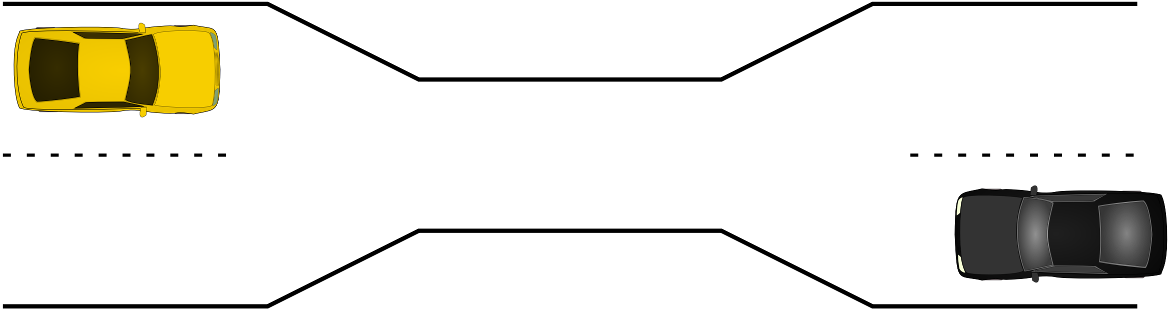
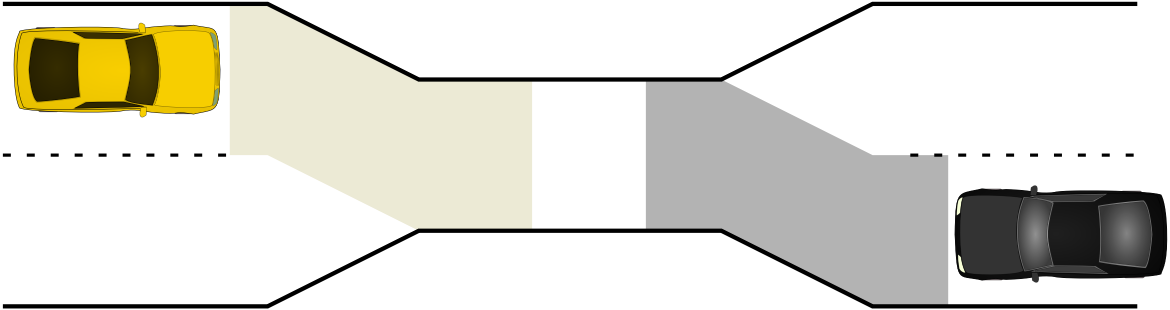
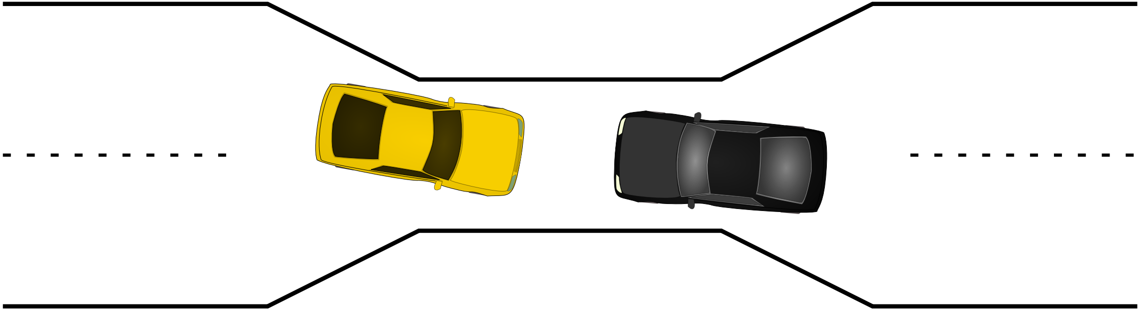
the one-way bridge



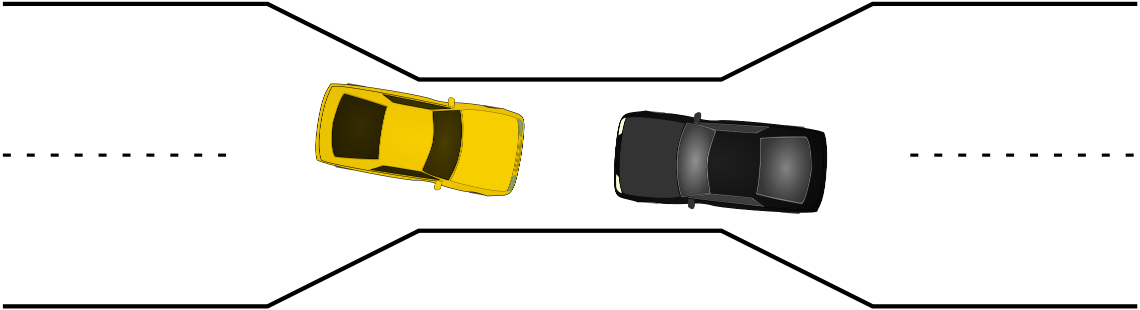
the one-way bridge



the one-way bridge



the one-way bridge



moving two files

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

  Map_put(to_dir->entries, filename,
        Map_get(from_dir->entries, filename));
  Map_erase(from_dir->entries, 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	Thread 2
`MoveFile(A, B, "foo")`	`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);`

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	Thread 2
`MoveFile(A, B, "foo")`	`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

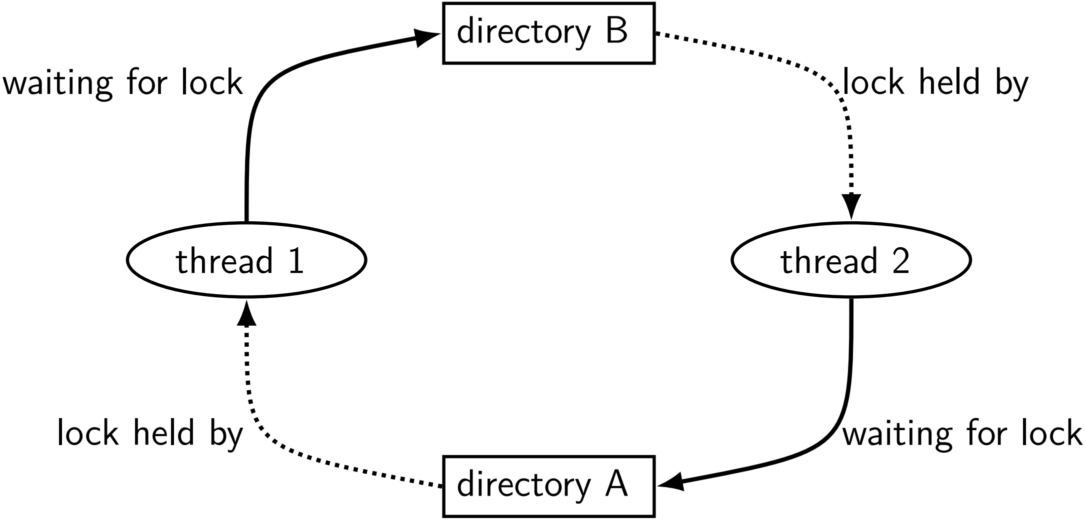
Thread 1	Thread 2
`MoveFile(A, B, "foo")`	`MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock…` stalled	
(waiting for lock on B)	`lock(&A->lock…` stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~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

