lock`, wait to be notified of `cv`

`notify(cv)`: notify waiting threads of `cv`

our second job today is to understand how **wait()** and **notify()** work, and also where **yield()** ends up in all of this

```
bb.has_space = indicates that the buffer is not full (and so has space for at least one message)
bb.has_message = indicates that the buffer has at least one message in it
```

wait(cv, lock) releases **lock**, sets the current thread to be waiting on **cv**, yields, and then re-acquires **lock**

`t_lock` makes **yield()** and **wait()** atomic actions

`threads` is a table that contains information about each of the current threads

for each thread it stores the thread's
- state: RUNNABLE, RUNNING, WAITING
- stack pointer (sp)
- page table register (ptr)
- condition to be notified of (cv)

`cpus` is a table that keeps track of the id of the thread currently running on each cpu

SP = current stack pointer
PTR = current page table register
CPU = current cpu

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

for right now, you can assume that **yield_wait()** is the same as **yield()**

we're giving it a different name, because we're going to find that it needs to be a slightly different function

condition variables let threads wait for events ("conditions"), and get notified when they occur
can **wait** on a condition, and be **notified** of it occurring

```
notify(cv):
  acquire(t_lock)
  for id = 0 to N-1:
    if threads[id].cv == cv &&
       threads[id].state == WAITING:
      threads[id].state = RUNNABLE
  release(t_lock)
```

`notify(cv)` finds all threads waiting on `cv`, and sets their state to `RUNNABLE` (i.e., ready to be run; *not* `RUNNING`)

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

we're going to get back to **`yield_wait()`** in a second, but just for context, here's how **`notify()`** works

yield_wait() is the version of yield() called by wait(); it functions similarly to yield()

but let's find out why it needs to be slightly different

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

problem: wait() holds t_lock

```
yield_wait():
  acquire(t_lock)

  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].state = RUNNABLE
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  release(t_lock)
```

`yield_wait()` is the version of `yield()` called by `wait()`; it functions similarly to `yield()`

but let's find out why it needs to be slightly different

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

problem: current thread's state shouldn't be set to `RUNNABLE`

(`wait()` has already set it to `WAITING`)

```
yield_wait():
  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].state = RUNNABLE
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

yield_wait() is the version of **yield()** called by **wait()**; it functions similarly to **yield()**

but let's find out why it needs to be slightly different

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

problem: deadlock

(`wait()` holds `t_lock`, but `notify()` also needs it)

```
yield_wait():
  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

yield_wait() is the version of **yield()** called by **wait()**; it functions similarly to **yield()**

but let's find out why it needs to be slightly different

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

problem: stack corruption

```
yield_wait():
  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
    release(t_lock)
    acquire(t_lock)
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

yield_wait() is the version of **yield()** called by **wait()**; it functions similarly to **yield()**

but let's find out why it needs to be slightly different

```
wait(cv, lock):
  acquire(t_lock)
  release(lock)
  id = cpus[CPU].thread
  threads[id].cv = cv
  threads[id].state = WAITING
  yield_wait()
  release(t_lock)
  acquire(lock)
```

```
yield_wait():
  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].sp = SP
  threads[id].ptr = PTR
  SP = cpus[CPU].stack

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
    release(t_lock)
    acquire(t_lock)
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id
```

we've done so much work. but what if threads just never call `wait()` (or `yield()`)?

preemption: forcibly interrupt threads

```
timer_interrupt():
  push PC
  push registers
  yield()
  pop registers
  pop PC
```

problem: what if timer interrupt occurs while running `yield()` or `yield_wait()`?

solution: hardware mechanism to disable interrupts

notice the need for some assistance from hardware here! just like how we required an atomic exchange operation for locks, and how we require the physical addresses of the page table registers for virtual memory to work

(and also similar to how the IP addresses of the DNS root servers are hardcoded into clients)

choosing a new thread to run is the problem of **scheduling**

```
yield():
  acquire(t_lock)

  // Suspend the running thread
  id = cpus[CPU].thread
  threads[id].state = RUNNABLE
  threads[id].sp = SP
  threads[id].ptr = PTR

  // Choose a new thread to run
  do:
    id = (id + 1) mod N
  while threads[id].state != RUNNABLE

  // Resume the new thread
  SP = threads[id].sp
  PTR = threads[id].ptr
  threads[id].state = RUNNING
  cpus[CPU].thread = id

  release(t_lock)
```

first-come first-serve: whichever thread yielded first is scheduled first

priority scheduling: threads that *need* to finish sooner are scheduled before threads that can be scheduled later

shortest remaining time first: threads that need the least amount of time to finish are scheduled first

round robin: assign a *quantum* of time per thread, and schedule threads to get one quantum in a "round robin" order; repeat as needed

how threads are scheduled has a large **impact** on performance and **fairness**; there is no *best* scheduling algorithm

operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → virtualize memory

2. programs should be able to **communicate** with each other → bounded buffers (virtualize communication links)

3. programs should be able to **share a CPU** without one program halting the progress of the others → **threads** (virtualize processors)

threads virtualize a processor so that we can share it among programs. **`yield()`** allows the kernel to suspend the current thread and resume another

condition variables provide a more efficient API for threads, where they **wait** for an event and are **notified** when it occurs. **`wait()`** requires a new version of **`yield()`**, **`yield_wait()`**

preemption forces a thread to be interrupted so that the kernel doesn't have to rely on programmers correctly using **`yield()`**. requires a special **interrupt** and hardware support to disable other interrupts