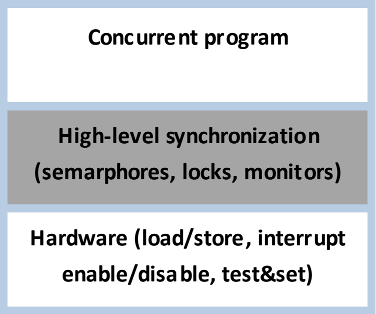
Implementing locks – atomicity in the thread library

Concurrent programs use high-level synchronization operations



Implementing these high-level synchronization operations

- Used by multiple threads so they need to worry about atomicity (e.g. they use data structures shared across threads)
- Can't use the high-level synchronization operations themselves

Use interrupt disable/enable to ensure atomicity

On uni-processor, operation is atomic as long as context switch doesn't occur in middle of the operation

- How does thread get context switched out?

- Prevent context switches at wrong time by preventing these events

With interrupt disable/enable to ensure atomicity, why do we need locks?

- User program could call interrupt disable before entering critical section and call interrupt enable after leaving critical section (and make sure not to call yield in the critical section)

Lock implementation #1 (disable interrupts with busy waiting)

```
lock () {
  disable interrupts
  while (value != FREE) {
    enable interrupts
    disable interrupts
  }
  value = BUSY
  enable interrupts
}

unlock () {
  disable interrupts
  value = FREE
  enable interrupts
}
```

Why does lock() disable interrupts in the beginning of the function?

Why is it ok to disable interrupts in lock()'s critical section (it wasn't ok to disable interrupts while user code was running)?

Do we need to disable interrupts in unlock()?

Why does the body of the while loop enable, then disable interrupts?

Another atomic primitive: read-modify-write instructions

Interrupt disable works on a uni-processor by preventing the current thread from being switched out.

But this doesn't work on a multi-processor

- Disabling interrupts on one processor doesn't prevent other processors from running
- Not acceptable (or provided) to modify interrupt disable to stop other processors from running

Could use atomic load/store instructions (remember Too Much Milk solution #3)

Modern processors provide an easier way with atomic read-modify-write instructions

- Atomically {reads value from memory into a register, then writes new value to that memory location}

Test_and_set: atomically writes 1 to a memory location (set) and returns the value that used to be there (test)

```
Test_and_set(X) {
  tmp = X
  X = 1
  return (tmp)
}
```

- Note that only 1 process can see a transition from 0 to 1

Exchange (x86)

- Swaps value between register and memory

Lock implementation #2 (test&set with busy waiting)

```
(value is initially 0)

lock () {
  while (test_and_set(value) == 1) {
  }
}

unlock () {
  value = 0
}
```

If lock is free (value = 0), test_and_set sets value to 1 and returns 0, so the while loop finishes.

If lock is busy (value = 1), test_and_set doesn't change the value and returns 1, so loop continues.

Busy Waiting

Problem with lock implementations #1 and #2

- Waiting thread uses lots of CPU time just checking for the lock to become free. This is called "busy waiting."
- Better for thread to go to sleep and let other threads run
- Strategy for reducing busy-waiting: integrate the lock implementation with the thread dispatcher data structure and have lock code manipulate thread queues

Lock implementation #3 (interrupt disable, no busy-waiting)

Waiting thread gives up processor so that other threads (e.g. the thread with the lock) can run more quickly. Someone wakes up thread when the lock is free.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY
  } else {
    add thread to queue of threads waiting for
      this lock

    switch to next runnable thread
  }
  enable interrupts
}

unlock () {
  disable interrupts
  value = FREE
  if (any thread is waiting for this lock) {
    move waiting thread from waiting queue to
      ready queue
    value = BUSY
  }
  enable interrupts
}
```

This is a **handoff lock**

- Thread calling unlock() gives lock to the waiting thread

- Why have a separate waiting queue? Why not put waiting thread onto the ready queue?

Interrupt disable/enable pattern

When should lock() re-enable interrupts before calling switch?

Enable interrupts before adding thread to wait queue?

```
lock () {
  disable interrupts
  …
  if (lock is busy) {
    enable interrupts
    add thread to lock wait queue
    switch to next runnable thread
  }
}
```

When could this fail?

Enable interrupts after adding thread to wait queue, but before switching to next thread?