moving two files: unlucky timeline

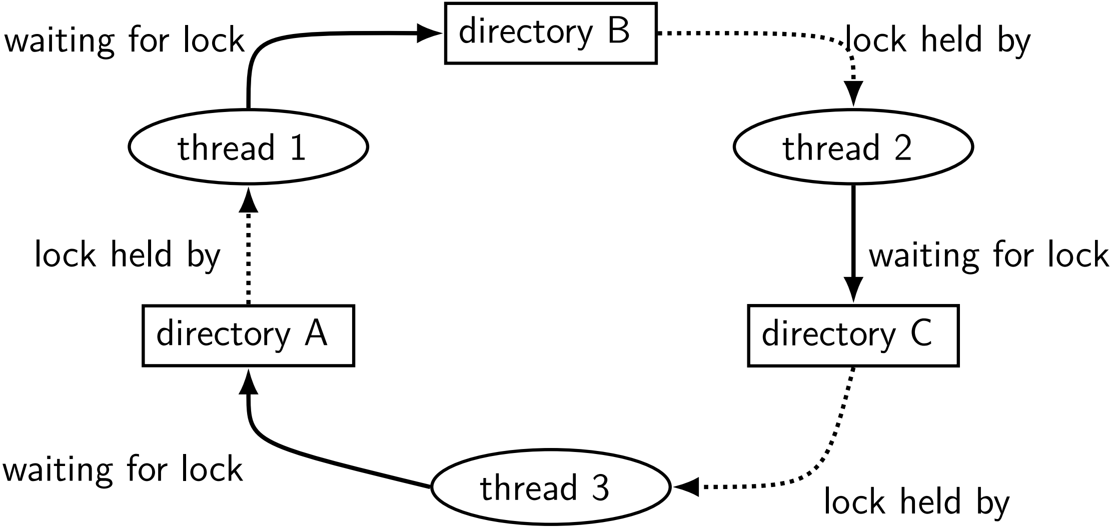
Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`… stalled	
(waiting for lock on B)	`lock(&A->lock`… stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~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

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

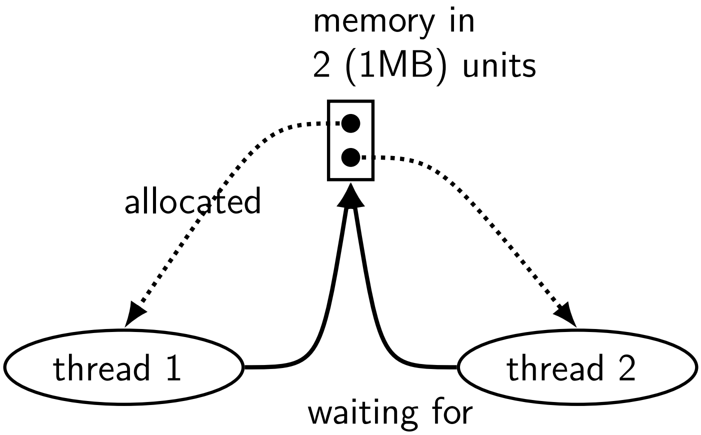
Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

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

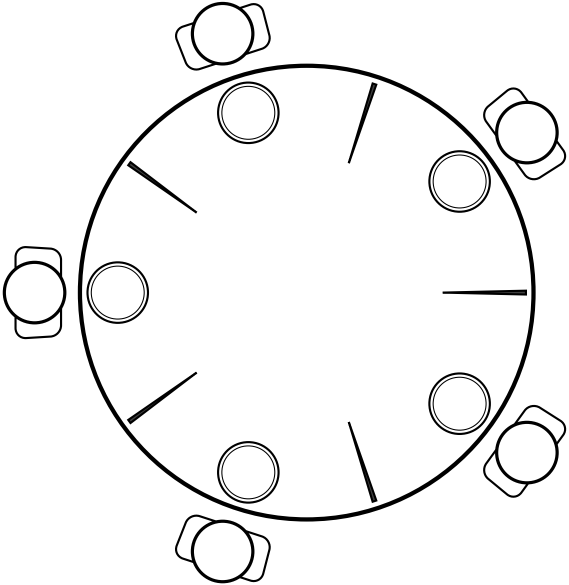
free space: dependency graph



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);`

dining philosophers



five philosophers either think or eat

to eat:

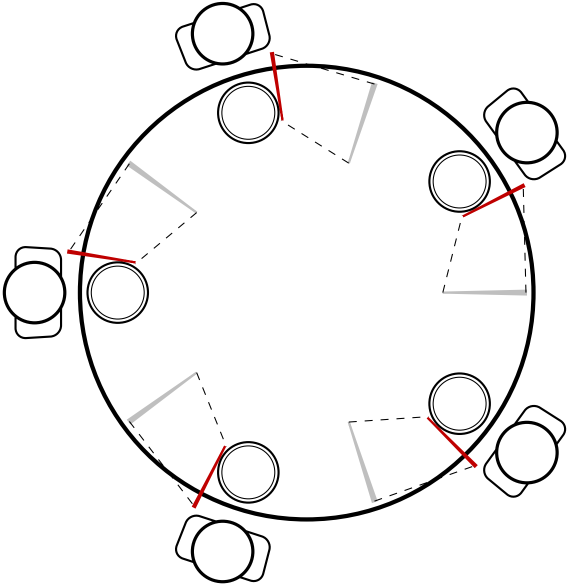
grab chopstick on left, then

grba chopstick on right, then

then eat, then

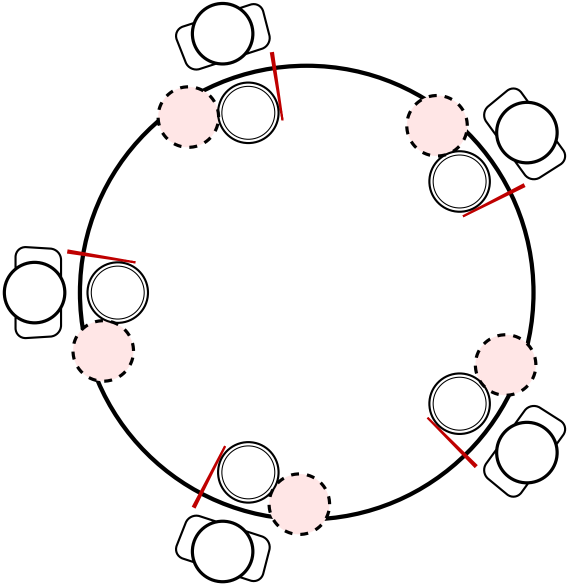
return chopsticks

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

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

how is deadlock possible?

Given list: A, B, C, D, E

```
RemoveNode(LinkedListNode *node) {
    pthread_mutex_lock(&node->lock);
    pthread_mutex_lock(&node->prev->lock);
    pthread_mutex_lock(&node->next->lock);
    node->next->prev = node->prev; node->prev->next = node->next;
    pthread_mutex_unlock(&node->next->lock); pthread_mutex_unlock(&node->p
    pthread_mutex_unlock(&node->lock);
}
```

Which of these (all run in parallel) can deadlock?

A. RemoveNode(B) and RemoveNode(C)
B. RemoveNode(B) and RemoveNode(D)
C. RemoveNode(B) and RemoveNode(C) and RemoveNode(D)
D. A and C
E. B and C
F. all of the above
G. none of the above

how is deadlock — solution

Remove B	Remove C
lock B	lock C
lock A (prev)	wait to lock B (prev)
wait to lock C (next)	

With B and D — only overlap in in node C — no circular wait possible

deadlock prevention techniques

infinite resources

or at least enough that never run out

no *mutual exclusion*

no shared resources

no *mutual exclusion*

no waiting

"busy signal" — abort and (maybe) retry

revoke/preempt resources

no *hold and wait*/ *preemption*

acquire resources in **consistent order**

no *circular 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

“busy signal” — abort and (maybe) retry

revoke/preempt resources

no *hold and wait*/ *preemption*

acquire resources in **consistent order**

no *circular wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/ *preemption*

“busy signal” — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

memory allocation: malloc() fails rather than waiting (no deadlock)
locks: `pthread_mutex_trylock` fails rather than waiting
...

no waiting — no *hold and wait*/ *preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

stealing locks???

how do we make stealing locks possible

unclean: just kill the thread

- problem: inconsistent state?

clean: have code to undo partial oepration

- some databases do this

won't go into detail in this class

revokable locks?

