Synchronization 4: Deadlock, Misc Lock Issues

Changelog

Changes made in this version not seen in first lecture:

- 4 October: livelock: move slides earlier (next to abort discussion)
- 4 October: revocable locks, Linux OOM killer: move slides earlier (next to steal discussion)
- 4 October: event-based programming: some single-threaded code: fix broken slide
- 4 October: add backup slides on dining philospher ordering/abort
- 5 October: deadlock detection with variable resources: iterate over threads with owned/requested resources only; not all threads

last time (1)

monitor intuition:

- mutex locked before touching anything
- find reasons to wait
- condition variable for each
- broadcast (or maybe signal) when changing condition

reader/writer lock

- many readers at a time
- one writer

last time (2)

reader-priority/writer-priotity

- want writers to go before readers? count writers
- change wait conditions to account for who else is waiting

choosing priorities generally

- can track whatever you want, wait while not true
- worst case: own queue with boolean + one condition variable/semaphore per waiter
- (probably duplicates internal queue of mutex/cond var)

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (b                                              k); }
    buffer.e
    pthread_
    pthread_
}

Consume() {
    pthread_
    while (b
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

error last time:

```
if (buffer.size() == buffer.capacity() - 1)
    pthread_cond_signal(&space_ready);
```

what if two waiting producers and

two consumers run right after each other

problem: only one woken up

potential fixes

unconditionally signal

- each consume allows one produce to go
- rely on condition variable knowing if no one is waiting

broadcast if buffer changed from full to not-full

- every thread waiting because it was full could go buffer it becomes full again

explicitly count number of waiting producers — buffer not full and waiter

how could I have avoided this?

question: who might be waiting when condition changes

almost always multiple threads!

if not broadcasting, explain why each waiting thread gets to go

my implicit non-explanation: queue will be full again first

- not actually true: can keep consuming before producers go

how could I have avoided this?

question: who might be waiting when condition changes

almost always multiple threads!

if not broadcasting, explain why each waiting thread gets to go

my implicit non-explanation: queue will be full again first
- not actually true: can keep consuming before producers go

alternate view: consuming causes what threads to go?
- not just when the buffer was full
- since if I empty the buffer by consuming...

last week's quiz

"after one processor finishes updating a value, another processor could still have an old version of the value cached"

invalid state $\rightarrow$ can never read it

generally, called "not cached"

from comments, significant number of people did not interpret it this way

life HW notes

some common ways students seem to get confused

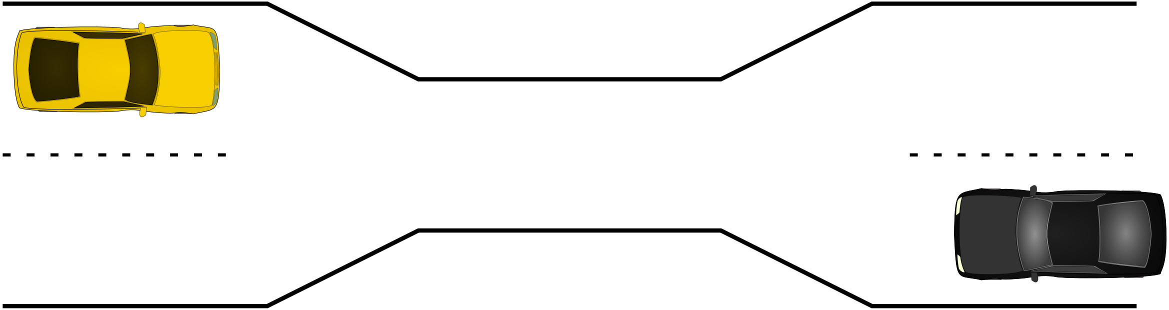
`LifeBoard my_copy; ... my_copy = state` makes a copy of state

> (even if my_copy is in a struct, etc.)

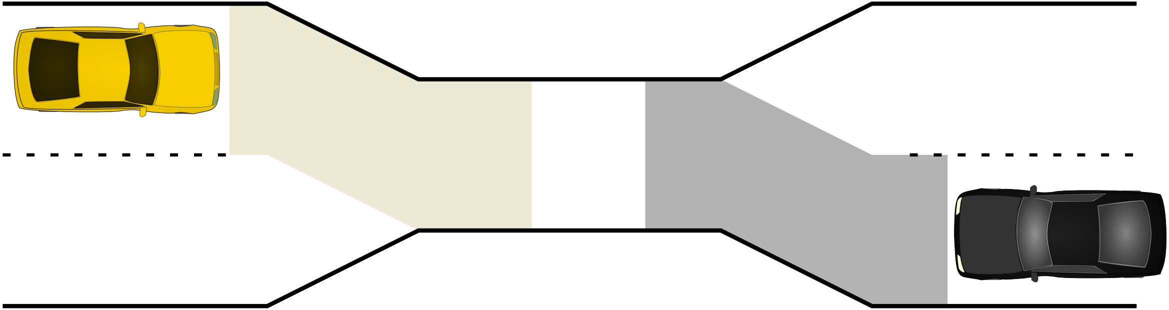
the simulate function modifies the state reference passed it

