Course Operating Systems

Lecture 6: Classic Synchronization Problems and Resource Allocation with emphasis on Deadlock Prevention

EDA093, DIT 401

Study Period 1

Ack: several figures in the slides are from the books
- Modern Operating Systems by A. Tanenbaum, H. Bos
- The art of multiprogramming, by M. Herlihy, Shavit
- OS Concepts by Silberschatz et-al
- Operating systems by W. Stallings

Classic Problems of Synchronization

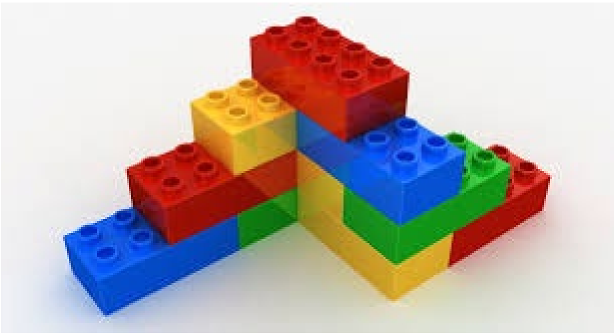
- Bounded-Buffer (producer-consumer) today
- Dining-Philosophers (Resource allocation: we will use it as running example problem, to study deadlock prevention): today
- Narrow Bridge (in Synchronization Exercise session: will be needed for your upcoming labs; synchronization&scheduling problem): today, with Hannah
- Readers and Writers (paved the way to lock-free/wait-free synch) next lecture
- Sleeping barber, and more such fun ☺

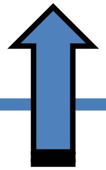
practice these: it is useful and fun!

Reflect: this is what we are doing …

Construct: objects / solve specific synchronization problems

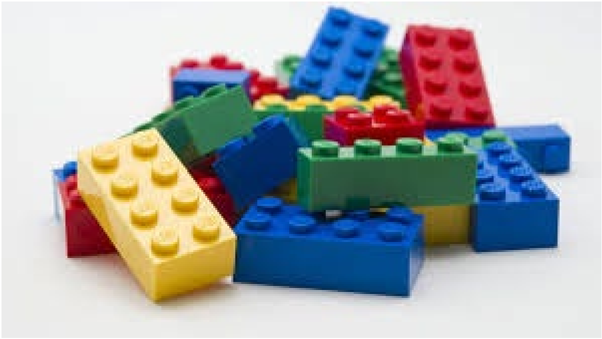
- 2 thread CS, n-thread-CS
- Semaphores, mutex-locks, …
- Producer-consumer (bounded buffer)
- Dining philosophers
- Transactions
- …



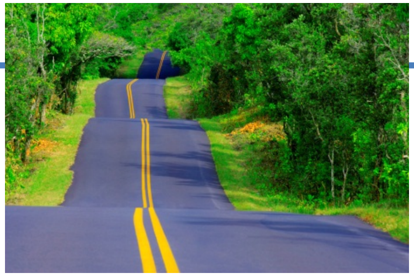


Using: primitives

- R/W variables
- RMW variables
- Transactions
- Semaphores, etc
- …



Roadmap



- The bounded buffer producer-consumer problem
- Resource allocation (dining philosophers is such a problem)
  - Intro
  - We elaborate on deadlocks
- What is the problem with the Dining philosophers…
  - … and how to help them
  - Well, we failed; let's try to eliminate the deadlock's necessary conditions
    - First the cyclical wait
    - Then the no-preempt
    - Now the hold-and-wait

Bounded producer-consumer buffer: requirements

Buffer with **space for N** items;

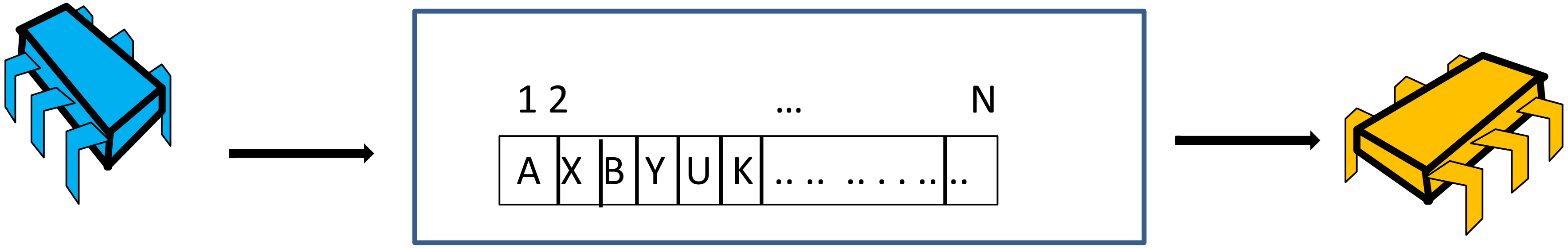
accessing common entries is a **critical section**

