6.033 in the news

As Automakers Add Technology to Cars, Software Bugs Follow

Faulty computer systems are prompting class-action lawsuits by disgruntled car owners, a symptom of automakers’ bumpy transition to the digital age.

“After every update we get complaints CarPlay is not working,” said Serhat Kurt, who operates a website, macReports, that provides advice on fixing problems with Apple devices.

Mr. Kurt faulted both the carmakers and Apple — the carmakers for being “not very good with software,” and Apple for not doing enough to ensure that software updates work with older vehicles.

questions that arise:

how do we come to understand the impacts of our design decisions, especially ones we don’t tend to think about as much (e.g., how software update cycles sync — or don’t sync — with other development timelines)

what incentivizes designers to make systems backwards-compatible? how do we decide how far back to go?

6.033 Spring 2022

Lecture #5: Threads
understanding the “most mysterious code” in an OS
**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**

2. programs should be able to **communicate** with each other

3. programs should be able to **share a CPU** without one program halting the progress of the others

   ───► **virtual memory**

   ───► **bounded buffers**
   (virtualize communication links)

   ───► assume one program per CPU
   (for today)
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

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others

---

today’s goal: implement threads, which allow multiple programs to share a CPU
thread: a virtual processor
a **thread** is a virtual processor

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

```plaintext
// send a message by placing it in bb
send(bb, message):
    acquire(bb.lock)
    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
```
a **thread** is a virtual processor

can **suspend** and **resume** a thread

```python
// 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
```
A thread is a virtual processor. a thread can suspend and resume a 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

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

yield():
    // Suspend the running thread
    // Choose a new thread to run
    // Resume the new thread
yield() suspends the running thread, chooses a new thread to run, and resumes the new thread.

`t_lock` makes `yield()` an atomic action.

```python
yield():
    acquire(t_lock)

    // Suspend the running thread
    // Choose a new thread to run
    // Resume the new thread

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

```plaintext
yield():
    acquire(t_lock)
    // Suspend the running thread
    // Choose a new thread to run
    // Resume the new thread
    release(t_lock)
```

t_lock makes yield() an atomic action.

threads is a table that contains information about each of the current threads.
yield() suspends the running thread, chooses a new thread to run, and resumes the new thread.

```
yield():
    acquire(t_lock)
    // Suspend the running thread
    // Choose a new thread to run
    // Resume the new thread
    release(t_lock)
```

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
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).

```c
yield():

acquire(t_lock)

// Suspend the running thread
// Choose a new thread to run
// Resume the new thread

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

```
yield():
    acquire(t_lock)
    // Suspend the running thread
    // Choose a new thread to run
    // Resume the new thread
    release(t_lock)
```

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)
**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

- state: RUNNABLE, RUNNING
- stack pointer (sp)
- page table register (ptr)

SP = current stack pointer
PTR = current page table register

**yield()**: acquire(**t_lock**)

// Suspend the running thread
id = id of current thread
**threads**[id].state = RUNNABLE
**threads**[id].sp = SP
**threads**[id].ptr = PTR

// Choose a new thread to run
// Resume the new thread
release(**t_lock**)
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.

<table>
<thead>
<tr>
<th>SP</th>
<th>PTR</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>current stack pointer</td>
<td>current page table register</td>
<td>current cpu</td>
</tr>
</tbody>
</table>

**yield()**:

```plaintext
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
// Resume the new thread
release(t_lock)
```
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

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

\[
t_{\text{lock}}\ \text{makes}\ \text{yield()}\ \text{an atomic action}
\]

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(\text{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:
\text{id} = (\text{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(\text{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

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.

```c
// 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 received a message, the buffer will still be full, and the sending thread will immediately yield again.

```c
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.

```plaintext
// 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
```

**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
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

**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

```c
// 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
```

```c
// 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

**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

```
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

**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

```c
// 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

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

```
wait(cv, lock):
    acquire(t_lock)
    release(lock)
    release(t_lock)
    acquire(lock)
```

\( t\_lock \) makes \texttt{yield()} and \texttt{wait()} atomic actions.

\texttt{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)

\texttt{cpus} is a table that keeps track of the id of the thread currently running on each cpu.

\texttt{SP} = current stack pointer
\texttt{PTR} = current page table register
\texttt{CPU} = current cpu
**wait(cv, lock)** releases lock, sets the current thread to be waiting on cv, yields, and then re-acquires lock

```plaintext
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
```

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

    release(t_lock)
    acquire(lock)
```
**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

```python
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)
```
**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

```python
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)
```

```python
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
**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.

```python
def 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()** 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

```plaintext
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)
```

```plaintext
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)
```

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
**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

```c
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)
```

**problem:** **wait()** holds **t_lock**

```c
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)
```

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

```c
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
```

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()** 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.

```plaintext
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
```

**problem:** current thread’s state shouldn’t be set to RUNNABLE (wait() has already set it to WAITING)
**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

```cpp
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
```

```cpp
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() 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

```
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
```

```
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()** 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.

```c
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

// 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
```
**problem:** stack corruption

```
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

   // 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

```c

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
```

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

**preemption**: forcibly interrupt threads
we’ve done so much work. but what if threads just never call \texttt{wait()} (or \texttt{yield()})?

\textbf{preemption:} forcibly interrupt threads

\begin{verbatim}
\texttt{timer\_interrupt()}:}
  push PC
  push registers
  yield()
  pop registers
  pop PC
\end{verbatim}
we’ve done so much work. but what if threads just never call wait() (or yield())?

**preemption**: forcibly interrupt threads

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

```c
timer_interrupt():
  push PC
  push registers
  yield()
  pop registers
  pop PC
```
we’ve done so much work. but what if threads just never call wait() (or yield())?

**preemption:** forcibly interrupt threads

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

**solution:** hardware mechanism to disable interrupts

timer_interrupt():
  push PC
  push registers
  yield()
  pop registers
  pop PC
we’ve done so much work. but what if threads just never call wait() (or yield())?

**preemption**: forcibly interrupt threads

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

**solution**: hardware mechanism to disable interrupts

**timer_interrupt()**:
- push PC
- push registers
- yield()
- pop registers
- pop PC

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**

```c
void 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)
}
```
choosing a new thread to run is the problem of **scheduling**

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

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

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

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

```c
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
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()
**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