> you better change that LifeBoard to return the result

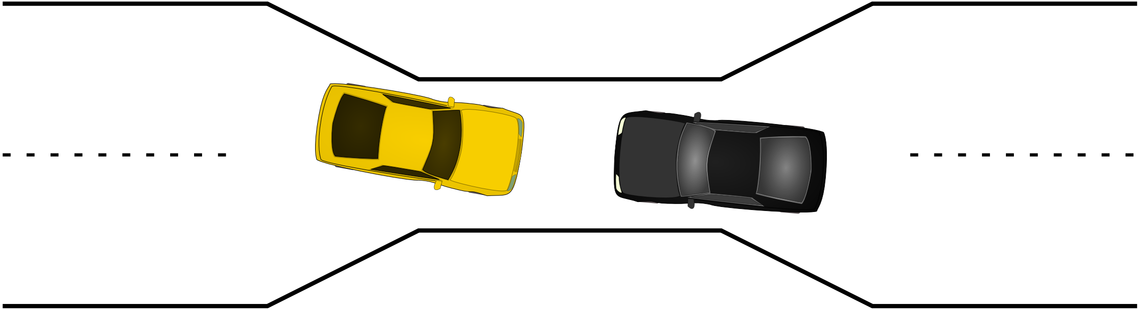
the one-way bridge



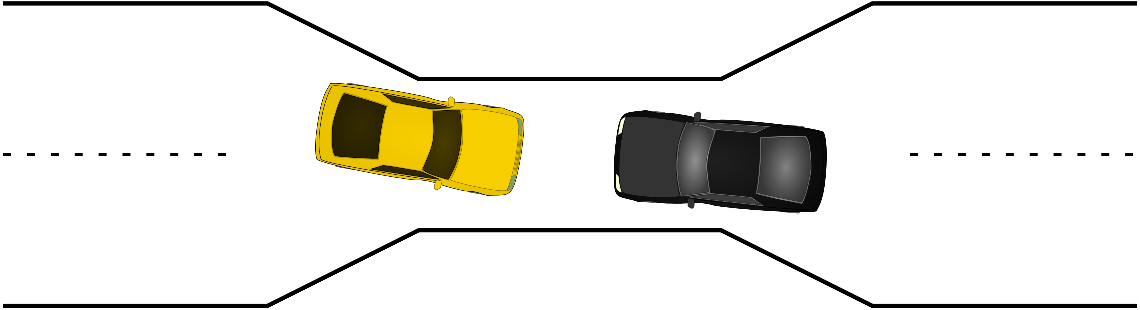
the one-way bridge



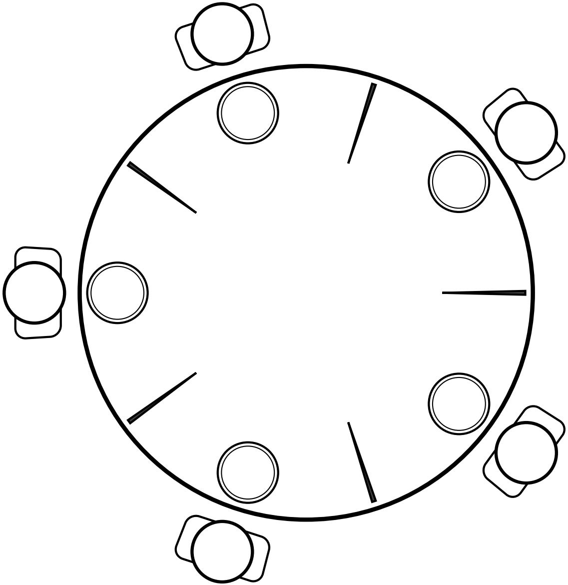
the one-way bridge



the one-way bridge



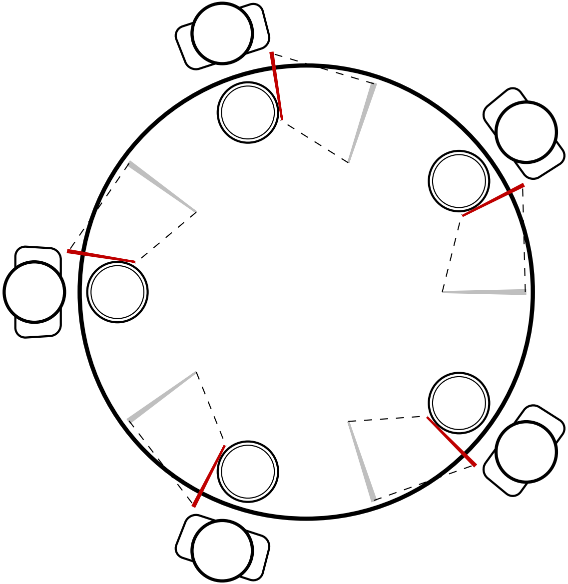
dining philosophers



five philosophers either think or eat

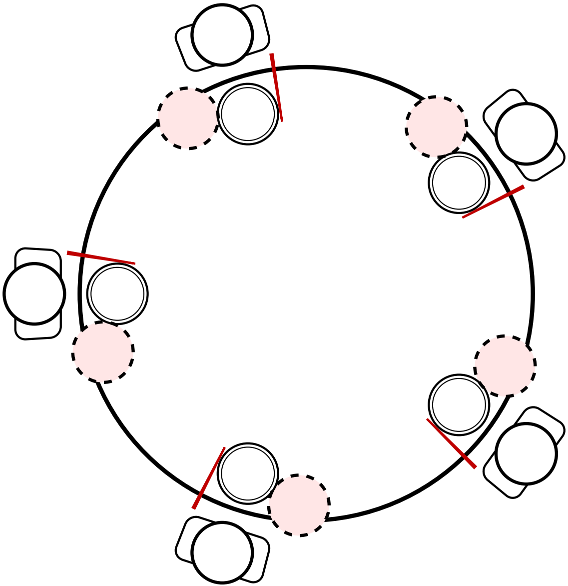
to eat, grab chopsticks on either side

dining philosophers



everyone eats at the same time?
grab left chopstick, then...

dining philosophers



everyone eats at the same time?
grab left chopstick, then
try to grab right chopstick, …
we're at an impasse

pipe() deadlock

BROKEN example:

```
int child_to_parent_pipe[2], parent_to_child_pipe[2];
pipe(child_to_parent_pipe); pipe(parent_to_child_pipe);
if (fork() == 0) {
    /* child */
    write(child_to_parent_pipe[1], buffer, HUGE_SIZE);
    read(parent_to_child_pipe[0], buffer, HUGE_SIZE);
    exit(0);
} else {
    /* parent */
    write(parent_to_child_pipe[1], buffer, HUGE_SIZE);
    read(child_to_parent[0], buffer, HUGE_SIZE);
}
```

This will hang forever (if HUGE_SIZE is big enough).

deadlock waiting

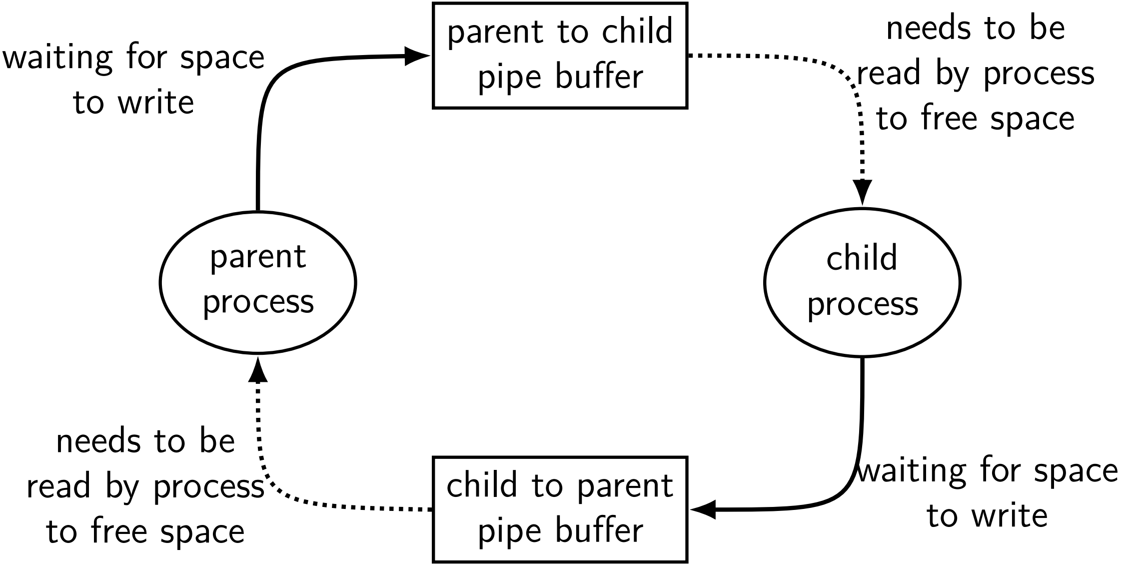
child writing to pipe waiting for free buffer space

...which will not be available until parent reads

parent writing to pipe waiting for free buffer space

...which will not be available until child reads

circular dependency



moving two files

```
struct Dir {
  mutex_t lock; map<string, DirEntry> entries;
};
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {
  mutex_lock(&from_dir->lock);
  mutex_lock(&to_dir->lock);

  to_dir->entries[filename] = from_dir->entries[filename];
  from_dir->entries.erase(filename);

  mutex_unlock(&to_dir->lock);
  mutex_unlock(&from_dir->lock);
}
```

Thread 1: `MoveFile(A, B, "foo")`

Thread 2: `MoveFile(B, A, "bar")`

moving two files: lucky timeline (1)

Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
`lock(&B->lock);`	
(do move)	
`unlock(&B->lock);`	
`unlock(&A->lock);`	
	`lock(&B->lock);`
	`lock(&A->lock);`
	(do move)
	`unlock(&B->lock);`
	`unlock(&A->lock);`

moving two files: lucky timeline (2)

Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
`lock(&B->lock);`	
	`lock(&B->lock…`
(do move)	(waiting for B lock)
`unlock(&B->lock);`	
	`lock(&B->lock);`
	`lock(&A->lock…`
`unlock(&A->lock);`	
	`lock(&A->lock);`
	(do move)
	`unlock(&A->lock);`
	`unlock(&B->lock);`

moving two files: unlucky timeline

Thread 1	Thread 2
`MoveFile(A, B, "foo")`	`MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`

moving two files: unlucky timeline

Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`... stalled	
(waiting for lock on B)	`lock(&A->lock`... stalled
(waiting for lock on B)	(waiting for lock on A)

moving two files: unlucky timeline

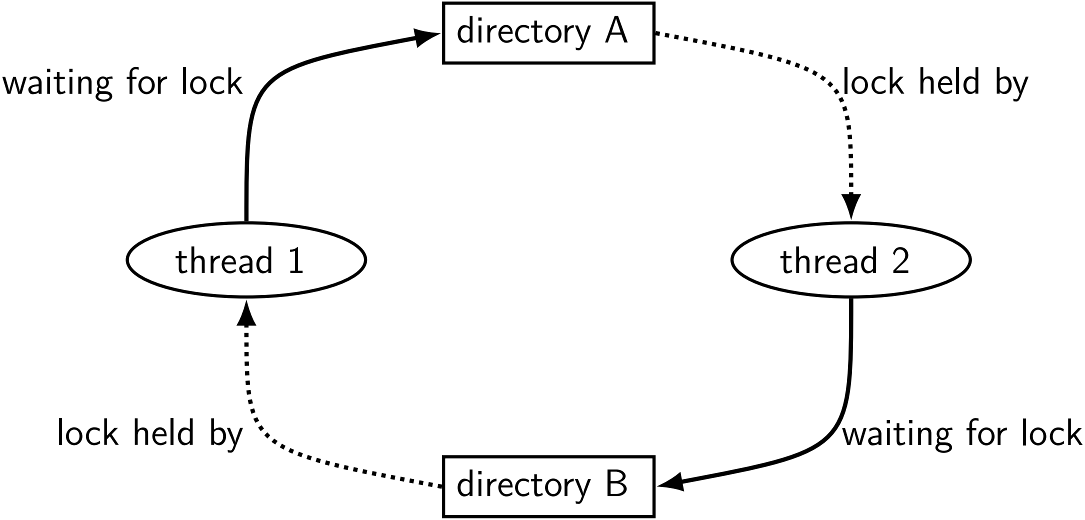
Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock…` stalled	
(waiting for lock on B)	`lock(&A->lock…` stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~ unreachable	~~`unlock(&A->lock);`~~ unreachable
~~`unlock(&A->lock);`~~ unreachable	~~`unlock(&B->lock);`~~ unreachable