Producer inserts items;

must wait if buffer full

Consumer removes items;

must wait if buffer empty



Solve this synch problem using semaphores

Bounded producer-consumer buffer: what synch do we need?

- **Producer** inserts items; must wait if buffer full
- **Consumer** removes items; must wait if buffer empty
- Accessing the buffer is a **critical section**

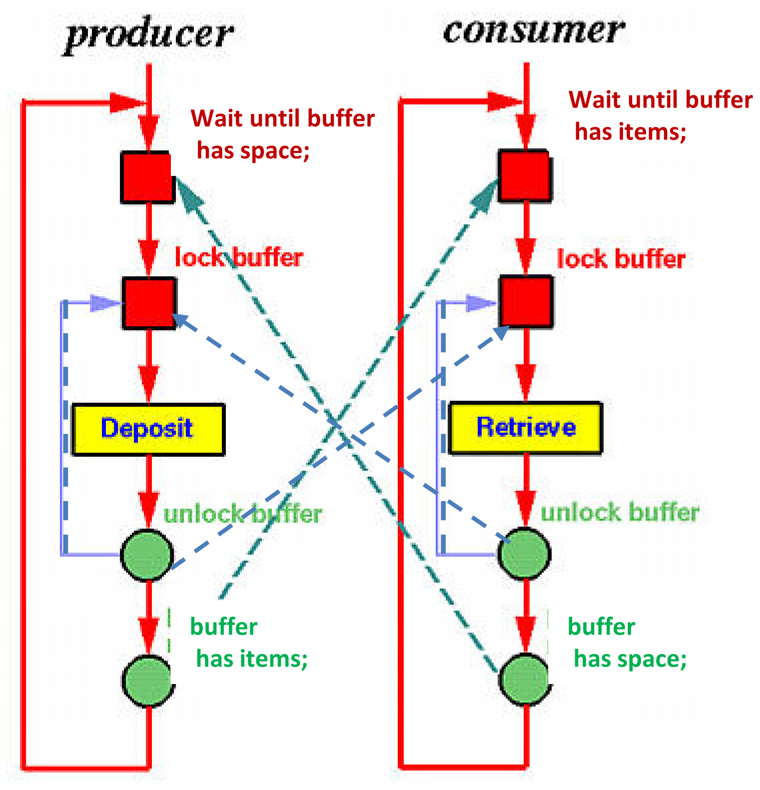


fig C.K. Shene
http://www.cs.mtu.edu/~shene/NSF-3/e-Book/

Bounded producer-consumer buffer: a solution

Synchronization variables:

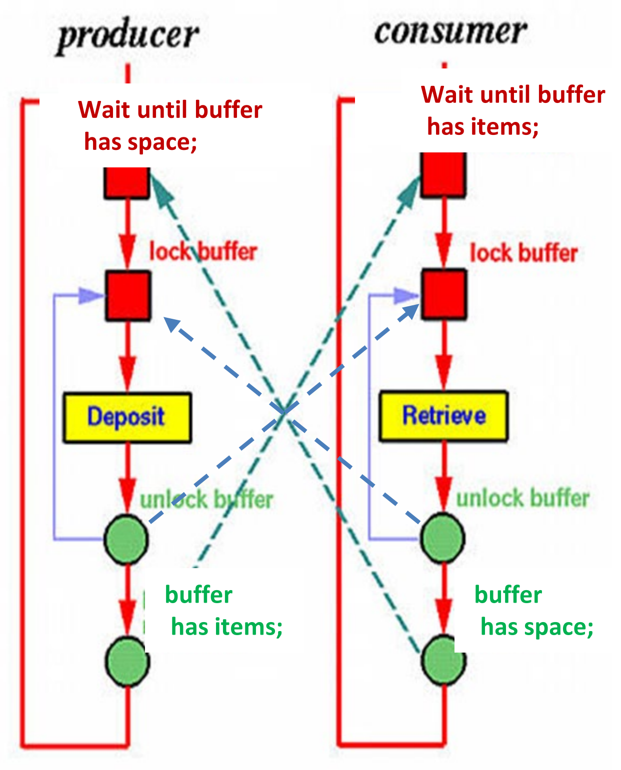
- Binary semaphore mutex_sem initialized to 1
- General semaphore buffer-has-items initialized to 0
- General semaphore buffer-has-space initialized to N

