6.033 Spring 2021

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

--- 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

--- 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
   → 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
thread: a virtual processor
**thread**: a virtual processor can *suspend* and *resume* a thread
thread: a virtual processor can suspend and resume a thread

// send a message by placing it in bb

```python
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
```
thread: a virtual processor 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
**thread**: a virtual processor can *suspend* and *resume* a thread

// send a message by placing it in bb

```plaintext
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 causes 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

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

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

\textit{threads} is a table that contains information about each of the current threads

yield():

\text{acquire}(t_{\text{lock}})

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

\text{release}(t_{\text{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

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.

- **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)

```
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

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)

```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.

\[ t\_lock \text{ makes } \text{yield()} \text{ an atomic action} \]

\[ \text{threads} \text{ is a table that contains information about each of the current threads} \]

\[ \text{for each thread it stores the thread’s} \]
\[ - \text{state: \text{RUNNABLE}, \text{RUNNING}} \]
\[ - \text{stack pointer (sp)} \]
\[ - \text{page table register (ptr)} \]

\[ \text{SP} = \text{current stack pointer} \]
\[ \text{PTR} = \text{current page table register} \]

\[ \text{yield():} \]
\[ \text{acquire(t\_lock)} \]

\[ \text{// Suspend the running thread} \]
\[ \text{id = id of current thread} \]
\[ \text{threads[id].state = \text{RUNNABLE}} \]
\[ \text{threads[id].sp = SP} \]
\[ \text{threads[id].ptr = PTR} \]

\[ \text{// Choose a new thread to run} \]
\[ \text{// Resume the new thread} \]

\[ \text{release(t\_lock)} \]
yield() suspends the running thread, chooses a new thread to run, and resumes the new thread

\[
\text{t\_lock} \text{ makes } \text{yield()} \text{ an atomic action}
\]

\[
\text{threads} \text{ is a table that contains information about each of the current threads}
\]

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

\[
\text{cpus} \text{ is a table that keeps track of the id of the thread currently running on each cpu}
\]

\[
\text{SP} = \text{current stack pointer}
\text{PTR} = \text{current page table register}
\text{CPU} = \text{current cpu}
\]

yield():

\[
\begin{align*}
\text{acquire(t\_lock)} \\
\text{// Suspend the running thread} \\
\text{id} &= \text{cpus[CPU]\_thread} \\
\text{threads[id]\_state} &= \text{RUNNABLE} \\
\text{threads[id]\_sp} &= \text{SP} \\
\text{threads[id]\_ptr} &= \text{PTR} \\
\text{// Choose a new thread to run} \\
\text{// Resume the new thread} \\
\text{release(t\_lock)}
\end{align*}
\]
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_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)

**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()**:  

```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
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(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.

// 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.

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

```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

// 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

```
// 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

**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:**

```c
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:

```c
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**() 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:
  - 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

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

    yield()

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

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

---

**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)
```

```
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)
```

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

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

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

```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)
```
\textbf{yield\textunderscore\_wait()} is the version of \textbf{yield()} called by \textbf{wait()}; it functions similarly to \textbf{yield()}

\textbf{wait(cv, lock)}:
\begin{itemize}
  \item acquire\textbf{(t\_lock)}
  \item release\textbf{(lock)}
  \item \textbf{id} = \textbf{cpus}[CPU].thread
  \item \textbf{threads}[id].cv = cv
  \item \textbf{threads}[id].state = WAITING
  \item \textbf{yield\textunderscore\_wait()}
  \item release\textbf{(t\_lock)}
  \item acquire\textbf{(lock)}
\end{itemize}

\textbf{problem:} \textbf{wait()} holds \textbf{t\_lock}

\textbf{yield\textunderscore\_wait():}
\begin{itemize}
  \item acquire\textbf{(t\_lock)}
  \item // Suspend the running thread
  \item \textbf{id} = \textbf{cpus}[CPU].thread
  \item \textbf{threads}[id].state = RUNNABLE
  \item \textbf{threads}[id].sp = SP
  \item \textbf{threads}[id].ptr = PTR
  \item // Choose a new thread to run
  \item do:
  \item \hspace{1em} \textbf{id} = (\textbf{id} + 1) \mod \textbf{N}
  \item \hspace{1em} while \textbf{threads}[id].state \neq \textbf{RUNNABLE}
  \item // Resume the new thread
  \item \hspace{1em} \textbf{SP} = \textbf{threads}[id].sp
  \item \hspace{1em} \textbf{PTR} = \textbf{threads}[id].ptr
  \item \hspace{1em} \textbf{threads}[id].state = \textbf{RUNNING}
  \item \hspace{1em} \textbf{cpus}[CPU].thread = \textbf{id}
  \item \hspace{1em} release\textbf{(t\_lock)}
\end{itemize}
yield_wait() is the version of yield() called by wait(); it functions similarly to yield():

```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()

```
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()

```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)
```

```c
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
```

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
**problem:** deadlock
(wait() holds `t_lock`, but `notify()` also needs it)

```c
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
```

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

```python
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
```

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

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 != RUNNING

    // 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()

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

\textbf{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

```c
void timer_interrupt()
{
    push PC
    push registers
    yield()
    pop registers
    pop PC
}
```

**problem**: what if timer interrupt occurs while running `yield()` or `yield_wait()`?
we’ve done so much work. but what if threads just never call `wait()` (or `yield()`)?

**preemption**: forcibly interrupt threads

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

```java
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**

```
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**

```
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**

```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
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