moving two files: unlucky timeline

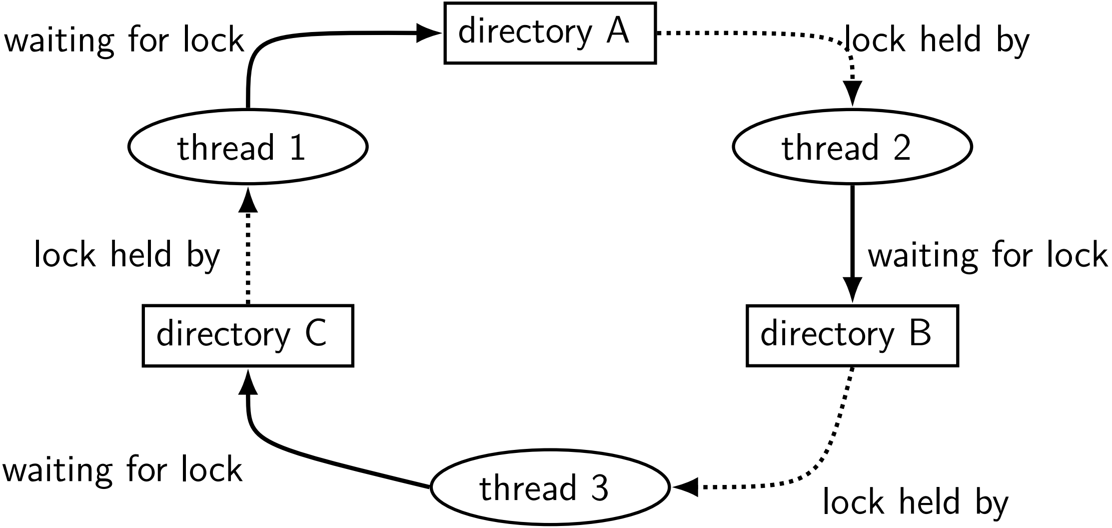
Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock…` stalled	
(waiting for lock on B)	`lock(&A->lock…` stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~ unreachable	~~`unlock(&A->lock);`~~ unreachable
~~`unlock(&A->lock);`~~ unreachable	~~`unlock(&B->lock);`~~ unreachable

Thread 1 holds A lock, waiting for Thread 2 to release B lock
Thread 2 holds B lock, waiting for Thread 1 to release A lock

moving two files: dependencies



moving three files: dependencies



moving three files: unlucky timeline

Thread 1	**Thread 2**	**Thread 3**
`MoveFile(A, B, "foo")`	`MoveFile(B, C, "bar")`	`MoveFile(C, A, "quux")`
`lock(&A->lock);`		
	`lock(&B->lock);`	
		`lock(&C->lock);`
`lock(&B->lock`… stalled		
	`lock(&C->lock`… stalled	
		`lock(&A->lock`… stalled

deadlock with free space

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

2 MB of space — deadlock possible with unlucky order

deadlock with free space (unlucky case)

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	
	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB`… stalled	
	`AllocateOrWaitFor(1 MB`… stalled

deadlock with free space (lucky case)

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	
`AllocateOrWaitFor(1 MB)`	
`(do calculation)`	
`Free(1 MB);`	
`Free(1 MB);`	
	`AllocateOrWaitFor(1 MB)`
	`AllocateOrWaitFor(1 MB)`
	`(do calculation)`
	`Free(1 MB);`
	`Free(1 MB);`

deadlock

deadlock — circular waiting for resources

resource = something needed by a thread to do work

- locks
- CPU time
- disk space
- memory
- ...

often non-deterministic in practice

most common example: when acquiring multiple locks

deadlock

deadlock — circular waiting for resources

resource = something needed by a thread to do work

- locks
- CPU time
- disk space
- memory
- ...

often non-deterministic in practice

most common example: when acquiring multiple locks

deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

- example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

- example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

starvation: once starvation happens, taking turns will resolve

- low priority thread just needed a chance...

deadlock: once it happens, taking turns won't fix

deadlock requirements

mutual exclusion

one thread at a time can use a resource

hold and wait

thread holding a resources waits to acquire *another* resource

no preemption of resources

resources are only released voluntarily

thread trying to acquire resources can't 'steal'

circular wait

there exists a set $\{T_1, \ldots, T_n\}$ of waiting threads such that

T_1 is waiting for a resource held by T_2

T_2 is waiting for a resource held by T_3

…

T_n is waiting for a resource held by T_1

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/*preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/*preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

AllocateOrFail

Thread 1	Thread 2
`AllocateOrFail(1 MB)`	
	`AllocateOrFail(1 MB)`
`AllocateOrFail(1 MB)` fails!	
	`AllocateOrFail(1 MB)` fails!
`Free(1 MB)` (cleanup after failure)	
	`Free(1 MB)` (cleanup after failure)

okay, now what?

- give up?
- both try again? — maybe this will keep happening? (called livelock)
- try one-at-a-time? — gaurenteed to work, but tricky to implement

AllocateOrSteal

Thread 1	Thread 2
`AllocateOrSteal(1 MB)`	
	`AllocateOrSteal(1 MB)`
`AllocateOrSteal(1 MB)`	`Thread killed to free 1MB`
(do work)	

problem: can one actually implement this?

problem: can one kill thread and keep system in consistent state?

fail/steal with locks

pthreads provides pthread_mutex_trylock — “lock or fail”

some databases implement *revocable locks*

- do equivalent of throwing exception in thread to ‘steal’ lock
- need to carefully arrange for operation to be cleaned up

livelock

abort-and-retry

how many times will you retry?

moving two files: abort-and-retry

```
struct Dir {
  mutex_t lock; map<string, DirEntry> entries;
};
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {
  while (mutex_trylock(&from_dir->lock) == LOCKED) {
    if (mutex_trylock(&to_dir->lock) == LOCKED) break;
    mutex_unlock(&from_dir->lock);
  }

  to_dir->entries[filename] = from_dir->entries[filename];
  from_dir->entries.erase(filename);

  mutex_unlock(&to_dir->lock);
  mutex_unlock(&from_dir->lock);
}
```

Thread 1: `MoveFile(A, B, "foo")`

Thread 2: `MoveFile(B, A, "bar")`

moving two files: lots of bad luck?

Thread 1 MoveFile(A, B, "foo")	Thread 2 MoveFile(B, A, "bar")
trylock(&A->lock) → LOCKED	
	trylock(&B->lock) → LOCKED
trylock(&B->lock) → FAILED	
	trylock(&A->lock) → FAILED
unlock(&A->lock)	
	unlock(&B->lock)
trylock(&A->lock) → LOCKED	
	trylock(&B->lock) → LOCKED
trylock(&B->lock) → FAILED	
	trylock(&A->lock) → FAILED
unlock(&A->lock)	
	unlock(&B->lock)

livelock

like deadlock — no one's making progress

potentially forever

unlike deadlock — threads are trying

...but keep aborting and retrying

preventing livelock

make schedule random — e.g. random waiting after abort

make threads run one-at-a-time if lots of aborting

other ideas?

stealing locks???

how do we make stealing locks possible

revokable locks

```
try {
    AcquireLock();
    use shared data
} catch (LockRevokedException le) {
    undo operation hopefully?
} finally {
    ReleaseLock();
}
```

Linux out-of-memory killer

Linux by default *overcommits* memory

- tell processes they have more memory than is available
- (some recommend disabling this feature)

problem: what if wrong?

- could wait for program to finish, free memory...
- but could be waiting forever because of deadlock

solution: kill a process

- (and try to choose one that's not important)

database transactions

databases operations organized into *transactions*

happens all at once or not at all

until transaction is *committed*, not finalized

code to undo transaction in case it's not okay

database deadlock solution: invoke undo transaction code

...then rerun transaction later

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {
  if (from_dir->path < to_dir->path) {
    lock(&from_dir->lock);
    lock(&to_dir->lock);
  } else {
    lock(&to_dir->lock);
    lock(&from_dir->lock);
  }
  ...
}
```

acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {
  if (from_dir->path < to_dir->path) {
    lock(&from_dir->lock);
    lock(&to_dir->lock);
  } else {
    lock(&to_dir->lock);
    lock(&from_dir->lock);
  }
  ...
}
```

any ordering will do
e.g. compare pointers

acquiring locks in consistent order (2)

often by convention, e.g. Linux kernel comments:

```
/*
 * ...
 * Lock order:
 *      contex.ldt_usr_sem
 *        mmap_sem
 *          context.lock
 */
```

```
/*
 * ...
 * Lock order:
 *   1. slab_mutex (Global Mutex)
 *   2. node->list_lock
 *   3. slab_lock(page) (Only on some arches and for debugging)
 * ...
 */
```

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting (e.g. abort and retry) "busy signal"	no *hold and wait*/*preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

allocating all at once?

for resources like disk space, memory

figure out maximum allocation when starting thread

- "only" need conservative estimate

only start thread if those resources are available

okay solution for embedded systems?

deadlock detection

idea: search for cyclic dependencies

detecting deadlocks on locks

let's say I want to detect deadlocks that only involve mutexes

- goal: help programmers debug deadlocks

…by modifying my threading library:

```
struct Thread {
    ... /* stuff for implementing thread */
    /* what extra fields go here? */


};

struct Mutex {
    ... /* stuff for implementing mutex */
    /* what extra fields go here? */


};
```

deadlock detection

idea: search for cyclic dependencies

need:

- list of all contended resources
- what thread is waiting for what?
- what thread 'owns' what?

aside: deadlock detection in reality

instrument all contended resources?