```
lock () {
  disable interrupts
  …
  if (lock is busy) {
    add thread to lock wait queue
    enable interrupts
    switch to next runnable thread
  }
}
```

But this fails if interrupt happens after thread enable interrupts

- Lock() adds thread to wait queue
- Lock() enables interrupts
- Interrupts causes pre-emption, i.e. switch to another thread. Pre-emption moves thread to ready queue. Now thread is on two queues (wait and ready)!

Also, switch is likely to be a critical section

Adding thread to wait queue and switching to next thread must be atomic

Solution: waiting thread leaves interrupts disabled when it calls switch. Next thread to run has the responsibility of re-enabling interrupts before returning to user code. When waiting thread wakes up, it returns from switch with interrupts disabled (from the last thread)

Invariant

- All threads promise to have interrupts disabled when they call switch
- All threads promise to re-enable interrupts after they get returned from switch

```
Thread A                      Thread B

                              yield() {
                                disable interrupts
                                switch

  enable interrupts
}
<user code runs>
lock() {
  disable interrupts
  …
  switch
                                back from switch
                                enable interrupts
                              }
                              <user code runs>
                              unlock()(move thread
                                A to ready queue)
                              yield () {
                                disable interrupts
                                switch
  back from switch
  enable interrupts
}
```

Lock implementation #4 (test&set, minimal busy-waiting)

Can't implement locks using test&set without some amount of busy-waiting, but can minimize it

Idea: use busy-waiting only to atomically execute lock code. Give up CPU if busy.

```
lock() {
  while (test&set(guard)) {
  }

  if (value == FREE) {
    value = BUSY
  } else {
    Add thread to queue of threads waiting for
      this lock

    switch to next runnable thread
  }
  guard = 0
}
```

```
unlock() {
  while (test&set (guard)) {
  }

  value = FREE
  if (any thread is waiting for this lock) {
    move waiting thread from waiting queue to
      ready queue
    value = BUSY
  }
  guard = 0
}
```

Deadlock

Resources

- Something needed by a thread
- A thread **waits** for resources
- E.g. locks, disk space, memory, CPU

Deadlock

- A circular waiting for resources, leading to the threads involved not being able to make progress

Example

Thread A	Thread B
`lock(x)`	`lock(y)`
`lock(y)`	`lock(x)`
…	…
`unlock(y)`	`unlock(x)`
`unlock(x)`	`unlock(y)`

- Can deadlock occur with this code?

- Will deadlock always occur with this code?

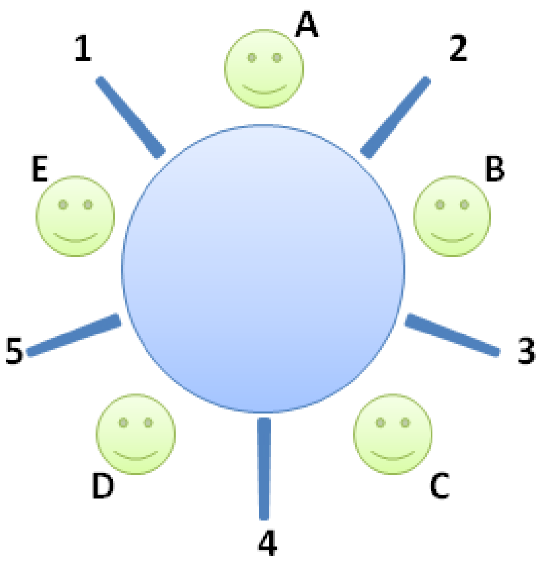
General structure of thread code

```
Phase 1. while (not done) {
           acquire some resources
           work
         }
Phase 2. Release all resources
```

Assume phase 1 has finite amount of work

Dining philosophers

Five philosophers sitting around a round table, 1 chopstick in between each pair of philosophers (five chopsticks total). Each philosopher needs two chopsticks to eat.



Algorithm for each philosopher

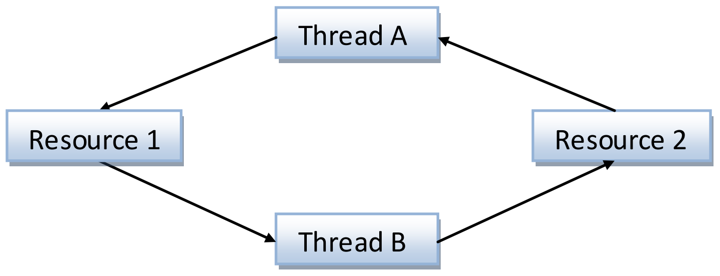
```
Wait for chopstick on right to be free, then pick it up
Wait for chopstick on left to be free, then pick it up
Eat
Put both chopsticks down
```

Can this deadlock?

Conditions for Deadlock

Four conditions must be true for deadlock to occur

- Limited resource: not enough resources to serve all threads simultaneously
- Hold and wait: threads hold resources while waiting to acquire other resources
- No pre-emption: thread system can't force thread to give up resource
- Circular chain of requests



Strategies for handling deadlock

Three general strategies

- Ignore
- Detect and fix
- Prevent

Detect and fix

- Can detect by looking for cycles in the wait-for graph
- How to fix once detected?

Deadlock prevention

Idea is to eliminate one of the four necessary conditions

Increase resources to decrease waiting (this minimizes the chance of deadlock)

Eliminate hold and wait

- Move resource acquisition to beginning