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

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/*preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

abort and retry limits?

abort-and-retry

pthread's mutexes:

```
pthread_mutex_trylock
pthread_mutex_timedlock
```

how many times will you retry?

moving two files: abort-and-retry

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

  Map_put(to_dir->entries, filename, Map_get(from_dir->entries, fil
  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")`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	
	`unlock(&B->lock)`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	

livelock

livelock: keep aborting and retrying without end

like deadlock — no one's making progress

potentially forever

unlike deadlock — threads are not waiting

preventing livelock

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

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

other ideas?

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

requires some way to undo partial changes to avoid errors

common approach for databases

...

no waiti

"busy signal" — abort and (maybe) retry

revoke/preempt resources

no *hold and wait*/ *preemption*

acquire resources in **consistent order** — no *circular 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 "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular 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

"busy signal" — abort and (maybe) retry

revoke/preempt resources

no *hold and wait*/ *preemption*

acquire resources in **consistent order**

no *circular wait*

backup slides

deadlock detection

why? debugging or fix deadlock by aborting operations

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

why? debugging or fix deadlock by aborting operations

idea: search for cyclic dependencies

need:

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

aside: divisible resources

deadlock is possible with divislbe resources like memory,...

example: suppose 6MB of RAM for threads total:
- thread 1 has 2MB allocated, waiting for 2MB
- thread 2 has 2MB allocated, waiting for 2MB
- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

not a deadlock — thread 3 can still finish
- and after it does, thread 1 or 2 can finish

aside: divisible resources

deadlock is possible with divislbe resources like memory,…

example: suppose 6MB of RAM for threads total:

- thread 1 has 2MB allocated, waiting for 2MB
- thread 2 has 2MB allocated, waiting for 2MB
- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

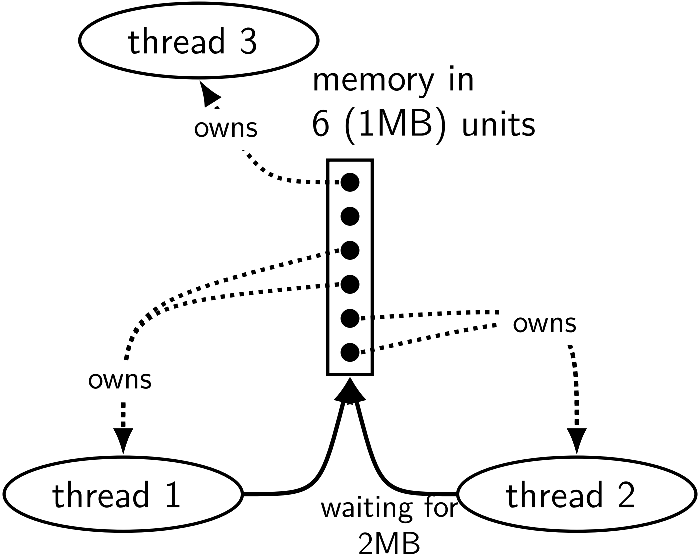
not a deadlock — thread 3 can still finish

- and after it does, thread 1 or 2 can finish

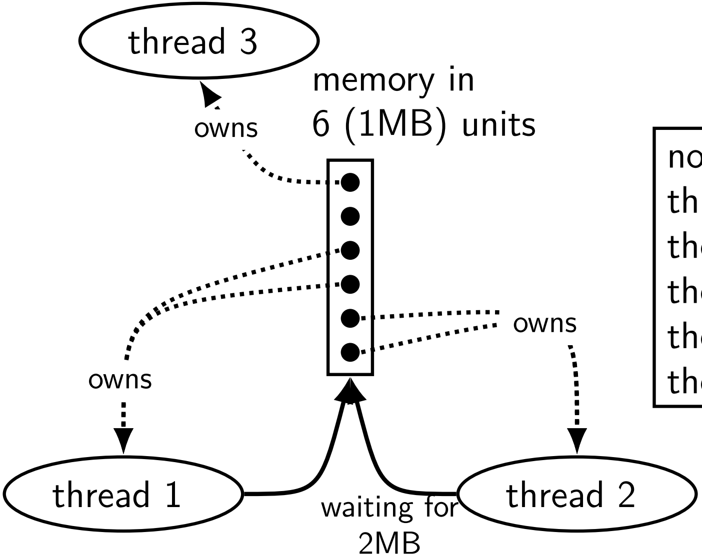
…but would be deadlock

- …if thread 3 waiting lock held by thread 1
- …with 5MB of RAM

divisible resources: not deadlock

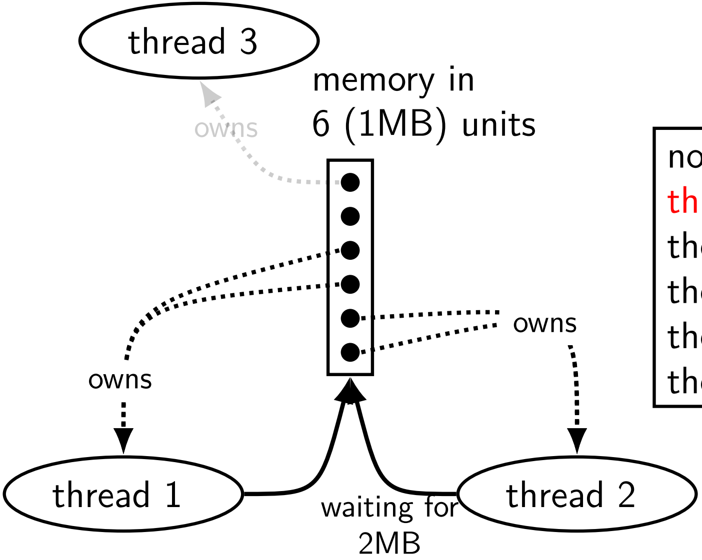


divisible resources: not deadlock



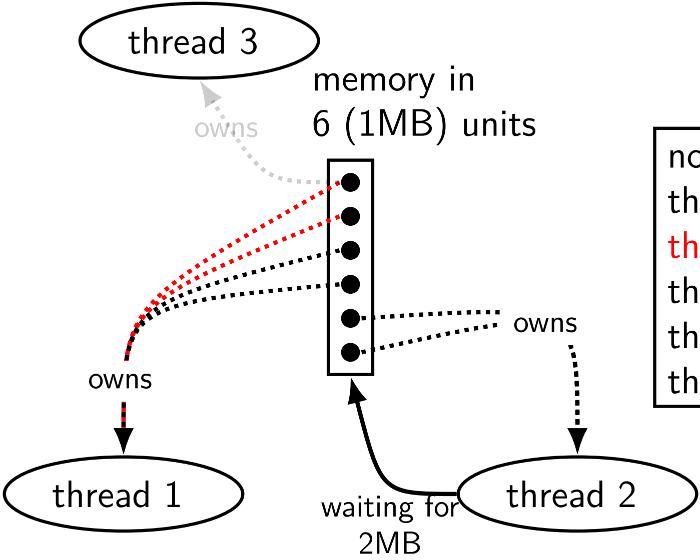
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



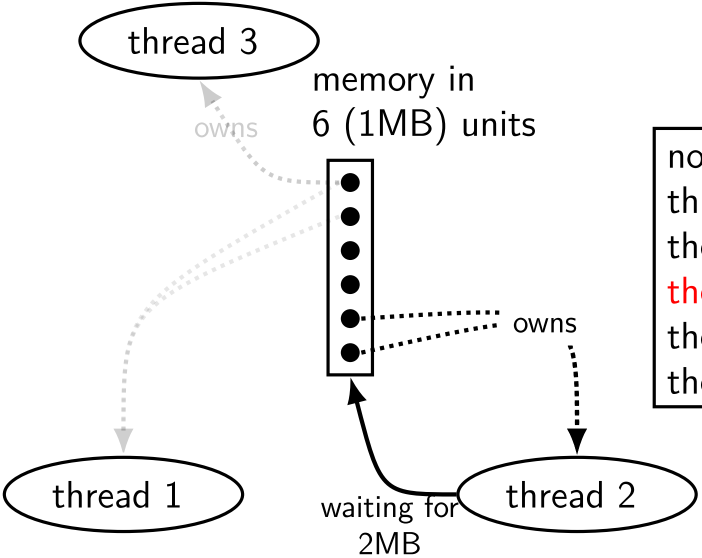
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