- add tracking of who locked what
- modify every lock implementation — no simple spinlocks?
- some tricky cases: e.g. what about counting semaphores?

doing something useful on deadlock?

- want way to "undo" partially done operations

...but done for some applications

common example: for locks in a database

- database typically has customized locking code
- "undo" exists as side-effect of code for handling power/disk failures

resource allocation graphs

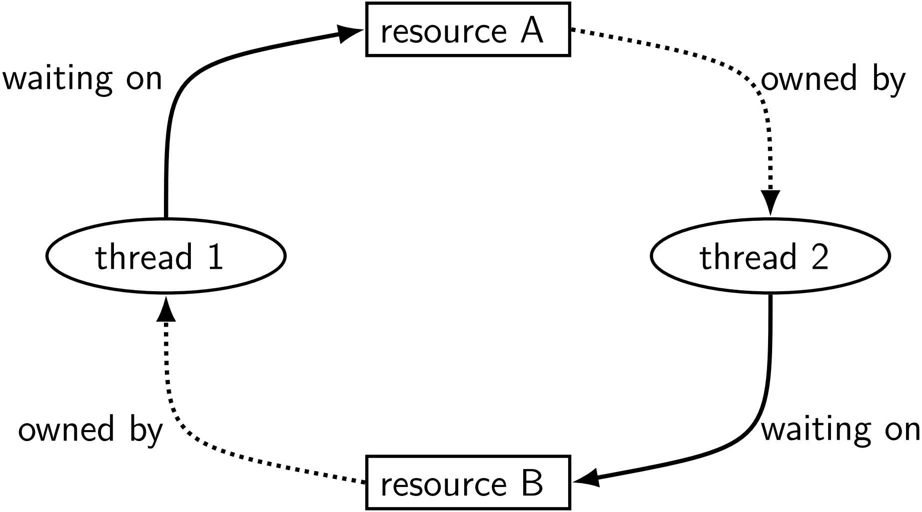
nodes: resources or threads

edge thread→resource: thread waiting for resource

edge resource→thread: resource is “owned” by thread

- holds lock on
- will be deallocated by
- ...

resource allocate graphs



searching for cycles

cycle $\rightarrow$ deadlock happened!

finding cycles: recall 2150 topological sort (maybe???)

divided resources

what about resources like memory?

allocating 1MB of memory:

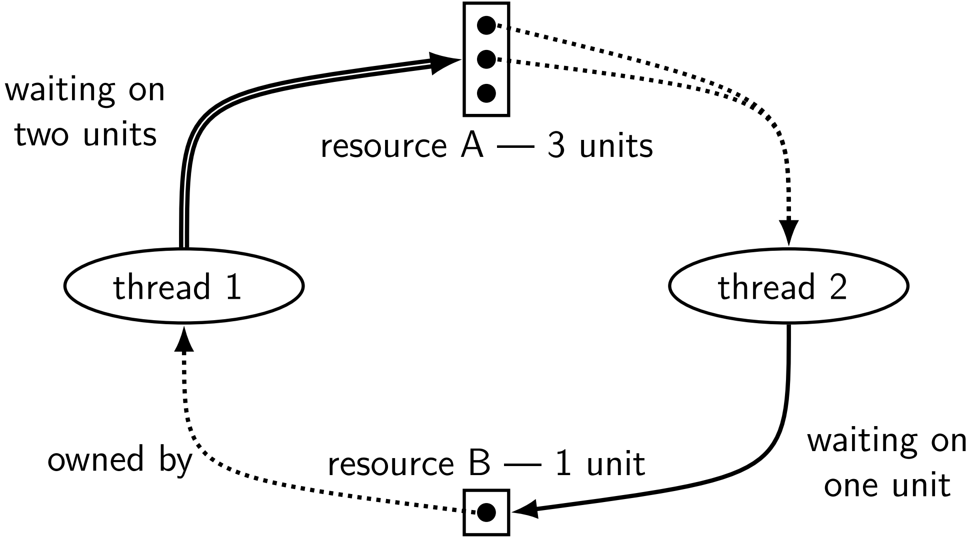
- thread ‘owns’ the 1MB, but...
- another thread can use can use any other 1MB

want to track all of memory together

“partial ownership”

- locked half the memory

dividable/interchangeable resources



deadlock detection

cycle-finding not enough

new idea: try to simulate progress

- anything not waiting releases resources (as it finishes)
- anything waiting on only free resources no one else wants takes resources

see if everything gets resources eventually

deadlock detection (with variable resources)

(pseudocode)

```
class Resources { map<ResourceType, int> amounts; ... };
Resources free_resources;
map<Thread, Resources> requested;
map<Thread, Resources> owned;
```

deadlock detection (with variable resources)

(pseudocode)

```
class Resources { map<ResourceType, int> amounts; ... };
Resources free_resources;
map<Thread, Resources> requested;
map<Thread, Resources> owned;

...
do { done = true;
  for (Thread t : all threads with owned or requested resources) {
    // if everything requested is free, finish
    if (requested[t] <= free_resources ) {
      requested[t] = no_resources;
      free_resources += owned[t];
      owned[t] = no_resources;
      done = false;
    }
  }
} while (!done);
if (owned.size() > 0) { DeadlockDetected() }
```

deadlock detection (with variable resources)

(pseudocode)

```
class Resources { map<ResourceType, int> amounts; ... };
Resources free_resources;
map<Thread, Resources> requested;
map<Thread, Resources> owned;

...
do { done =
  for (Threa                                                    {
    // if everything requested is free, finish
    if (requested[t] <= free_resources ) {
      requested[t] = no_resources;
      free_resources += owned[t];
      owned[t] = no_resources;
      done = false;
    }
  }
} while (!done);
if (owned.size() > 0) { DeadlockDetected() }
```

$\leq$ — free resources include *everything* being requested
(enough memory, disk, each lock requested, etc.)
note: not requesting anything right now? — always true

deadlock detection (with variable resources)

(pseudocode)

```
class Resources { map<ResourceType, int> amounts; ... };
Resources free_resources;
map<Thread, Resources> requested;
map<Thread, Resources> owned;

...
do { done = true;
  for (Thread t : all                                resources) {
    // if everything requested is free, finish
    if (requested[t] <= free_resources ) {
      requested[t] = no_resources;
      free_resources += owned[t];
      owned[t] = no_resources;
      done = false;
    }
  }
} while (!done);
if (owned.size() > 0) { DeadlockDetected() }
```

assume requested resources taken
then everything taken released

deadlock detection (with variable resources)

(pseudocode)

```
class Resources { map<ResourceType, int> amounts; ... };
Resources free_resources;
map<Thread, Resources> requested;
map<Thread, Resources> owned;

...
do { done = true;
  for (Thread t : all threads with owned or requested resources) {
    // if everything requested is free, finish
    if (requested[t] <= free_resources ) {
      requested[t] = no_resources;
      owned[t] = no_resources;
      done = false;
    }
  }
} while (!done);
if (owned.size() > 0) { DeadlockDetected() }
```

keep going until nothing changes

using deadlock detection for prevention

suppose you know the *maximum resources* a process could request

make decision when starting process ("*admission control*")

using deadlock detection for prevention

suppose you know the *maximum resources* a process could request

make decision when starting process ("*admission control*")

ask "what if every process was waiting for maximum resources"

including the one we're starting

would it cause deadlock? then don't let it start

called Baker's algorithm

recovering from deadlock?

what if it's too late?

kill a thread involved in the deadlock?
- hopefully won't mess things up???

tell owner to release a resource
- need code written to do this???

same concept as locks you can steal

additional threading topics (if time)

queuing spinlocks: ticket spinlocks?

Linux kernel support for user locks: futexes?

fast synchronization for read-mostly data: read-copy-update?

threads are hard

get synchronization wrong? weird things happen

...and only sometimes

are there better ways to handle the same problems?