```
Phase 1a. acquire all resources
Phase 1b. while (not done) {
            acquire some resources
            work
          }
Phase 2.  release all resources
```

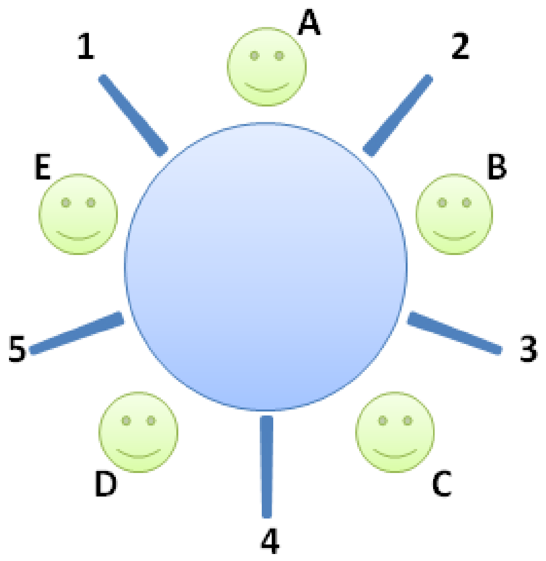
a. Wait until all resources you'll need are free, then grab them all at once (or) b. if you find resource busy, release all acquired resources and go back to beginning

Problems?

Allow pre-emption

- Can pre-empt CPU by saving its state to thread control block and resuming later
- Can pre-empt memory by swapping out memory to disk and loading it back later
- Can we pre-empt the holding of a lock?

Eliminate circular chain of requests



Banker's algorithm

Similar to reserving all resources at beginning, but more efficient

State maximum resource needs in advance (but don't actually acquire the resources). When thread later tries to acquire a resource, banker's algorithm determines when it's safe to satisfy the request (and blocks the thread when it's not safe).

General structure of the code

```
Phase 1a. state maximum resources needed
Phase 1b. while (not done) {
            Acquire some resources
            Work
          }
Phase 2.  Release all resources
```

Preventing deadlock by requesting all resources at beginning would block thread in phase 1a above (but phase 1b can proceed without waiting)

In banker's algorithm, phase 1a provides the information needed to determine when it's safe to satisfy each resource request in phase 1b.

"Safe" means guaranteeing the ability for all threads to finish (no possibility of deadlock)

Example: use banker's algorithm to model a bank loaning money to its customers

Bank has $6000. Customers sign up and establish a credit limit (maximum resources needed). They borrow money in stages (up to their credit limit). When they're done, they return all of their money.

Solution #1: bank gives money when requested, as long as money is available. Bank must reserve all resources when customer starts

```
Ann asks for credit limit of $2000
Bob asks for credit limit of $4000
Cat asks for credit limit of $6000
```

Can bank approve all these credit lines if it promises to give money upon request is money is available?

Solution #2: bank approves all credit limits, but customer may have to wait when actually asking for the money

```
Ann asks for credit limit of $2000 (bank oks)
Bob asks for credit limit of $4000 (bank oks)
Cat asks for credit limit of $6000 (bank oks)

Ann takes out $1000 (bank has $5000 left)
Bob takes out $2000 (bank has $3000 left)
Cat wants to take out $2000.  Is this allowed?
```

Allowed if and only if, after giving the money, there exists some sequential order of fulfilling all maximum resources (worst-case analysis)

- If give $2000 to Cat, bank has $1000 left
- Ann can finish even if she takes out her max (another $1000). When Ann finishes, she returns her money (bank has $2000).
- After Ann finishes, Bob can take out his max (another $2000), then finish
- Then Cat can finish, even if she takes out her max (another $4000)

What about this scenario?

```
Ann asks for credit limit of $2000 (bank oks)
Bob asks for credit limit of $4000 (bank oks)
Cat asks for credit limit of $6000 (bank oks)

Ann takes out $1000 (bank has $5000 left)
Bob takes out $2000 (bank has $3000 left)
Cat wants to take out $2500.  Is this allowed?
```

Banker allows system to over-commit resources without introducing the possibility of deadlock. Sum of max resource needs of all current threads can be greater than total resources, as long as there's some way for all the threads to finish without getting into deadlock.

How can we apply the banker's algorithm to dining philosophers?

Unfortunately, it is difficult to anticipate maximum resources needed

CPU scheduling

How should one choose the next thread to run? What are the goals of the CPU scheduler?

Minimize average response time

- Rate at which jobs complete in the system

Maximize throughput of the entire system

- Rate at which jobs complete in the system

Fairness

- Share CPU among thread in some "equitable" manner

First-come, first-served (FCFS)

FIFO ordering between jobs

No pre-emption (run until done)

- Thread runs until it calls yield() or blocks on I/O
- No timer interrupts

Pros and cons

+ simple

- Short jobs get stuck behind long jobs
- What about the user's interactive experience?

Example

- Job A takes 100 seconds
- Job B takes 1 second

