Synchronization between threads and processes

Conditional synchronization

Mutual exclusion

Deadlocks & Livelocks

Semaphores

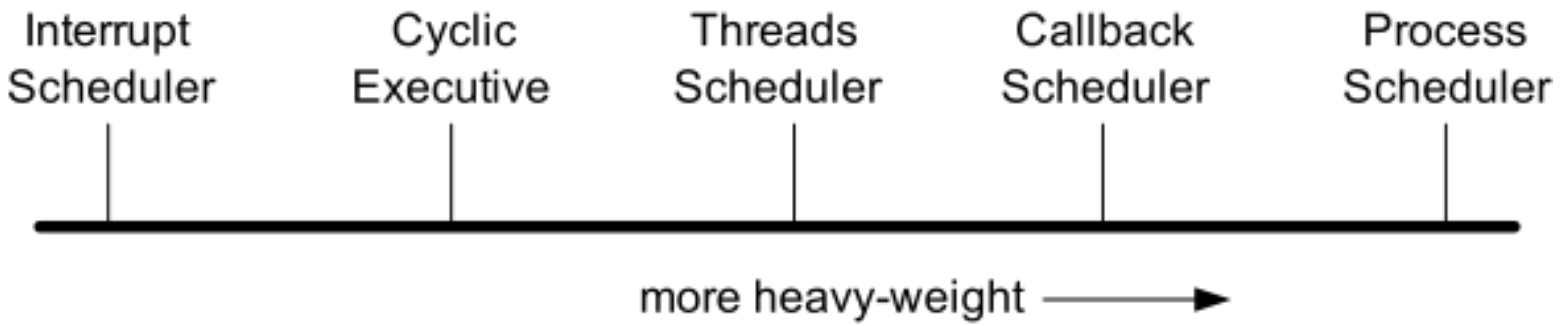
Dining Philosophers

Lecture overview

- In these slides, you will learn about various mechanisms that are available for synchronization between processes.
- We will talk specifically about semaphores, and their implementation in POSIX (mutex).
- POSIX also has condition variables, but we won't talk about them here.
- By the end of this slide set, you should be able to look at a RTOS and be able to understand what synchronization mechanisms are available.

About processes and threads

- **We have talked about how to create processes and threads and we've talked about how they can communicate. But they also need to synchronize.**

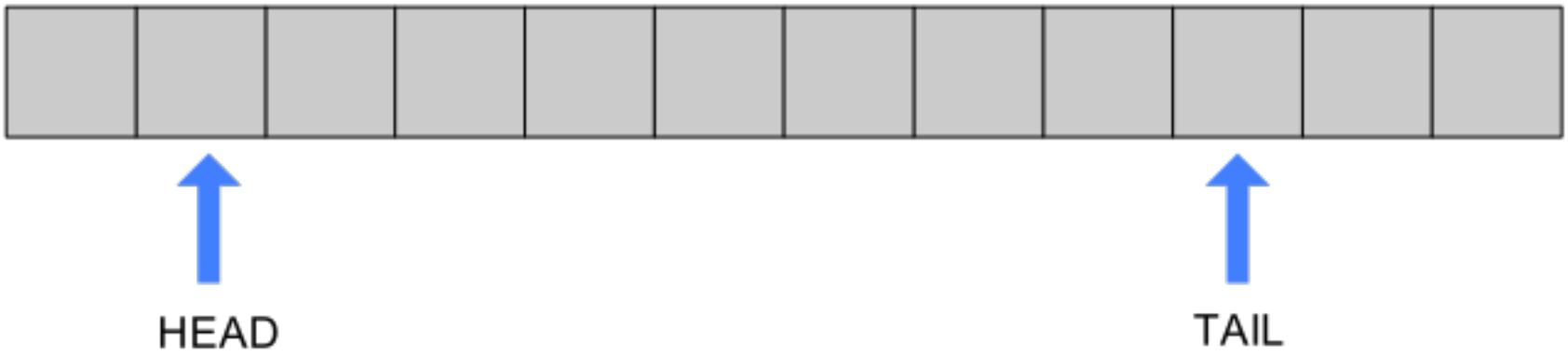


Synchronization

- Two reasons processes or threads have to synchronize:
- Condition synchronization: Needed when a process/thread wishes to perform an operation that can only sensibly/safely be performed if another process/thread has taken some action or is in some state.
- Mutual exclusion: Needed when more than one process/thread wants to share data, to ensure that the data is shared consistently.