producer

```
do {
        //   produce  item

        wait (buffer-has-space);
        wait (mutex_sem);

        //  add  item to  buffer

        signal (mutex_sem);
        signal (buffer-has-items);
} while (TRUE);
```

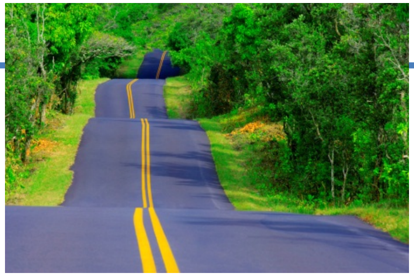
consumer

```
do {
        wait (buffer-has-items)
        wait (mutex_sem);

        //  remove item from buffer

        signal (mutex_sem);
        signal (buffer-has-space);

        //  use the item
} while (TRUE);
```



Homework: write arguments about correctness, i.e.to show that the solution meets the requirements

Roadmap



- The bounded buffer producer-consumer problem
- **Resource allocation (dining philosophers is such a problem)**
  - **Intro**
  - We elaborate on deadlocks
- What is the problem with the Dining philosophers…
  - … and how to help them
  - Well, we failed; let’s try to eliminate the deadlock’s necessary conditions
    - First the cyclical wait
    - Then the no-preempt
    - Now the hold-and-wait

What is resource allocation?

Processes/threads need resources (eg memory pages, printer, access to parts of shared data structure, etc)

- **Our focus: reusable resources:**

Eg. a human analogy: process = go fishing; needed resources: boat, fishing-rod

Process/thread P structure

do

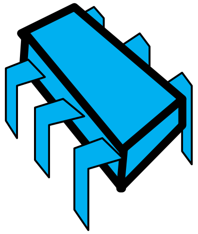
request resources (i.e. entry section)

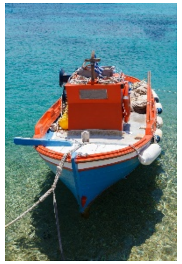
// use them

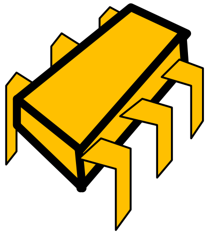
release resources (i.e. exit section)

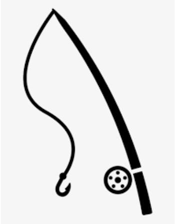
// remainder section

forever





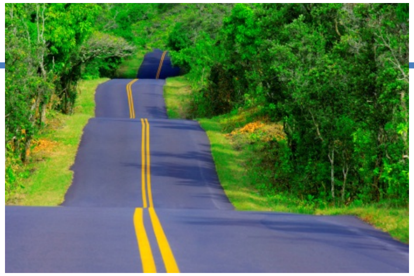




To solve the problem: provide the method for **each process to acquire <u>all</u> its needed resources and release them,** and guarantee (as in the Critical Section problem):

1. **Mutual exclusion**: each resource is used by only one process at a time
2. **Progress**: no deadlock
3. **Fairness**: FCFS, or no starvation, or other fairness formulation