- concurrency — multiple things at once
- parallelism — same thing, use more cores/etc.

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

issue: read from/write to multiple streams at once?

event loops

```
while (true) {
    event = WaitForNextEvent();
    switch (event.type) {
    case NEW_CONNECTION:
        handleNewConnection(event); break;
    case CAN_READ_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleRead(connection);
        break;
    case CAN_WRITE_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleWrite(connection);
        break;
        ...
    }
}
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + com
                     sizeof(command) –
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length – 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + con
                    sizeof(command) -
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length - 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
    int fd;
    char command[1024];
    size_t command_length;
    char response[1024];
    size_t total_written;
    ...
};
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

POSIX support for event loops

`select` and `poll` functions

- take list(s) of file descriptors to read and to write
- wait for them to be read/writeable without waiting
- (or for new connections associated with them, etc.)

many OS-specific extensions/improvements/alternatives:

- examples: Linux epoll, Windows IO completion ports
- better ways of managing list of file descriptors
- do read/write when ready instead of just returning when reading/writing is okay

message passing

instead of having variables, locks between threads...

send messages between threads/processes

what you need anyways between machines

- big 'supercomputers' = really many machines together

arguably an easier model to program

- can't have locking issues

message passing API

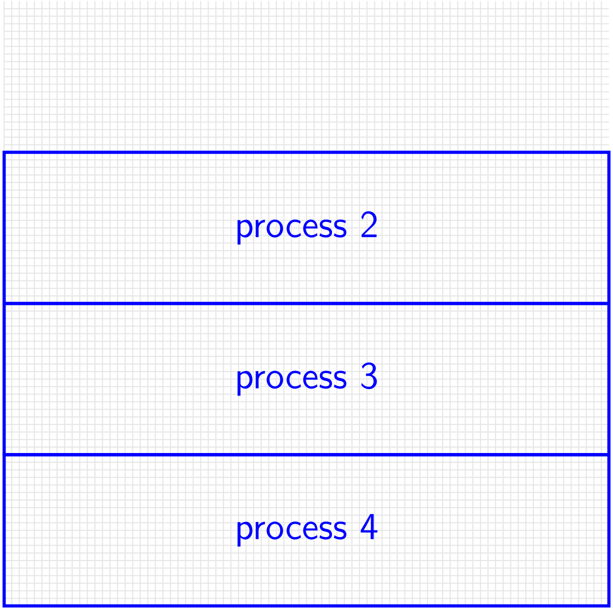
core functions: Send(toId, data)/Recv(fromId, data)

simplest version: functions wait for other processes/threads

- extensions: send/recv at same time, multiple messages at once, don't wait, etc.

```
if (thread_id == 0) {
    for (int i = 1; i < MAX_THREAD; ++i) {
        Send(i, getWorkForThread(i));
    }
    for (int i = 1; i < MAX_THREAD; ++i) {
        WorkResult result;
        Recv(i, &result);
        handleResultForThread(i, result);
    }
} else {
    WorkInfo work;
    Recv(0, &work);
    Send(0, ComputeResultFor(work));
```

message passing game of life

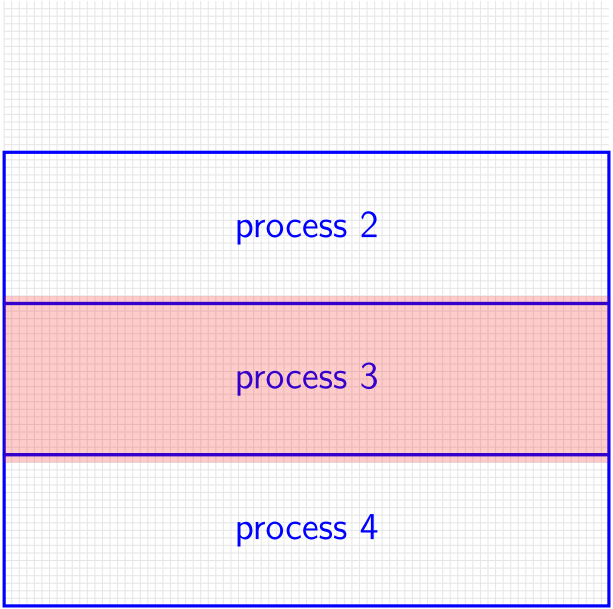


divide grid
like you would for normal threads

each process stores cells
in that part of grid

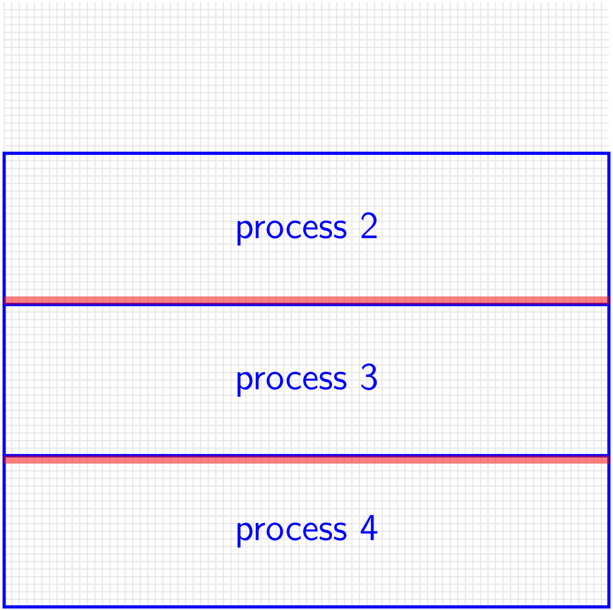
(no shared memory!)

message passing game of life



process 3 only needs values of cells around its area (values of cells adjacent to the ones it computes)

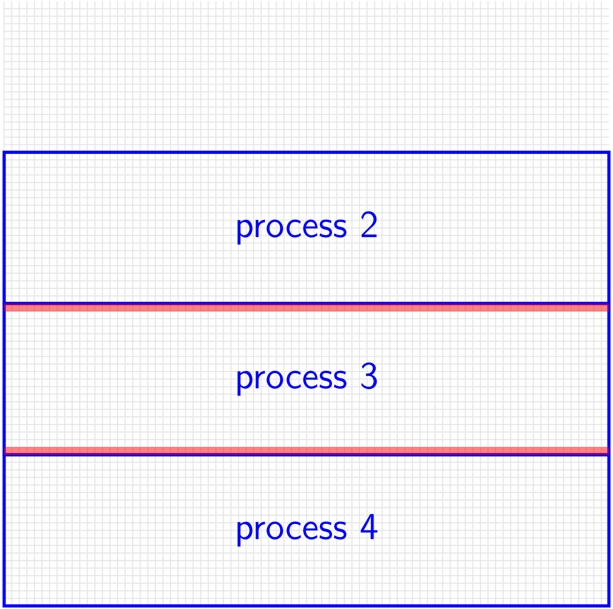
message passing game of life



small slivers of
other process's cells needed

solution: process 2, 4
send messages with cells every iterat

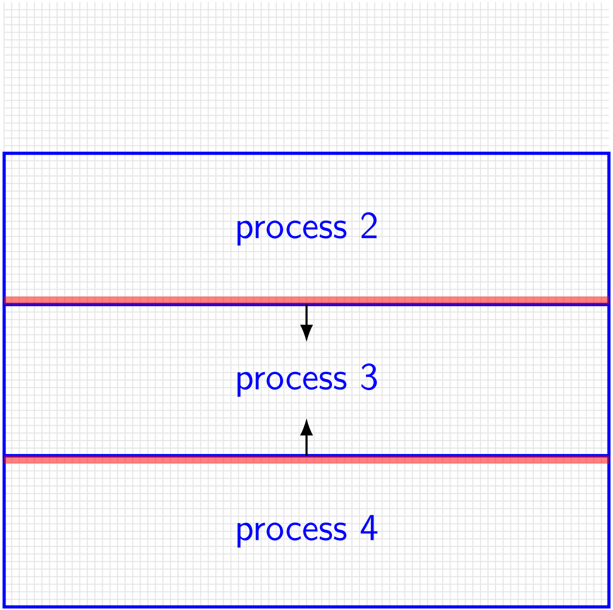
message passing game of life



some of process 3's cells
also needed by process 2/4

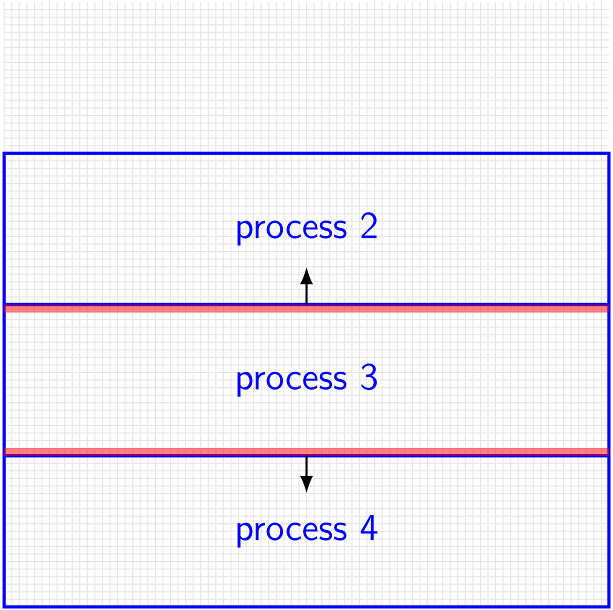
so process 3 also sends messages

message passing game of life



one possible pseudocode:
all even processes send messages
(while odd receives), then
all odd processes send messages
(while even receives)

message passing game of life



one possible pseudocode:

all even processes send messages (while odd receives), then

all odd processes send messages (while even receives)

backup slides

fairer spinlocks

so far — everything on spinlocks

- mutexes, condition variables — built with spinlocks

spinlocks are pretty ‘unfair’

- where fair = get lock if waiting longest

last CPU that held spinlock more likely to get it again

- already has the lock in its cache...

but there are many other ways to spinlocks...

ticket spinlocks