Condition synchronization

- Consider a bounded buffer
  - You shouldn't try to add elements when the buffer is full
  - You shouldn't try to remove an element when the buffer is empty



Mutual exclusion

- Suppose two threads update a shared variable with:
  - x = x + 1
- Problem: This is probably implemented in machine code
  - read x from memory into a register
  - add 1 to the register
  - store the register back into memory (x)
- If two tasks try to do this at the same time, x might not be updated properly.

Busy waiting (Spinning)

- Busy waiting: use shared variables as flags

```
/* Waiting process */

while (flag == down) {
      do nothing
}
……
```

```
/* Signaling process */

…
flag = up;
…..
```

- Fine for condition synchronization (except that it wastes CPU time)
- Bad for mutual exclusion → Deadlock

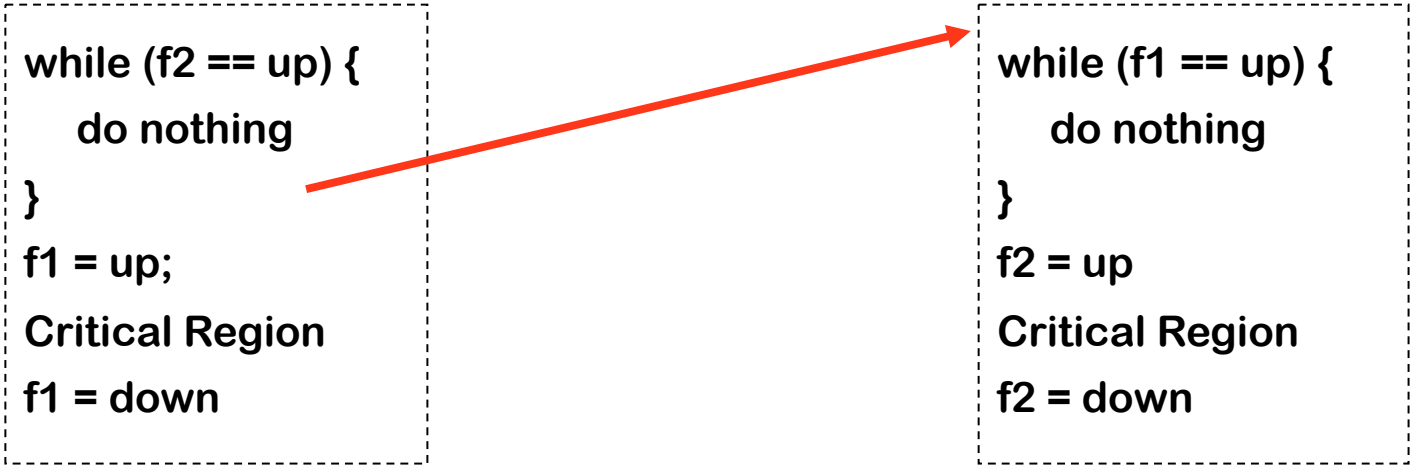
Busy waiting → deadlock (bad!)

```
f1 = up
while (f2 == up) {
    do nothing
}
Critical Region
f1 = down
```

```
f2 = up
while (f1 == up) {
    do nothing
}
Critical Region
f2 = down
```

- Intention: if a thread wants to enter the critical region, it raises its flag. If the other thread wants it, and sees the other thread's flag raised, it waits.
- Problem: What if the context switch occurs immediately after one thread raises its flag, causing the other thread to raise its flag?

Is this an improvement?



No, this can fail to provide mutual exclusion (both processes/threads could end up in the critical region at the same time).

A technique that would work

```
f1 = up
turn = 2
While (f2 == up and turn == 2) {
   do nothing
}
critical region
f1 = down
```

```
f2 = up
turn = 1
While (f1 == up and turn == 1) {
   do nothing
}
critical region
f2 = down
```

Can you convince yourself that:

- Only one thread can be in its critical region at a time
- Deadlock can not occur

It would be nice if we had some OS support for this sort of thing…

Semaphores

Define two building blocks:

```
wait(s):   if (s > 0) then {
               s = s – 1;
           }
           else {
               delay until s > 0
               s = s – 1;
           }
```

These routines are indivisible!