Roadmap



The bounded buffer producer-consumer problem

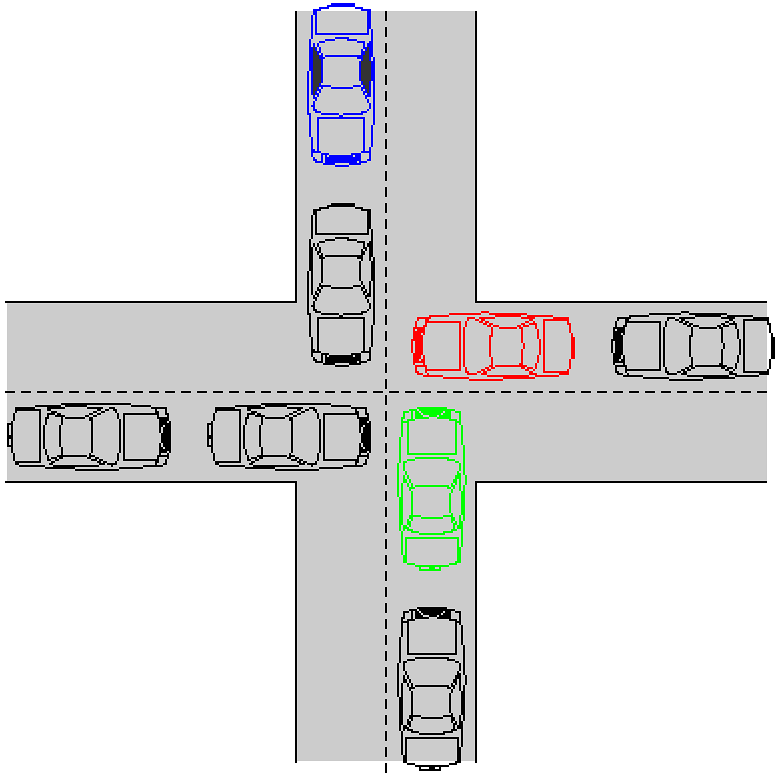
Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- … and how to help them
- Well, we failed; let's try to eliminate the deadlock's necessary conditions
  - First the cyclical wait
  - Then the no-preempt
  - Now the hold-and-wait

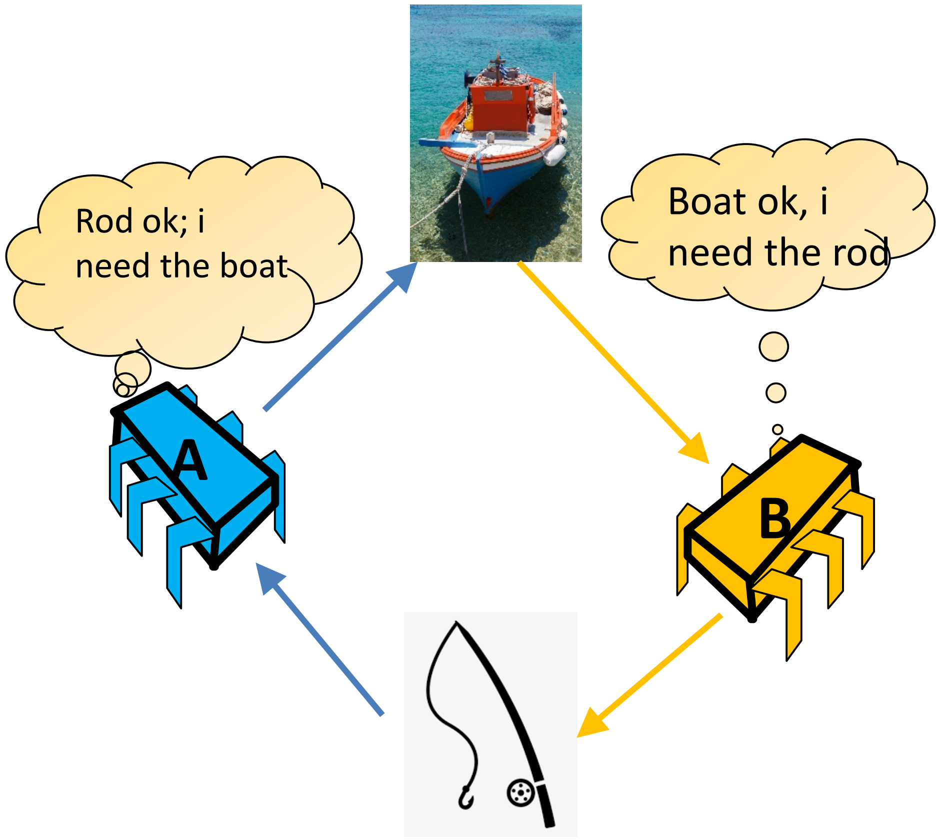
What is a deadlock?



A set of processes/threads blocking each-other s.t. none of them can proceed:
How can it occur?