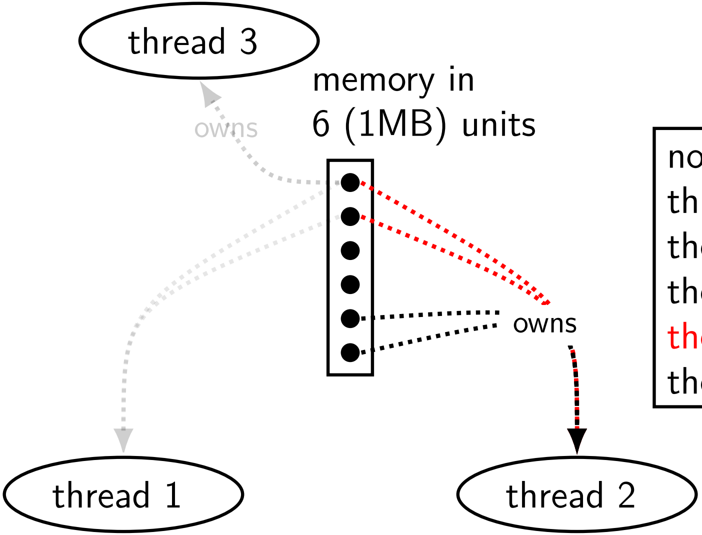
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



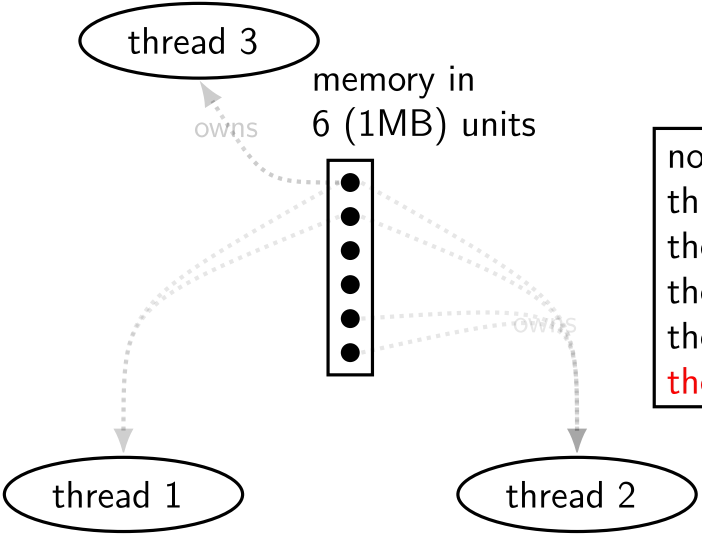
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



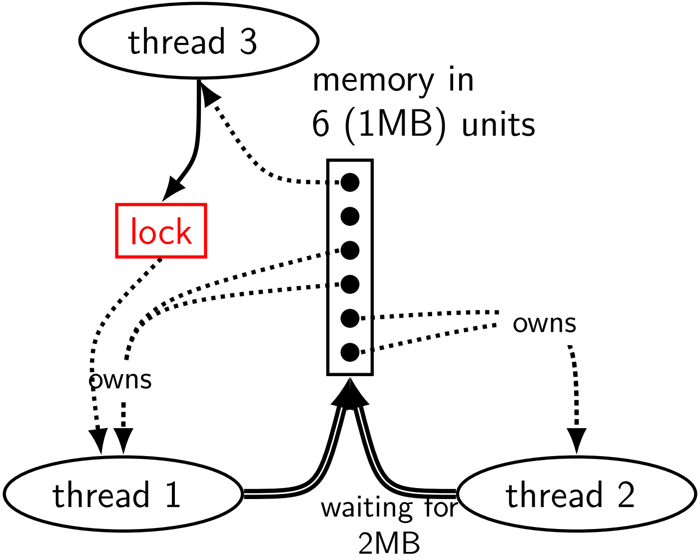
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock

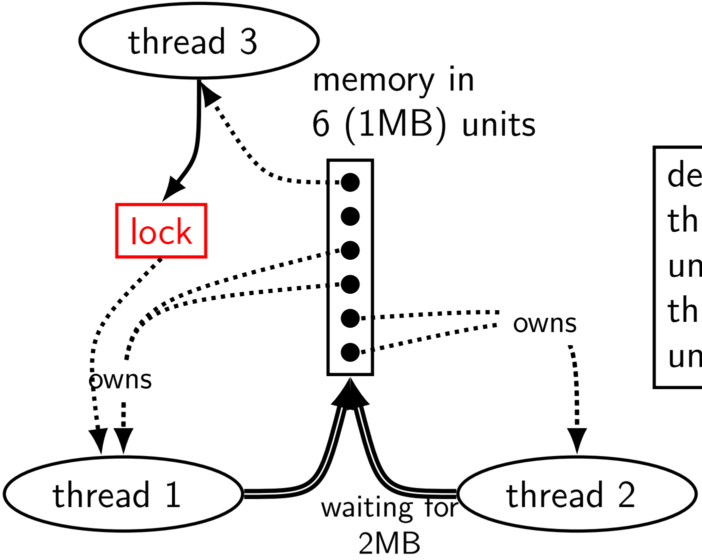


not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: is deadlock

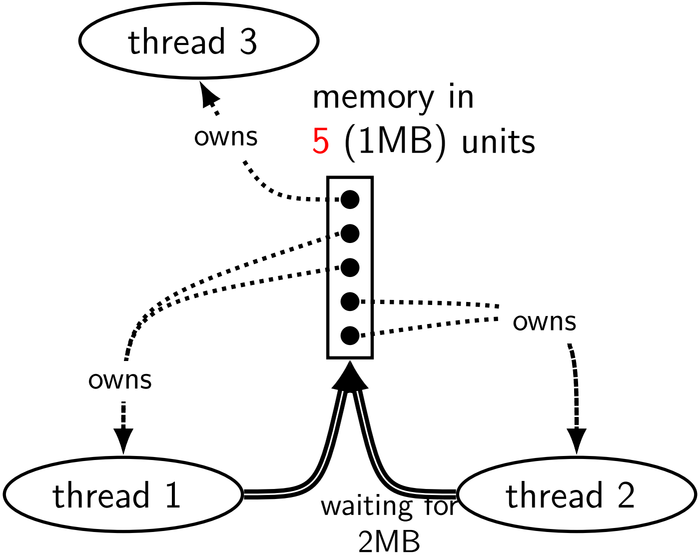


divisible resources: is deadlock

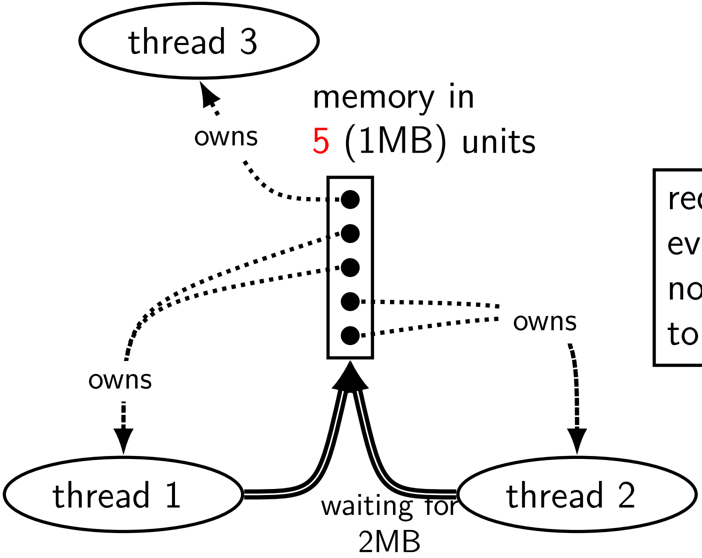


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

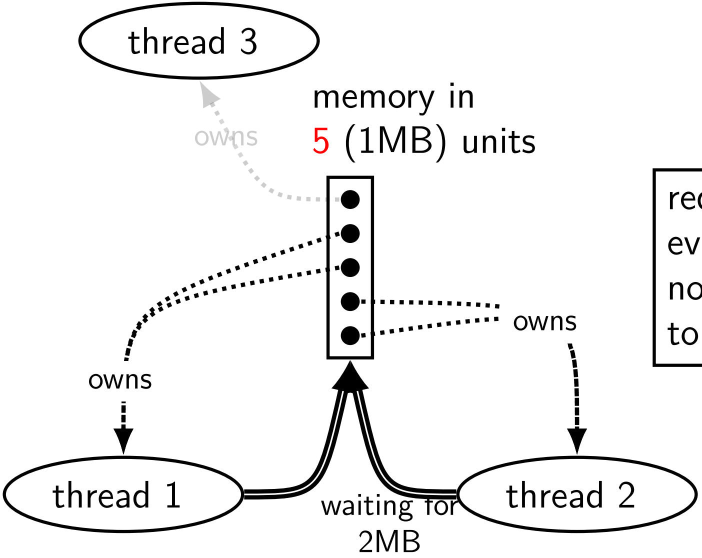


divisible resources: is deadlock



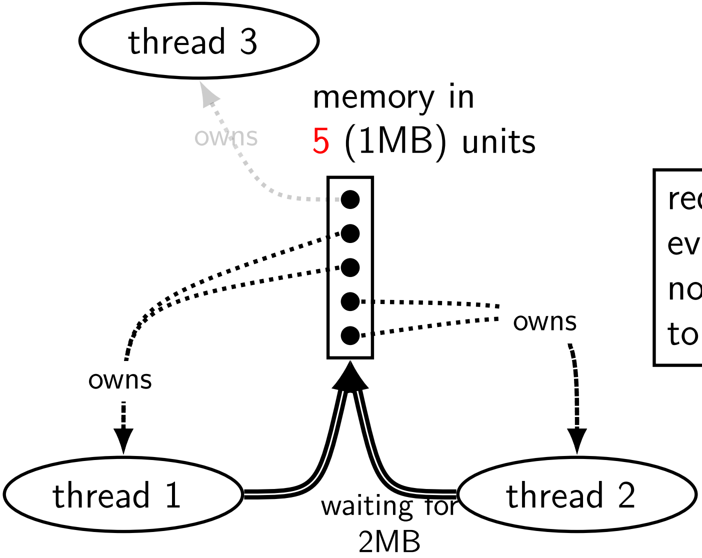
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



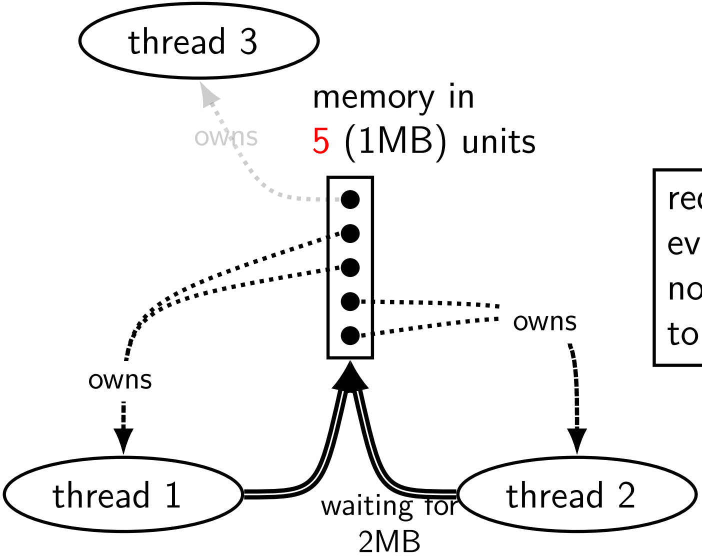
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



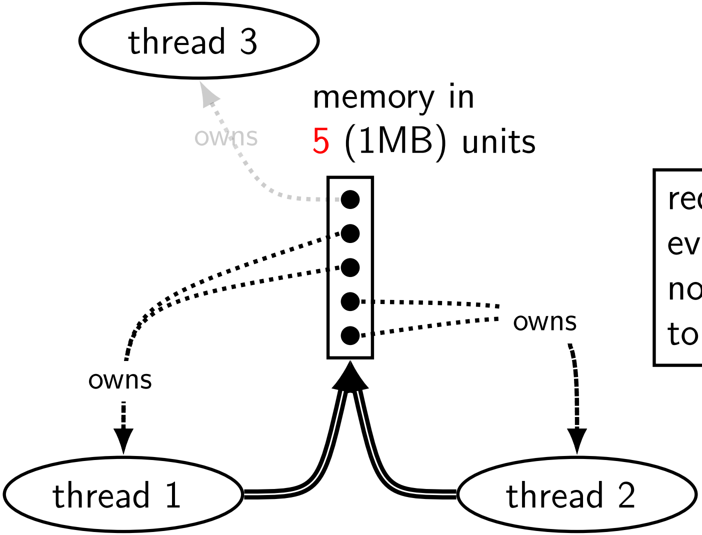
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

deadlock detection with divisible resources

for each resource: track which threads have those resources

for each thread: resources they are waiting for

repeatedly:

- find a thread where all the resources it needs are available
- remove that thread and mark the resources it has as free — it can complete now!

either: all threads eliminated *or* found deadlock

aside: deadlock detection in reality

requires:

- instrumenting contended resources
- "undo" to get out of deadlock

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

related idea: *avoid* deadlock with detection on "what if" scenario

- see Banker's algorithm

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_pipe[0], buffer, HUGE_SIZE);
}
```

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

deadlock waiting

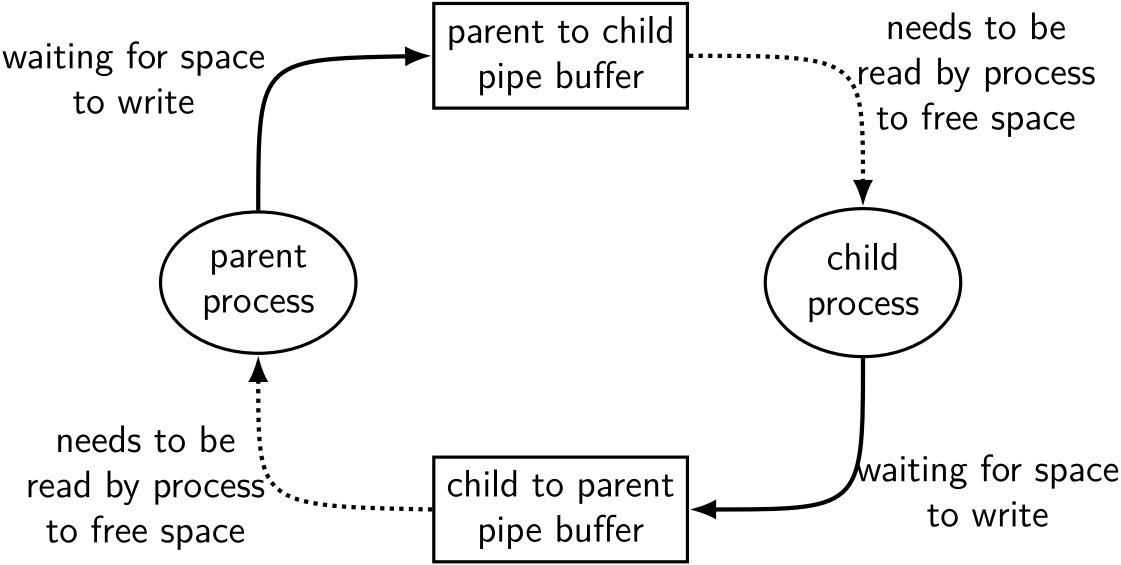
child writing to pipe waiting for free buffer space

…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



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 with free space

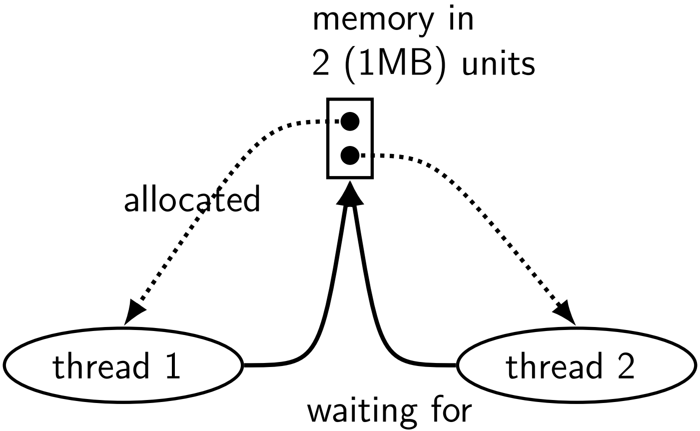
Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

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

free space: dependency graph



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);`

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?

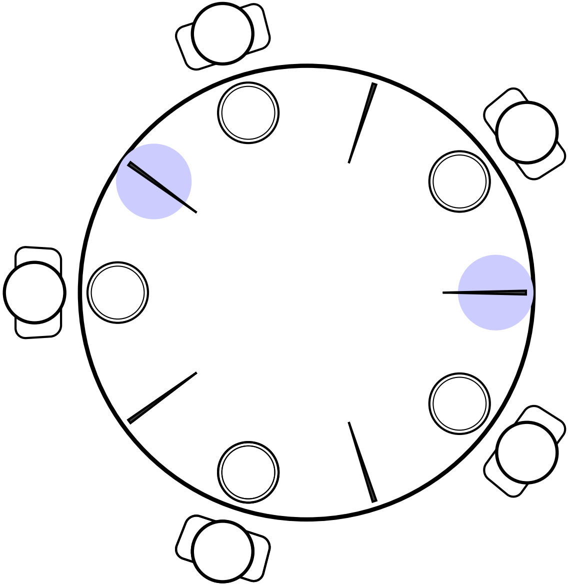
fail/steal with locks

pthreads provides pthread_mutex_trylock — “lock or fail”

some databases implement *revocable locks*

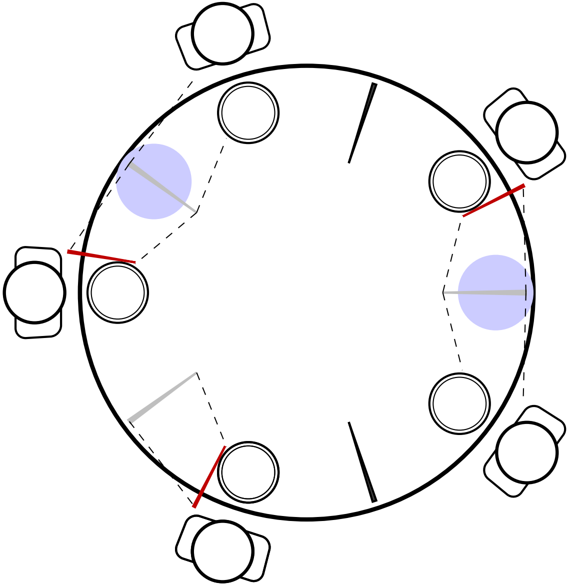
- do equivalent of throwing exception in thread to ‘steal’ lock
- need to carefully arrange for operation to be cleaned up