Historical notation

wait is called P; signal is called V

Signal(s): s = s + 1

Terminology due to Edsger Dijkstra
Proberen te verlangen (wait) [P]
verhogen (post) to increase a semaphore [V]

Mutual exclusion with semaphores

The number of threads allowed in the critical section depends on the initial

value of the semaphore:

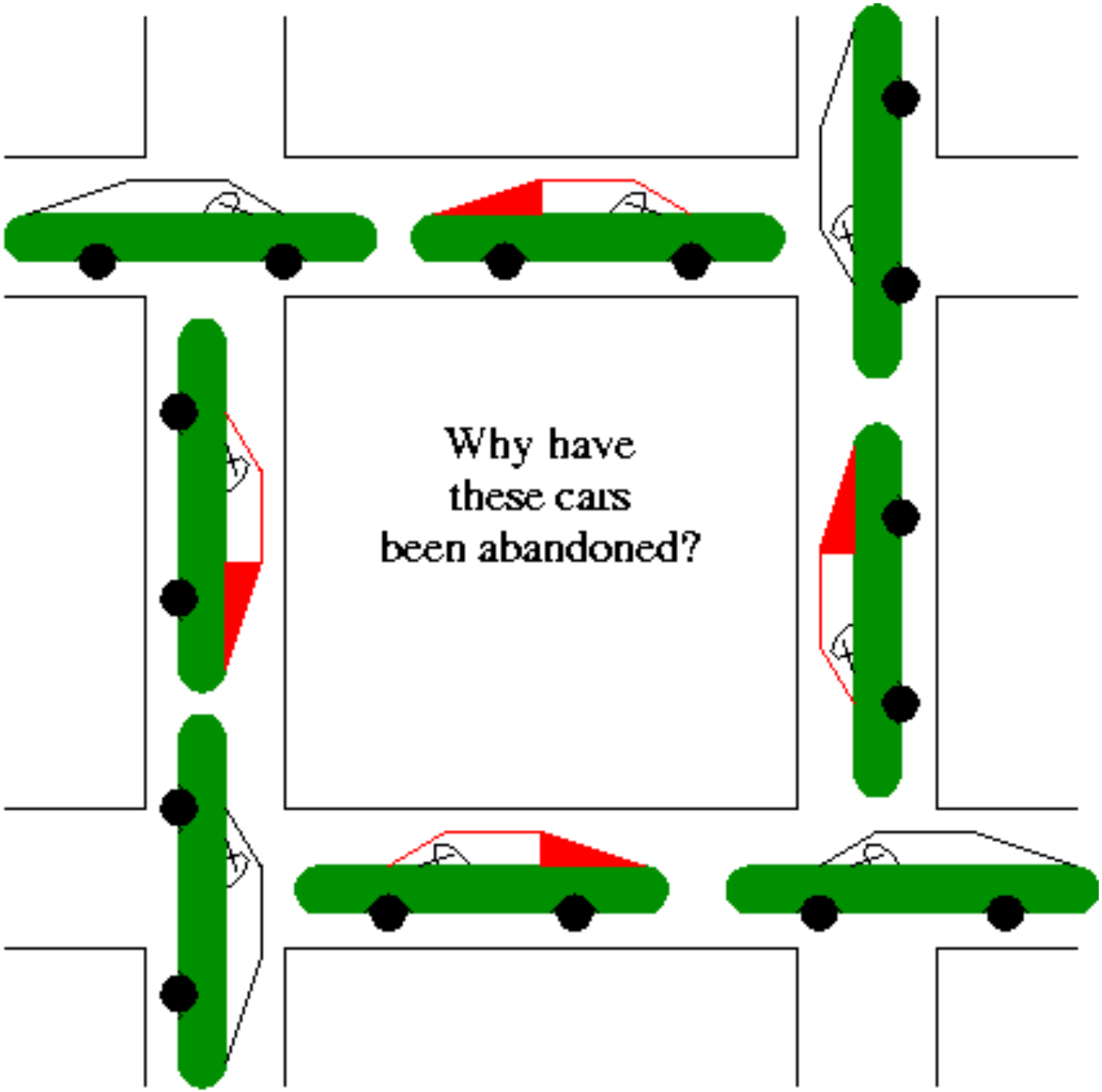
- initial value of s means that s threads are allowed in at once
- don't initialize it to 0

Binary semaphore: initialize it to 1

Semaphores in POSIX

- pthread_mutex_t mutex;

- thread_mutex_init (& mutex, NULL);
- …
- pthread_mutex_lock(& mutex);
- < critical section>
- pthread_mutex_unlock (& mutex);
- ….