```
Time 0  : Job A arrives and starts
Time 0+ : Job B arrives
Time 100: Job A finishes (response time = 100)
          Job B starts
Time 101: Job B finishes (response time 101)

Average response time = 100.5
```

Round robin

Goal: improve average response time for short jobs

Solution: periodically pre-empt all jobs (viz. long-running ones)

Is FCFS or round robin "fair"?

Example

- Job A takes 100 seconds
- Job B takes 1 second
- Time slice of 1 second (a job is pre-empted after running for 1 second)

```
Time 0  : Job A arrives and starts
Time 0+ : Job B arrives
Time 1  : Job A is pre-empted, Job B starts
Time 2  : Job B finishes (response time = 2)
          Job A resumes
Time 101: Job A finishes (response time = 101)

Average response time = 51.5
```

Does round robin always achieve lower average response time than FCFS?

Pros and cons

+ good for interactive computing

- round robin has more overhead due to context switches

How to choose time slice?

- Big time slice: degrades to FCFS
- Small time slice: more time spent context switching

- Typically a compromise, e.g. 10 milliseconds
- If context switch takes .1 ms, then round robin with 10 ms time slice wastes 1% of the CPU

STCF (shortest time to completion first)

STCF: run whatever job has the least amount of work to do before it finishes (or blocks for an I/O)

STCF-P: pre-emptive version of STCF

- If a new job arrives that has less work than the current job has remaining, then pre-empt the current job in favor of the new one

Idea is to finish short jobs first

- Improves response time of shorter jobs by a lot
- Doesn't hurt response time of longer jobs by too much

STCF gives optimal response time among pre-emptive policies (and non-pre-emptive policies)

I/O

- Is the following job a "short" job or a "long" job?

```
while (1) {
  use CPU for 1 ms
  use I/O for 10 ms
}
```

Pros and cons

- \+ Optimal response time
- \- Unfair. Short jobs can prevent long jobs from ever getting CPU time (starvation)
- \- Needs knowledge of future

STCF and STCF-P need knowledge of the future

- It is often hard to predict the future
- How do you find out the future time required by a job?

Example

```
Job A
  Compute for 1000 seconds

Job B
  Compute for 1000 seconds

Job C
  while (1) {
    use CPU for 1 ms
    use I/O for 10 ms
  }
```

C can use 91% of the disk by itself. A or B can use 100% of the CPU. What happens when we run them together?

Goal: keep both CPU and disk busy

FCFS

- If A or B run before C, they prevent C from issuing its disk I/O for up to 2000 seconds

Round robin with 100ms time slice (not to scale)

```
CA---------B---------CA---------B---------...
|--|                 |--|
C’s I/O               C’s I/O
```

- Disk is idle most of the time that A and B are running (about 10 ms disk for every 200 ms)

Round robin with 1ms time slice (also not to scale)

```
CABABABABABCABABABABABC...
|--------| |--------|
C’s I/O    C’s I/O
```

- C runs more often, so it can issue its disk I/O almost as soon as its last disk I/O is done
- Disk is utilized about 90% of the time
- Little effect on A or B’s performance
- General principle: first start the things that can run in parallel
- Problem: lots of context switches (+ context switch overhead)

STCF-P

- Runs C as soon as its disk I/O is done (because it has the next shortest CPU burst)

```
CA---------CA---------CA--------- ...
|--------| |--------| |---------|
C’s I/O    C’s I/O    C’s I/O
```

Real-time scheduling

So far, we've focused on average-case analysis (average response time, throughput)

Sometimes, the right goal is to get each job done before its deadline (irrelevant how far in advance of the deadline the job completes)

- Video or audio output. E.g. NTSC (National Television Standards Committee) outputs 1 TV frame every 33ms
- Control of physical systems, e.g. auto assembly, nuclear power plants

This requires worst-case analysis.

How do we do this in real life?

Earliest deadline first (EDF)

Always run the job that has the earliest deadline (i.e. the deadline coming up next)

If a new job arrives with an earlier deadline than the currently running job, pre-empt the running job and start the new one.

EDF is optimal—it will meet all deadlines if it is possible to do so

Example

- job A: takes 15 seconds, deadline is 20 seconds after entering system
- job B: takes 10 seconds, deadline is 30 seconds after entering system
- job C: takes 5 seconds, deadline is 10 seconds after entering system

```
time--->
  0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85
A +
B +
C +
```