dining philosophers — ordering



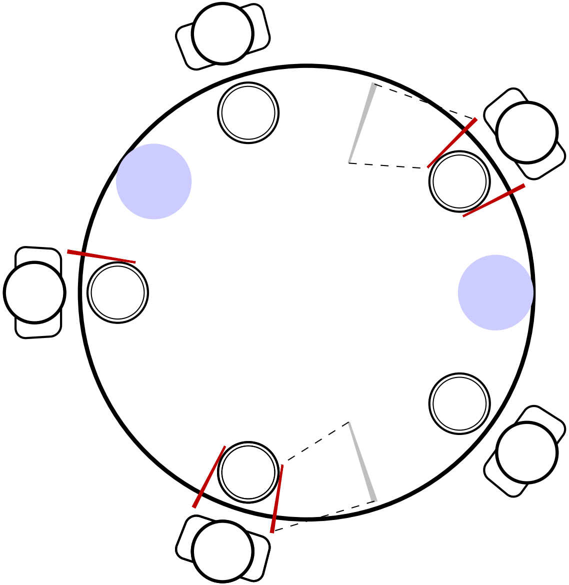
mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

dining philosophers — ordering



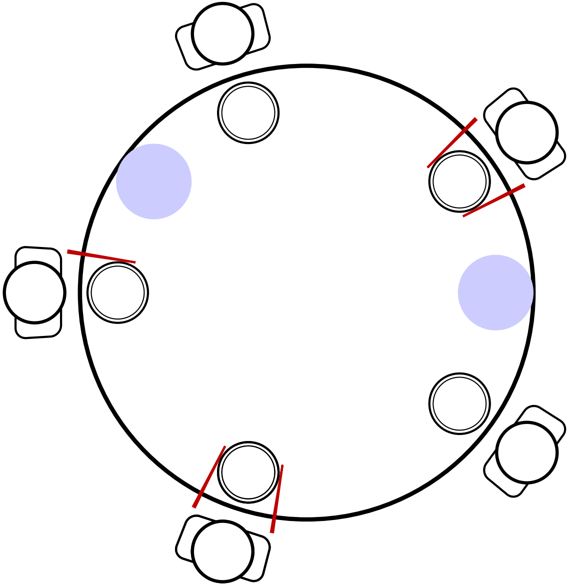
mark some chopsticks places
rule: grab from marked place first
only grab other chopstick after that

dining philosophers — ordering



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

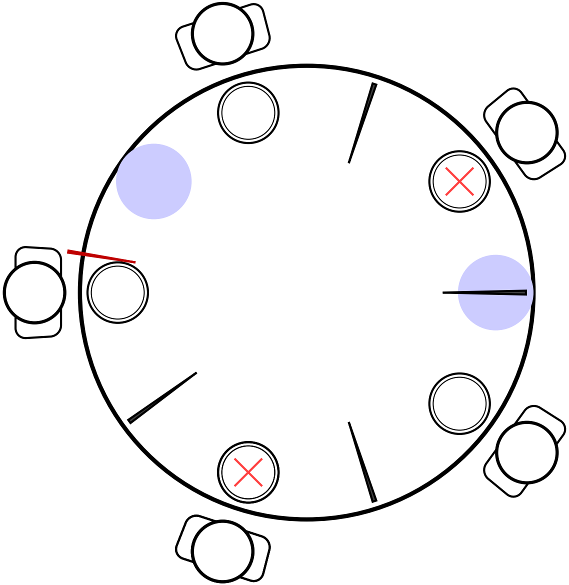
dining philosophers — ordering



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

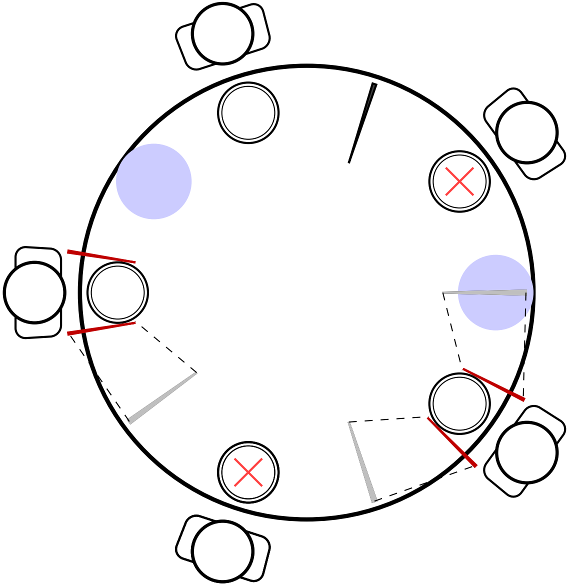
dining philosophers — ordering



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

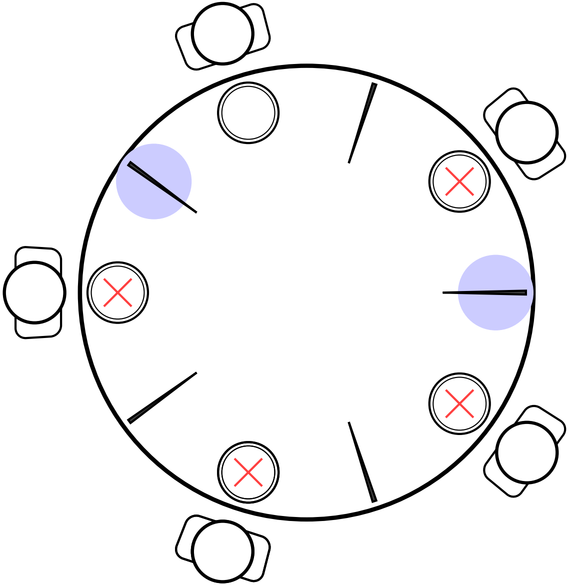
dining philosophers — ordering



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

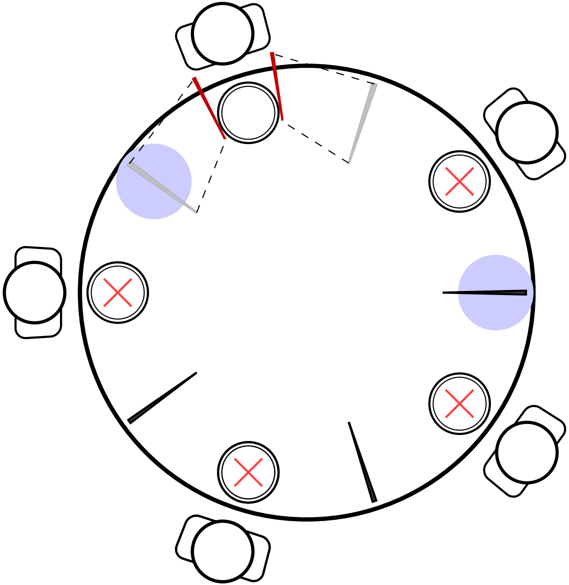
dining philosophers — ordering



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

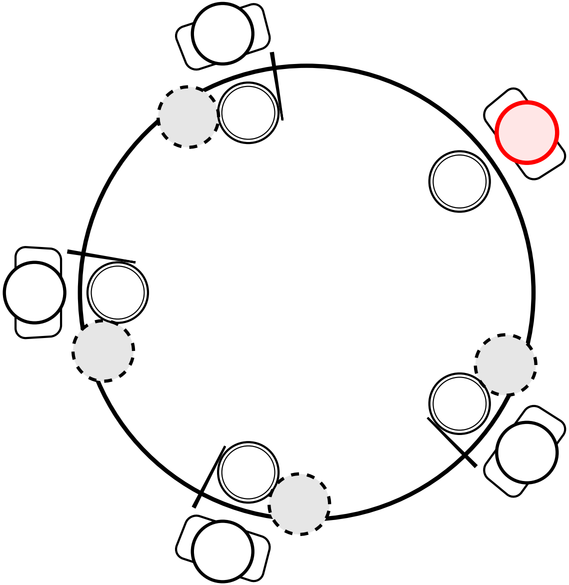
dining philosophers — ordering



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

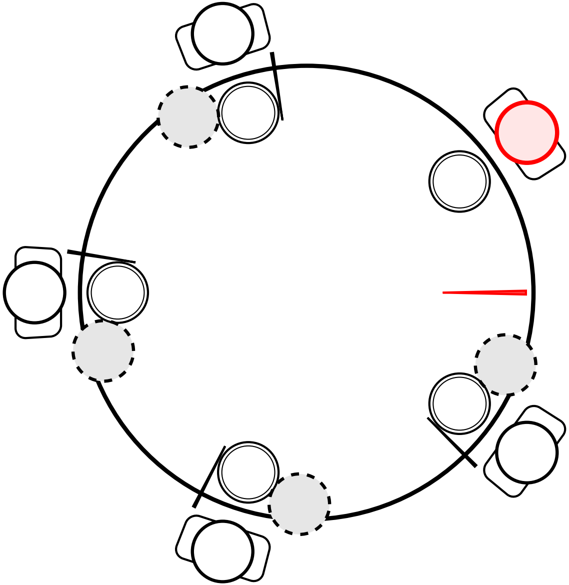
avoids circular dependency,
means everyone else
eventually gets a turn

dining philosophers — aborting