```
unsigned int serving_number;
unsigned int next_number;

Lock() {
    // "take a number"
    unsigned int my_number = atomic_read_and_increme
    // wait until "now serving" that number
    while (atomic_read(&serving_number) != my_number
        /* do nothing */
    }
    // MISSING: code to prevent reordering reads/wri
}

Unlock() {
```

ticket spinlocks and cache contention

still have contention to write next_number

...but no retrying writes!

- should limit 'ping-ponging'?

threads loop performing a read repeatedly while waiting

- value will be broadcasted to all processors
- 'free' if using a bus
- not-so-free if another way of connecting CPUs

beyond ticket spinlocks

Linux kernel used to use ticket spinlocks

now uses variant of MCS spinlocks — locks have linked-list queue!

- careful use of atomic operations to modify queue

still try

goal: even less contention

- unlocking value doesn't require broadcasting to all CPUs
- each processor waits on its own cache block

Linux futexes

`futex` — **f**ast **u**serspace mu**tex**

goal: implement waiting like 'proper' mutexes, but...

don't enter kernel mode most of the time

challenge: can't acquire lock to call scheduler from user mode

futex operations

```
futex(&lock_value, FUTEX_WAIT, expected_value, ...);
```

check if `lock_value` is `expected_value`

- if not — return immediately
- otherwise, sleep until it `futex(…, FUTEX_WAKE` is called

```
futex(&lock_value, FUTEX_WAKE, num_processes);
```

wakeup up to `num_processes` which called FUTEX_WAIT

mutexes with futexes