Why have
these cars
been abandoned?

Watch out for deadlocks

- **The Dining Philosophers** (probably seen in EECE 315)
- Five philosophers sitting around a table. There is one chopstick between each pair of philosophers (so 5 chopsticks)
- Each eats and thinks
- When one wants to eat, she/he must grab both chopsticks (if one is not available, the philosopher must wait)
- Three criteria:
  - Would like to minimize waiting time
  - Must avoid deadlock
  - Avoid starvation

See the last few slides for more about this problem

Many solutions have been proposed

- Dijkstra
- Chandy & Misra

(More reading: http://en.wikipedia.org/wiki/Dining_philosophers_problem)

Dealing with deadlocks

- The Ostrich Algorithm.
- The "put your head in the sand approach''.
- If the likelihood of a deadlock is sufficiently small and the cost of avoiding a deadlock is sufficiently high it might be better to ignore the problem. For example if each PC deadlocks once per 100 years, the one reboot may be less painful that the restrictions needed to prevent it.
- Clearly not a good philosophy for nuclear missile launchers.

Do ostriches bury their head in the sand?

Dear Yahoo!:
Do ostriches really bury their heads in the sand?
Joe
Alma, Arkansas

Dear Joe:

Strangely, a Yahoo! search on "ostrich" yielded sites focused on ostrich farming and little else. After scratching our heads, we decided to check Yahooligans!, Yahoo!'s directory for kids, which has an impressive animal category of its own.

At Yahooligans!, we drilled down to the Birds > Types of Birds > Ostrich category, which contained several non-commercial sites. We liked the sound of New Eclectic Ostrich*, so we eagerly clicked the link. We found the answer to your question in the ostrich myth section. It states, "Perhaps the most enduring myth about the ostrich is that it hides its head in the sand when in danger."

We were satisfied with that, but wanted to learn more, so we returned to Yahooligans! and chose a different ostrich link. We arrived at a page from The Canadian Museum of Nature that further elaborated on the myth:

If threatened while sitting on the nest, which is simply a cavity scooped in the earth, the hen presses her long neck flat along the ground, blending with the background. Ostriches, contrary to popular belief, do not bury their heads in the sand.

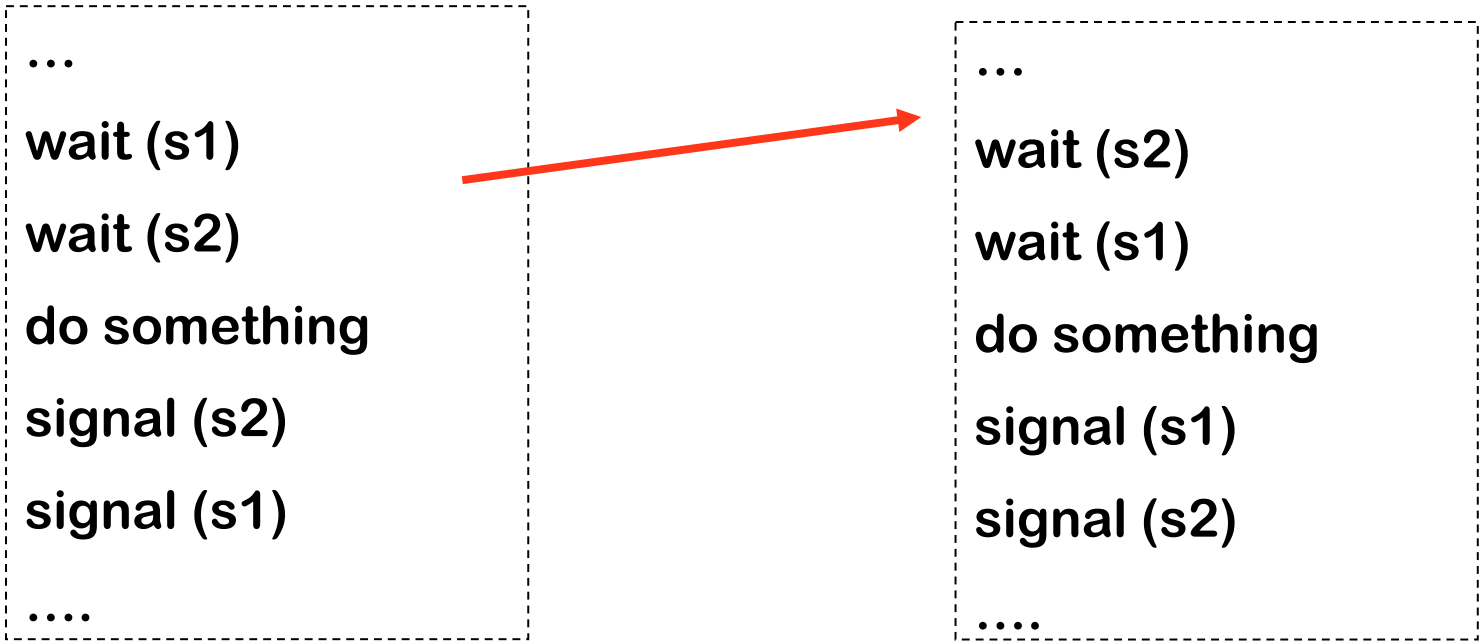
That sounded reasonable to us, and since ostriches grow up to 8 feet tall and weigh up to 300 pounds, we decided that first-hand research was out of the question. So, from what we've deduced, our final answer is ... no.