4 necessary conditions for Deadlock [Coffman et al 1971]

Theorem: all 4 conditions hold simultaneously when a deadlock occurs:



1. **Mutual exclusion:** only one process at a time can use a resource.
2. **Hold and wait:** a process holding some resource can request additional resources and wait for them if they are held by other processes.
3. **No preemption:** a resource can only be released by the process holding it, after that process has completed its task.
4. **Circular wait:** there exists a circular chain of 2 or more blocked processes, each waiting for a resource held by the next proc. in the chain

let’s think together: (ie as in )

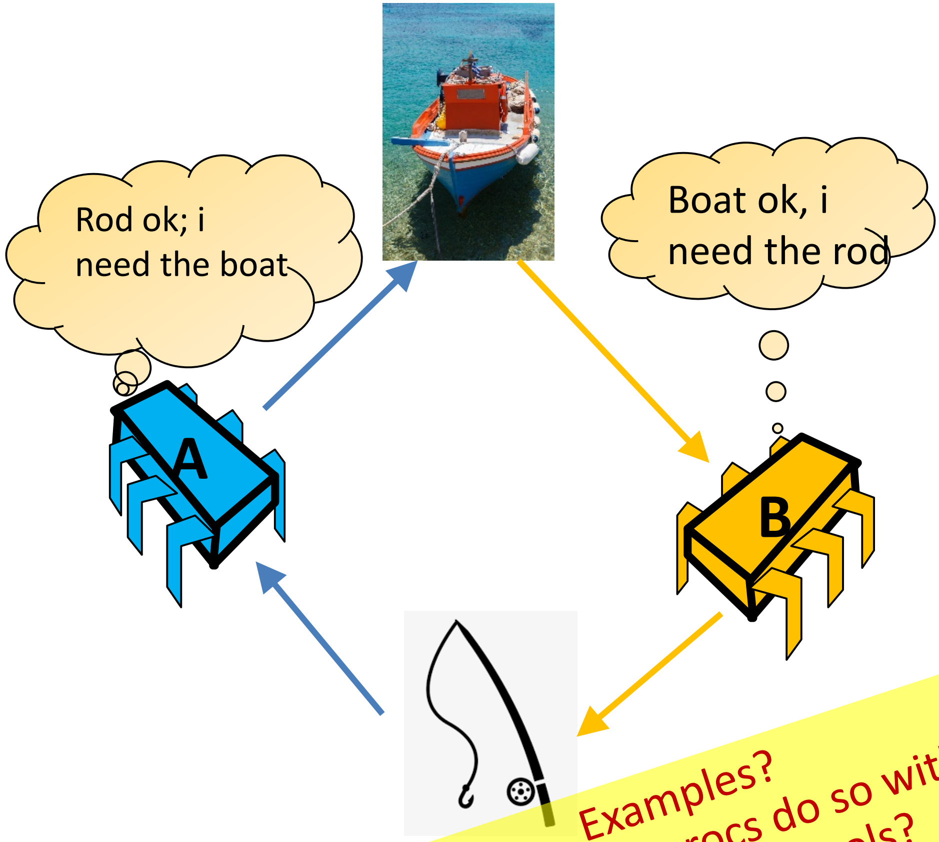
Q: What does the theorem imply wrt deadlock prevention?

A: see next slide ☺

Resource Allocation with Deadlock Prevention

How can a solution to RA be RESPONSIBLE AND PREVENT?

Restrain the ways requests can be made; eliminate at least one of the 4 conditions, so that deadlocks are impossible to happen. **How?**



Examples?
How to help procs do so with synchronization tools?

Eliminate Mutual Exclusion – (cannot do much here …)

Eliminate Circular Wait – how? E.g. impose that resources are acquired in a certain order

- e.g always first the boat, then the rod

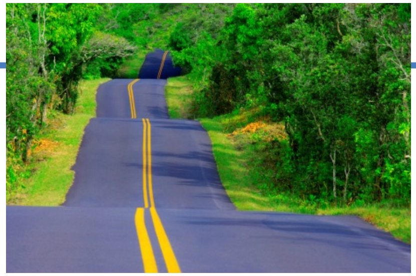
Eliminate No-Preemption – how? a process holding some resources & requesting another that is occupied, it releases the held resources and has to request them again.