```
int lock_value; // UNLOCKED or LOCKED_NO_WAITERS or LOCKED_WAITERS
Lock() {
retry:
    if (CompareAndSwap(&lock_value, UNLOCKED, LOCKED_NO_WAITERS) ==
        /* acquired lock */
        return;
    } else if (CompareAndSwap(&lock_value, LOCKED_NO_WAITERS, LOCKED
        futex(&lock_value, FUTEX_WAIT, LOCKED_WAITERS, ...);
    }
    goto retry;
}
Unlock() {
    if (CompareAndSwap(&lock_value, LOCKED_NO_WAITERS, UNLOCKED) ==
        return;
    } else {
        lock_value = UNLOCKED;
        futex(&lock_value, FUTEX_WAKE, 1, ...);
    }
}
```

implementing futex_wait

hashtable: address $\rightarrow$ queue of waiting threads

use hashtable to look-up queue

lock queue

check value hasn't changed

- if so abort, releasing lock

add thread to queue

set thread as WAITING (not runnable)

unlock queue

call scheduler

read-copy-update (high-level overview)

idea: read-mostly data structure

when reading:
- read normally via shared pointer

when writing:
- make a copy
- atomically update the shared pointer
- delete the old version eventually

tricky part: when is it safe to delete old version
- implementation: scheduler integration

RCU operations

read lock — record: "I am reading now"

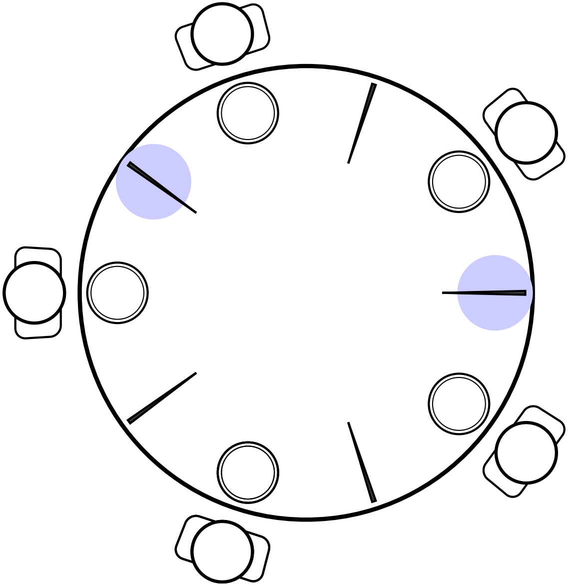
read unlock — record: "I am done reading now"

publish — atomically update pointer

after publish: wait until

all threads currently running have context switched
...and none of them set the "I am reading now" bit

dining philosophers — ordering

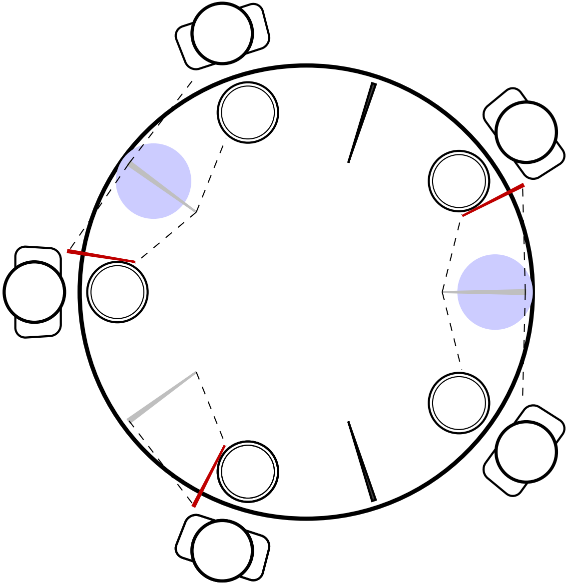


mark some chopsticks places

rule: grab from marked place first

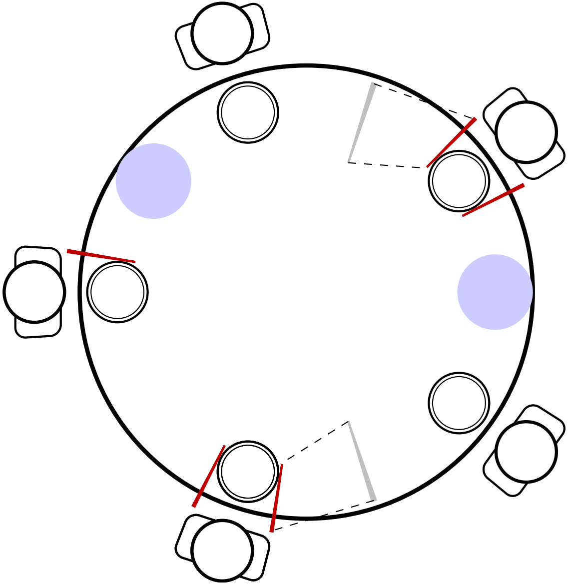
only grab other chopstick after that

dining philosophers — ordering



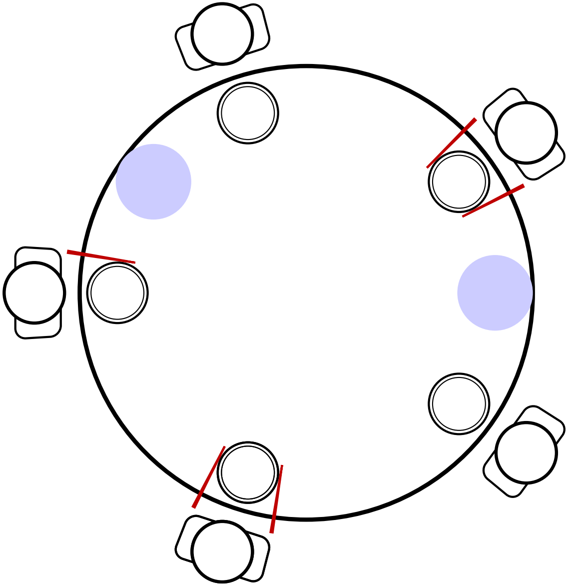
mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

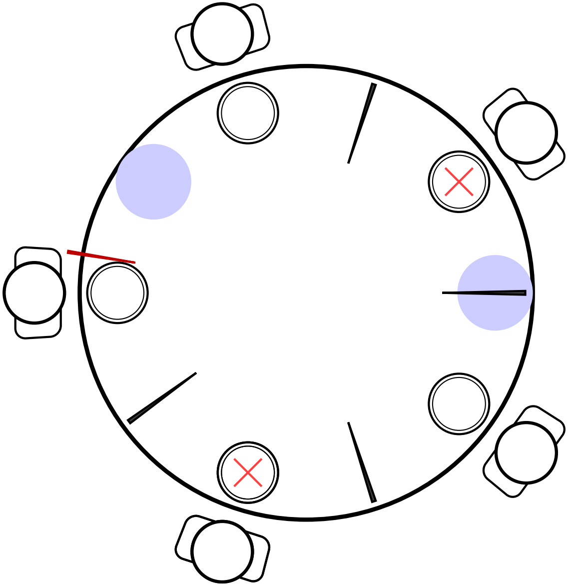
dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

avoids circular dependency,
means everyone else