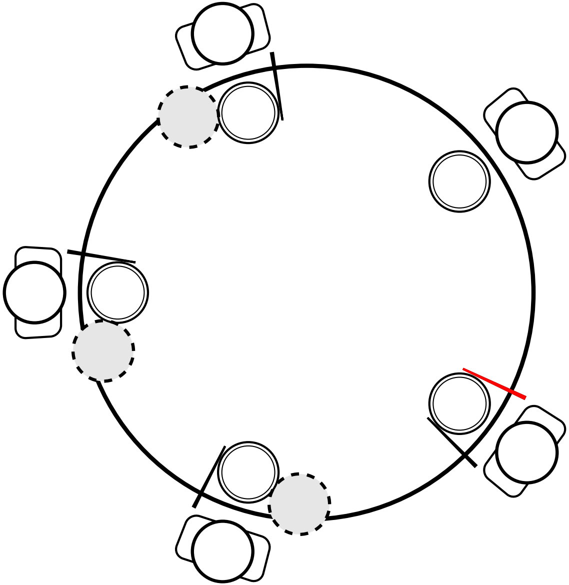
dining philosopher
what if someone's impatient
just gives up instead of waiting

dining philosophers — aborting



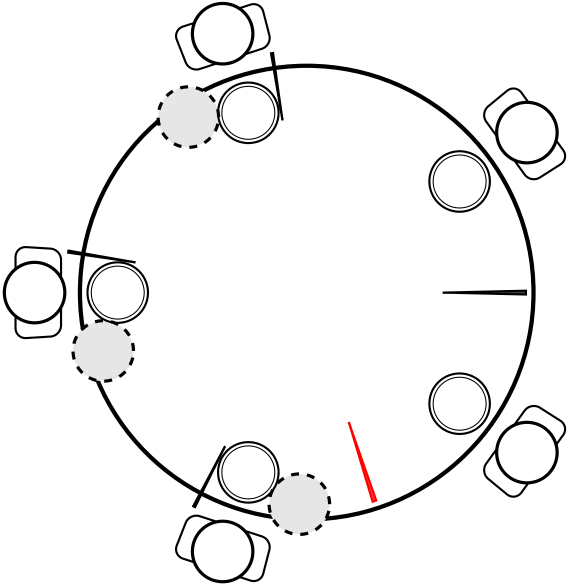
dining philosopher
what if someone's impatient
just gives up instead of waiting

dining philosophers — aborting



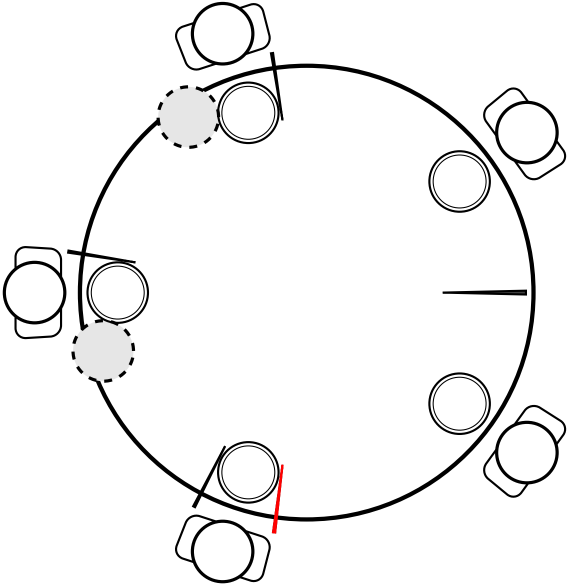
now everyone else can eat

dining philosophers — aborting



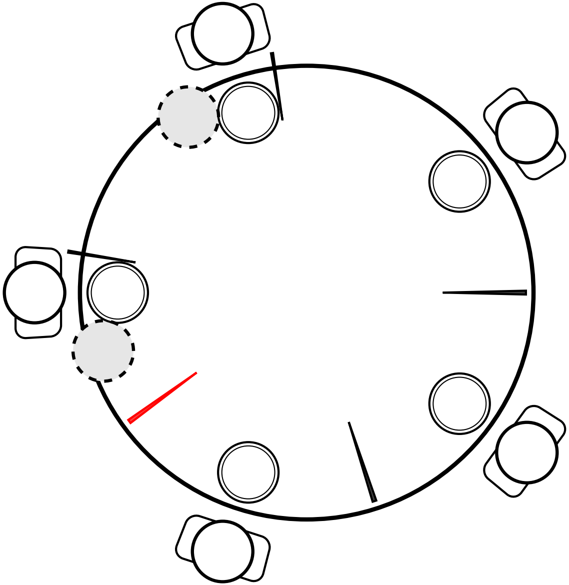
now everyone else can eat

dining philosophers — aborting



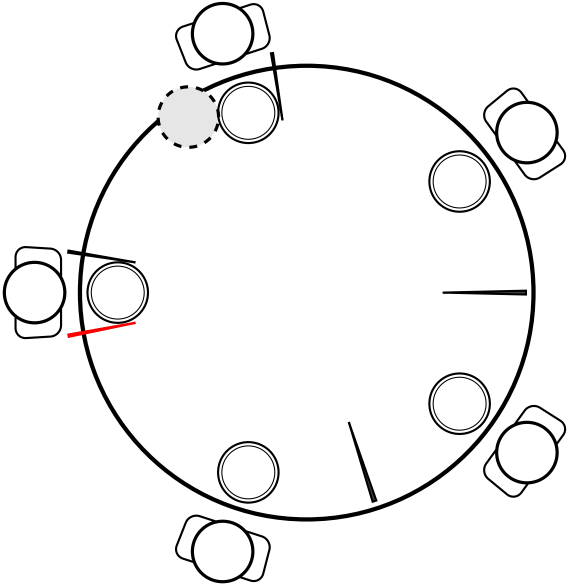
now everyone else can eat

dining philosophers — aborting



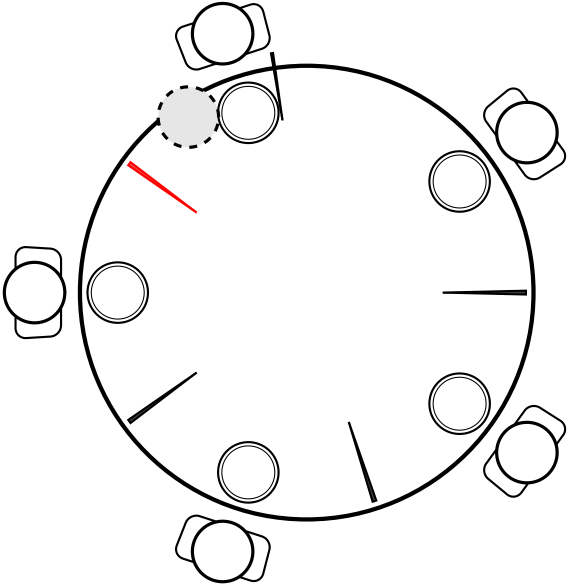
now everyone else can eat

dining philosophers — aborting



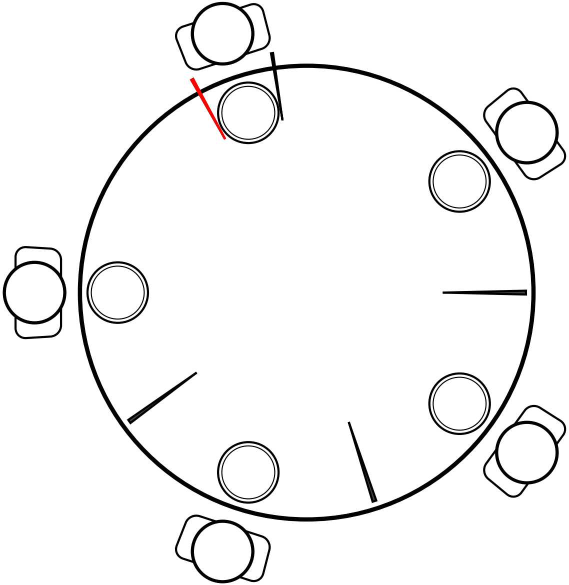
now everyone else can eat

dining philosophers — aborting



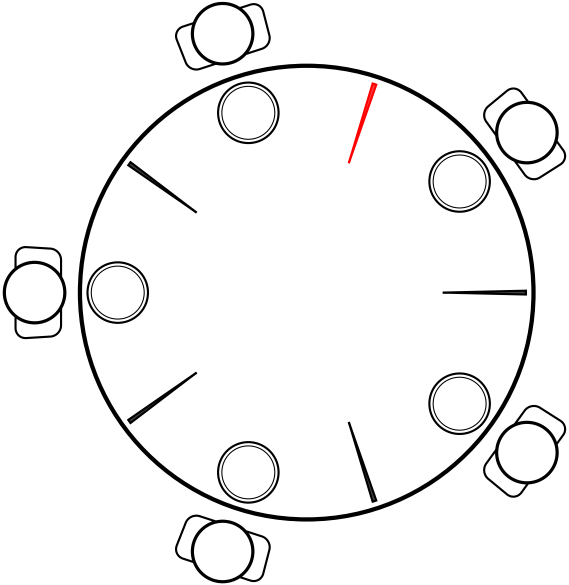
now everyone else can eat

dining philosophers — aborting



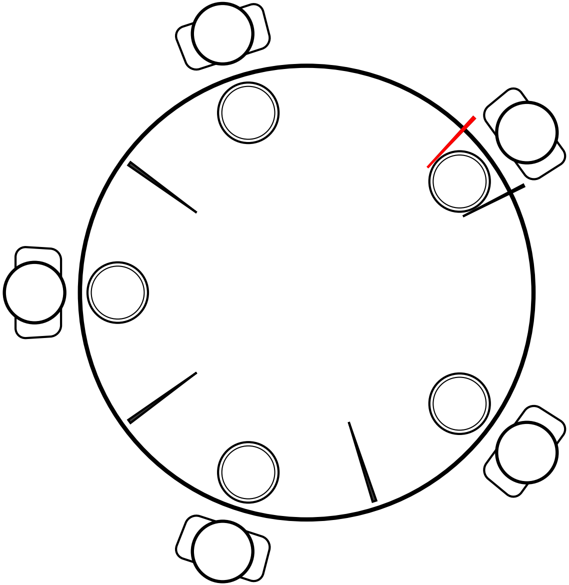
now everyone else can eat

dining philosophers — aborting



now everyone else can eat

dining philosophers — aborting



and person who gave up
might succeed later

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 Banker's algorithm