Detecting deadlocks

- It is possible to create a process that runs in the background and watches for deadlock.
- But what do you do when you find deadlock?
  - Preemption: take away a resource. Probably difficult to implement.
  - Rollback: if the system has made check-points, roll back to a recent check-point. Somehow still need to guarantee forward progress.
  - Kill a process: might be painful.
- Much better to avoid deadlock in the first place.

Deadlocks

- Suppose we have two threads that want access to two shared resources.
- In general, this sort of thing could cause deadlock.



Deadlock avoidance

- But if we lock resources in the same order, we are ok.

```
…
wait (s1)
wait (s2)
do something
signal (s2)
signal (s1)
….
```

```
…
wait (s1)
wait (s2)
do something
signal (s2)
signal (s1)
….
```

The problem with semaphores

- Programmers are only human.
- It is easy to make a mistake when you use semaphores.
- If you misplace or omit just one wait or signal, your program may go into deadlock, or mutual exclusion may not be guaranteed.
  - What's even worse, it might happen in only some rare but critical event.
- It would be nice to have something a bit more structured.

Conditional critical regions

- Language construct to specify regions of code that run in mutual exclusion.
- You can define a REGION and associate each region with a GUARD condition. The REGION is only entered when the guard condition is true.
- Problems
  - Guard condition has to be evaluated every time you leave a critical region.
  - Still not very structured: Regions can be distributed anywhere in the program.

Protected objects (Ada)

- Protected procedure
  - mutual exclusion
  - read/write access to encapsulated data
- Protected function
  - read only access
  - executes only when no procedure is running
- Protected entry
  - like protected procedure with guard (Boolean expression)
  - executes when guard = TRUE