eventually gets a turn

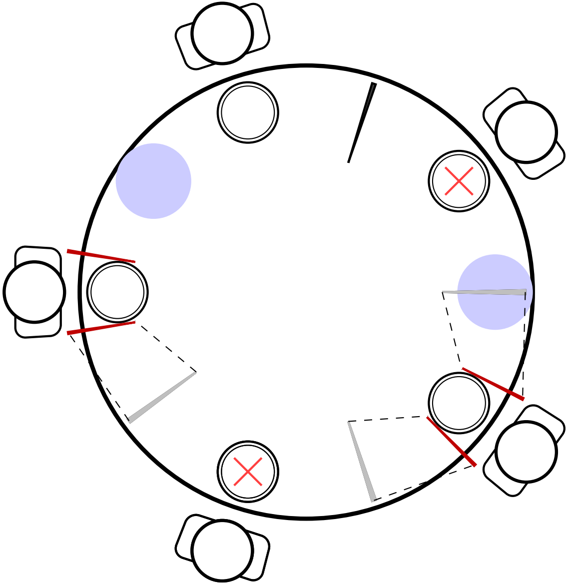
dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

avoids circular dependency,
means everyone else
eventually gets a turn

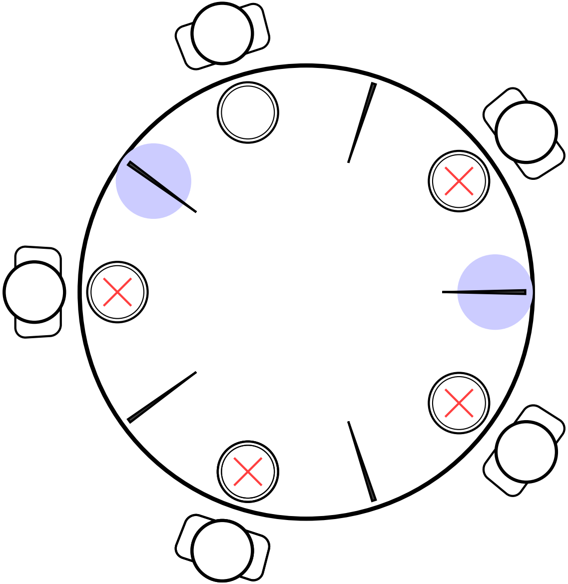
dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

avoids circular dependency,
means everyone else
eventually gets a turn

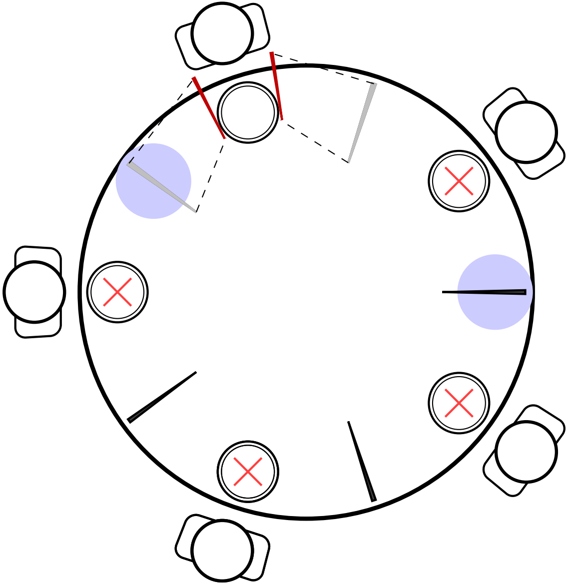
dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

avoids circular dependency,
means everyone else
eventually gets a turn

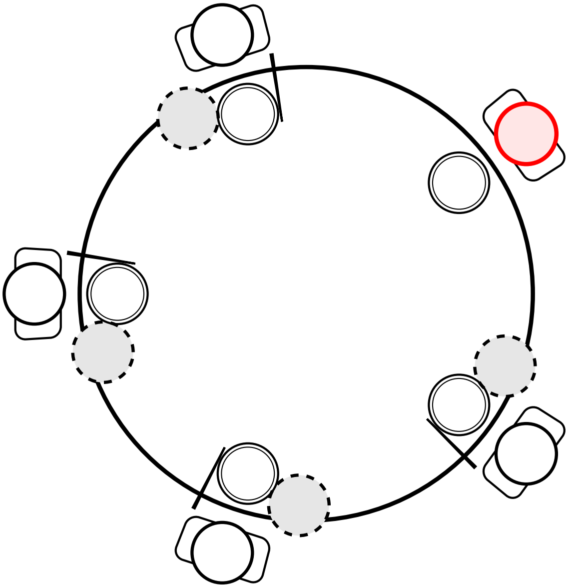
dining philosophers — ordering



mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

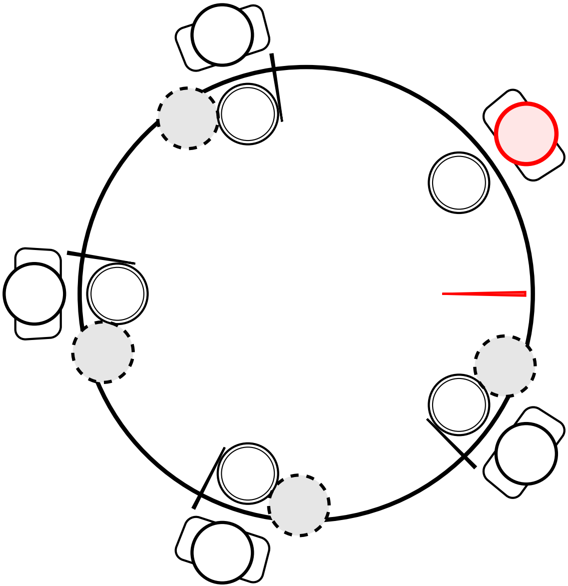
avoids circular dependency,
means everyone else
eventually gets a turn

dining philosophers — aborting



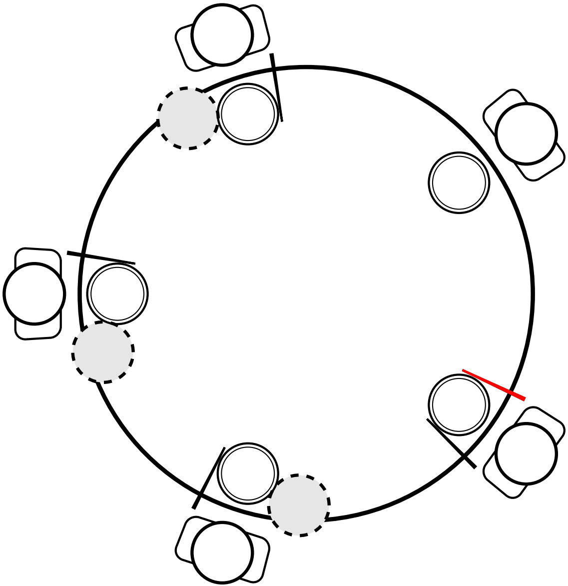
dining philosopher
what if someone's impatient
just gives up instead of waiting

dining philosophers — aborting



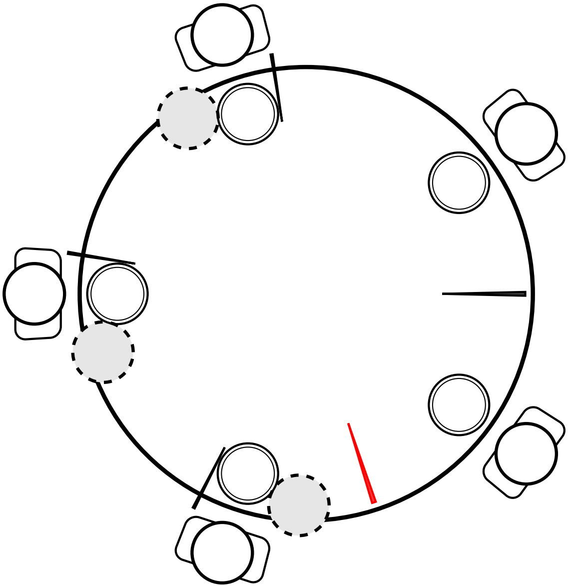
dining philosopher
what if someone's impatient
just gives up instead of waiting

dining philosophers — aborting



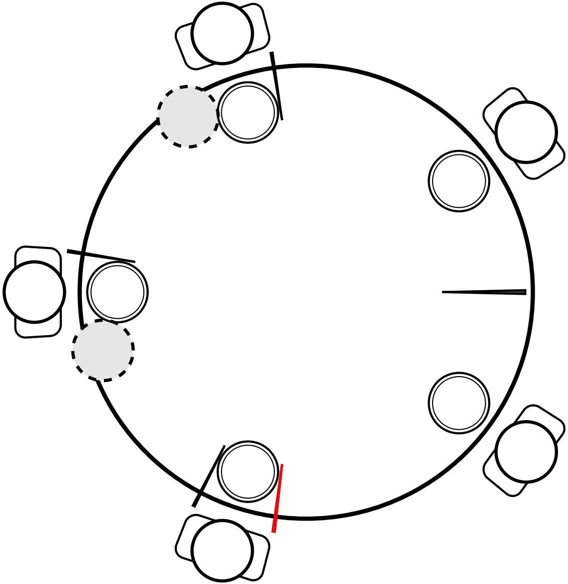
now everyone else can eat

dining philosophers — aborting



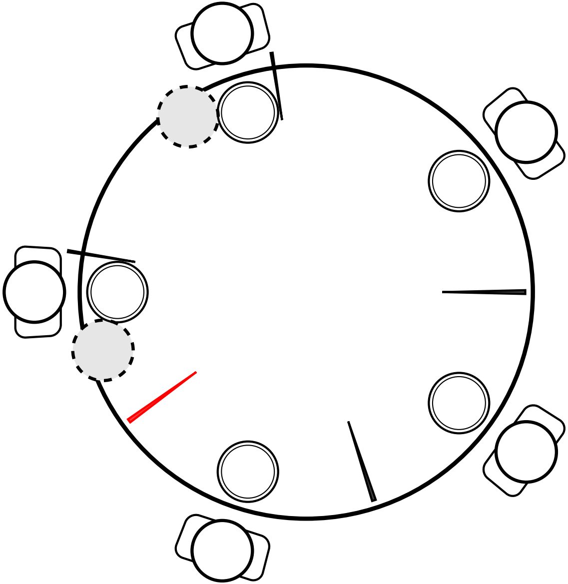
now everyone else can eat

dining philosophers — aborting



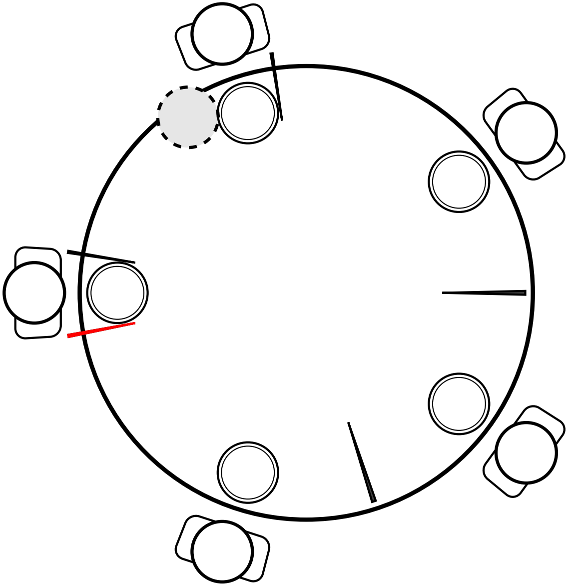
now everyone else can eat

dining philosophers — aborting



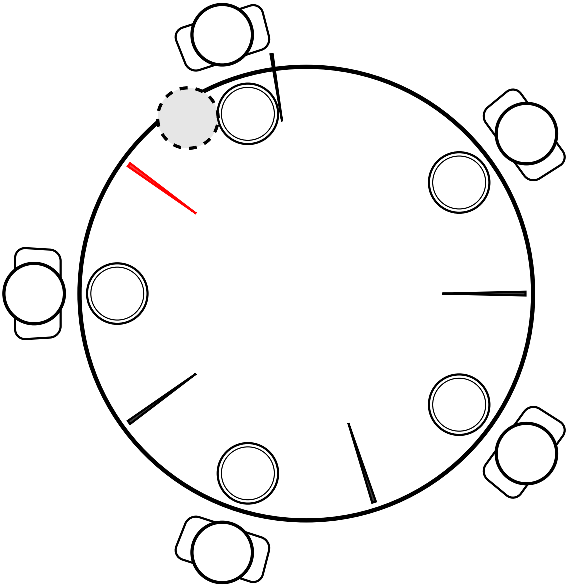
now everyone else can eat

dining philosophers — aborting



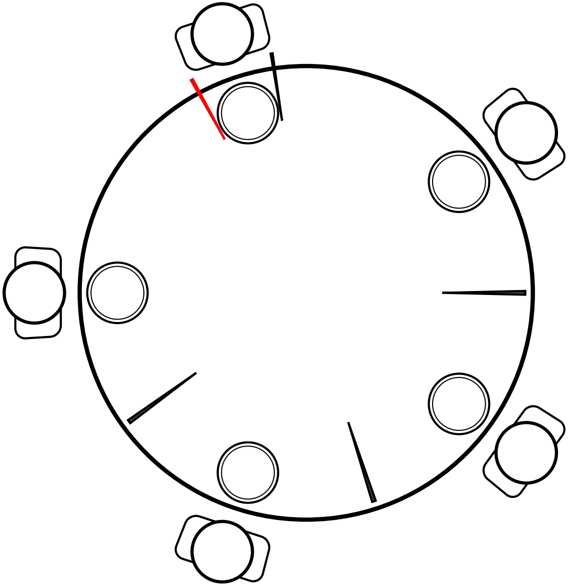
now everyone else can eat

dining philosophers — aborting



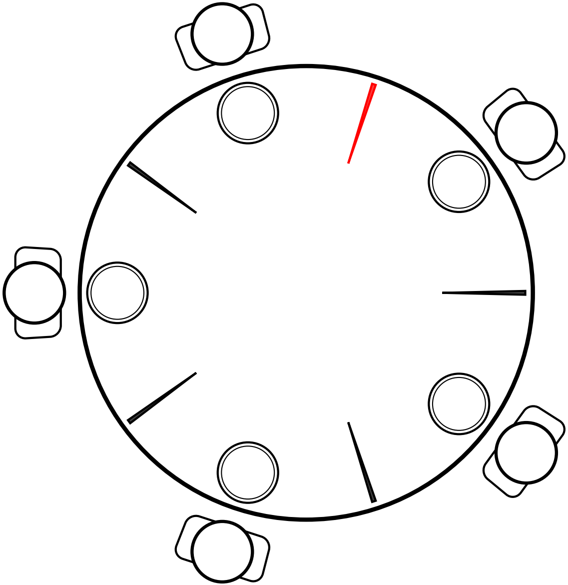
now everyone else can eat

dining philosophers — aborting



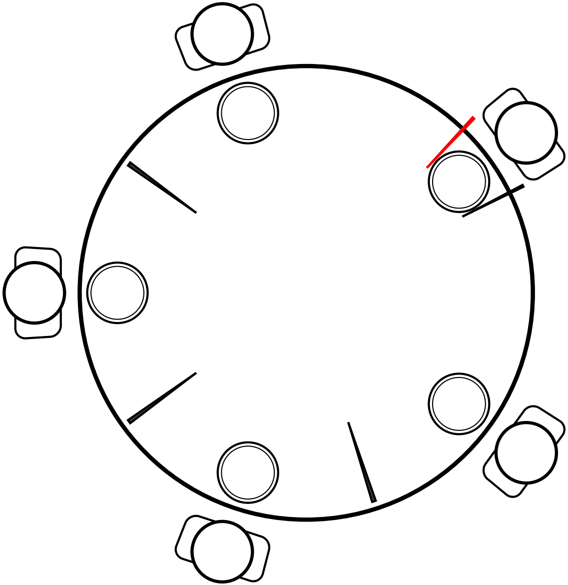
now everyone else can eat

dining philosophers — aborting



now everyone else can eat

dining philosophers — aborting



and person who gave up
might succeed later