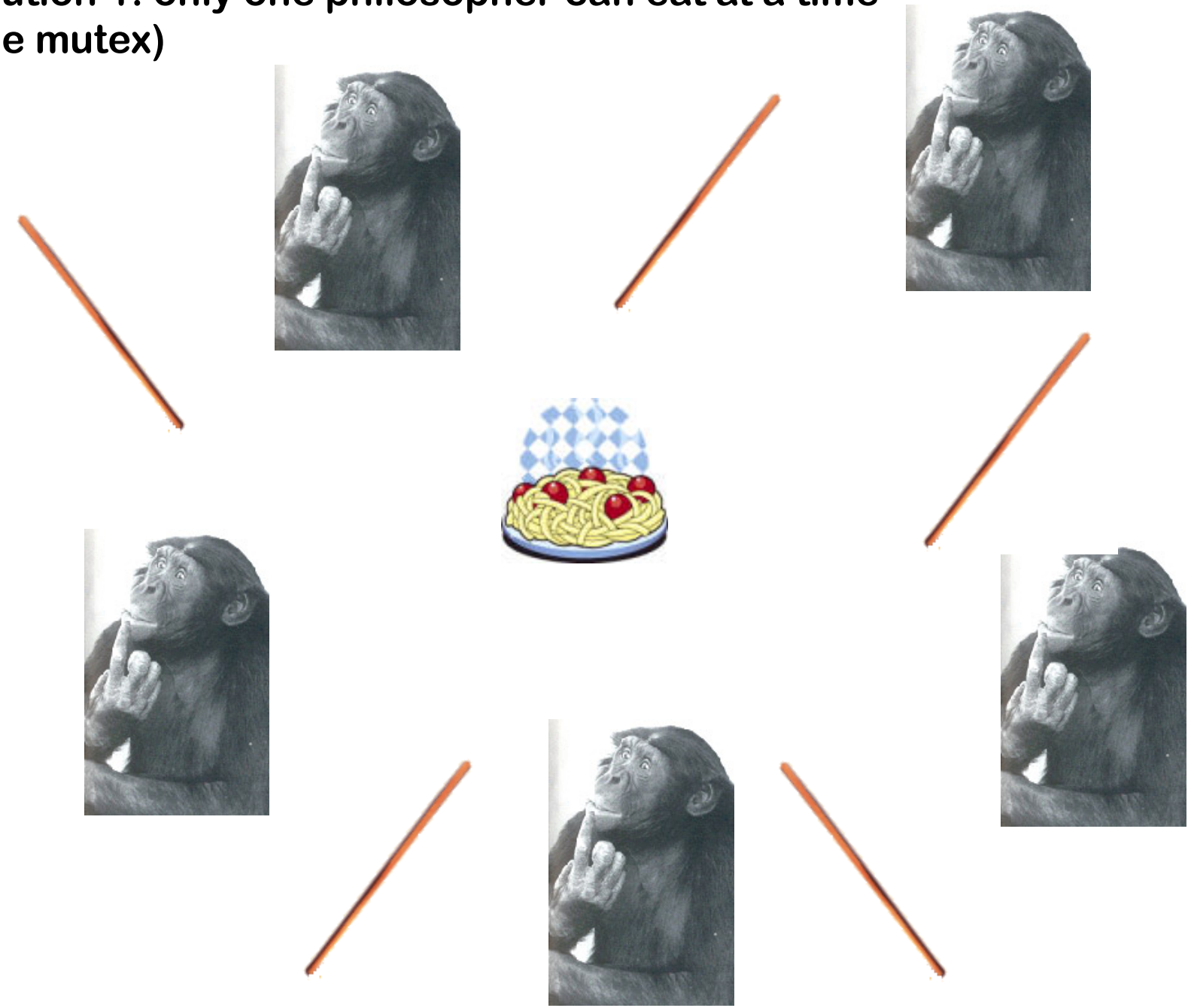
Highlights

- We discussed various mechanisms for synchronization between processes.
  - We focused on the POSIX mutex primitives.
- We also talked a bit about deadlocks and how to avoid deadlocks.
  - The Ostrich approach: just ignore it. Not a good idea for a real-time system.
  - Deal with it: not usually feasible.
  - Avoid it.
- The most important thing to remember from this slide set is that synchronization is an important part of any RTOS that supports multiple threads or processes.

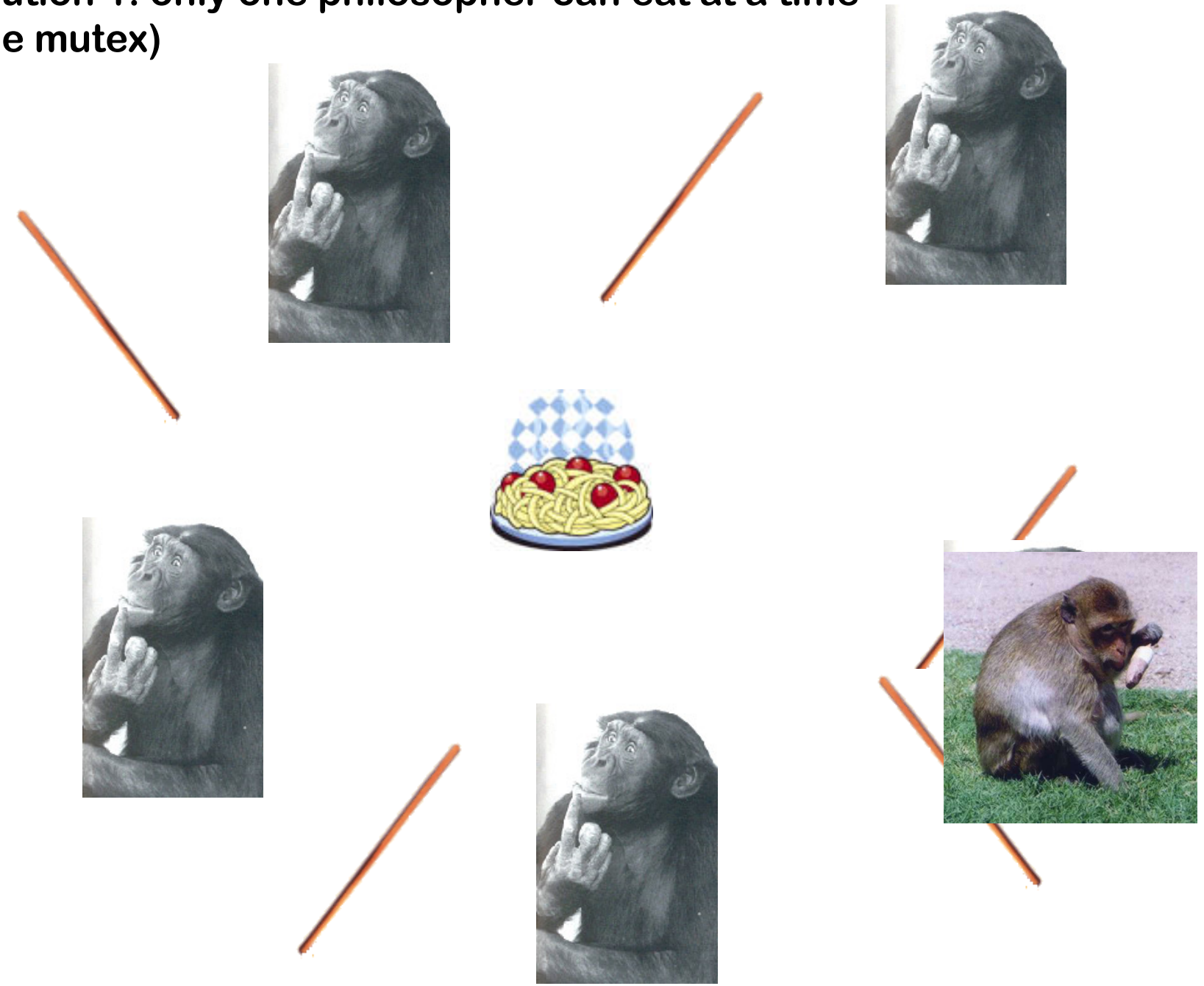
More about the dining philosophers

- A fundamental problem with shared resources.
- Requires careful resource allocation.

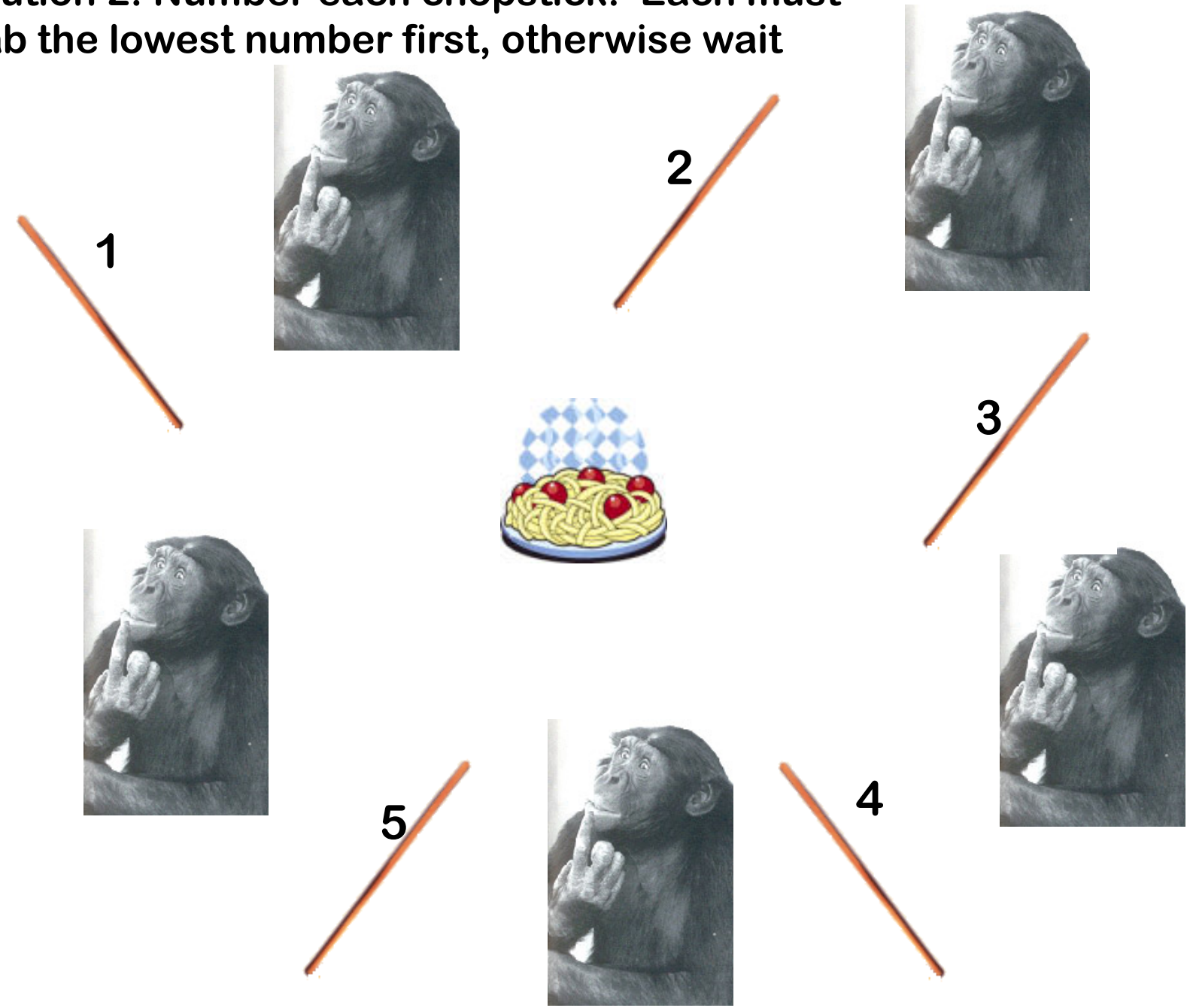
Solution 1: only one philosopher can eat at a time (one mutex)



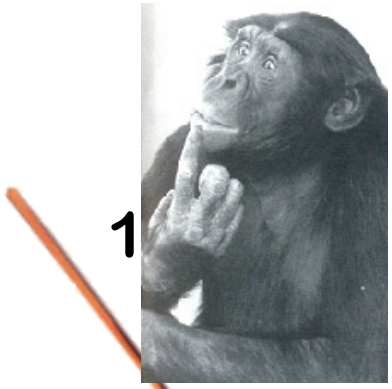
Solution 1: only one philosopher can eat at a time (one mutex)

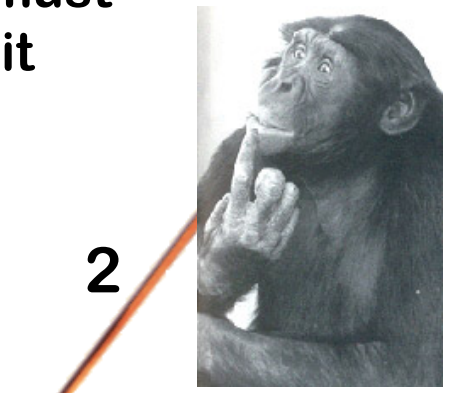


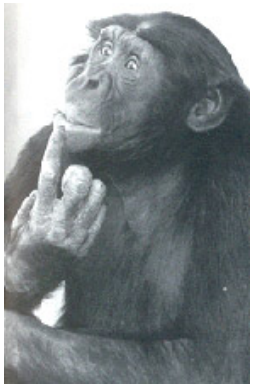
Solution 2: Number each chopstick. Each must grab the lowest number first, otherwise wait

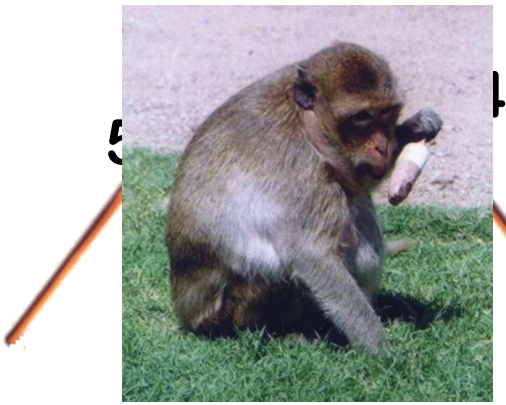


Solution 1: Number each chopstick. Each must grab the lowest number first, otherwise wait









Dining philosophers

- **Worst case, only one philosopher is eating.**
  - **But on average, more than one can be eating at a time.**
  - **Does this scheme guarantee no deadlock?**
  - **Convince yourself one way or the other.**
  - **What about starvation?**