- Eg be polite: B releases the boat for A to proceed (after which A releases both and B can proceed)

Eliminate Hold and Wait – how? E.g. process requests and gets all its resources at once

- Eg book both the boat and the rod through the same "agent"

Roadmap



The bounded buffer producer-consumer problem

Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- … and how to help them
- Well, we failed; let's try to eliminate the deadlock's necessary conditions
  - First the cyclical wait
  - Then the no-preempt
  - Now the hold-and-wait

Consider the dining philosophers problem [Dijkstra65]

n philosophers (processes); each philosopher *P_i*, when hungry, needs : both left & right fork, in order to eat

Process P_i structure

do

get resources (i.e. entry section)

// eat

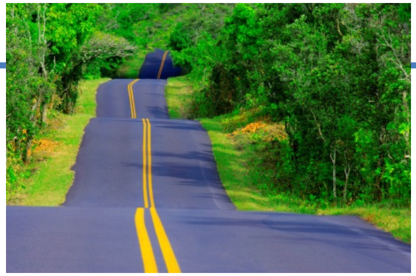
leave resources (i.e. exit section)

// think

forever

Pic: wikipedia

Roadmap



The bounded buffer producer-consumer problem

Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- **… and how to help them**
- Well, we failed; let's try to eliminate the deadlock's necessary conditions
  - First the cyclical wait
  - Then the no-preempt
  - Now the hold-and-wait

let's think together: (ie as in )

Trying to solving the dining philosophers problem: pick-left-pick-right-fork

```
Shared var f[0..n-1]: bin-semaphore
            // one for each fork; init all 1
P_i:
do
      Wait f[i];   // pick left fork;
      Wait f[(i+1) mod n];  // pick right fork
         //  Eat
      Signal f[i];  // leave left fork
      Signal f[(i+1) mod n];  leave right fork
        // Think
 forever
```

Does it solve the problem?

Recall the requirements:

1. **Mutual exclusion**: each resource is used by only one process at a time
2. **Progress**: no deadlock
3. **Fairness**: FCFS, or no starvation, or other fairness formulation

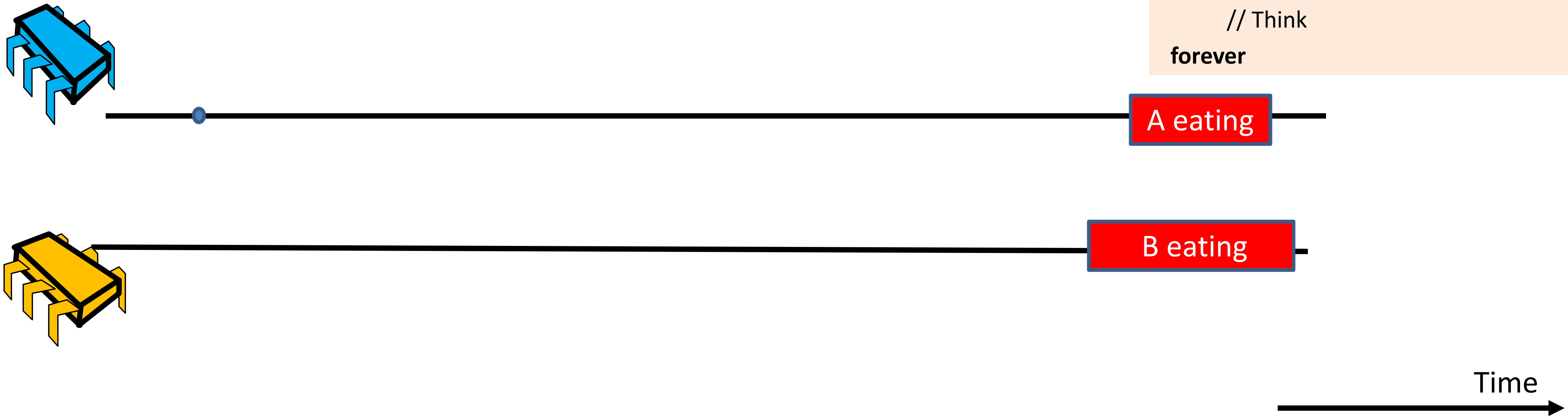
Does the ”pick-left-then-pick-right-fork” method satisfy the mutual exclusion property?

- Can it violate it? I.e. Can it happen that there is a point in time s.t. some proceses A and B concurrently access the *same resource* (i.e concurrently eat)?
- Assume it can and w.l.o.g. consider the decision step by A to eat; Can B (which *must be A’s neighbour*) decide to eat after A’s decision step and before A finishes?

Homework: fill in the details that lead to contradiction, in the figure and in text, using ”->” as we did when studying Peterson’s 2-CS algo

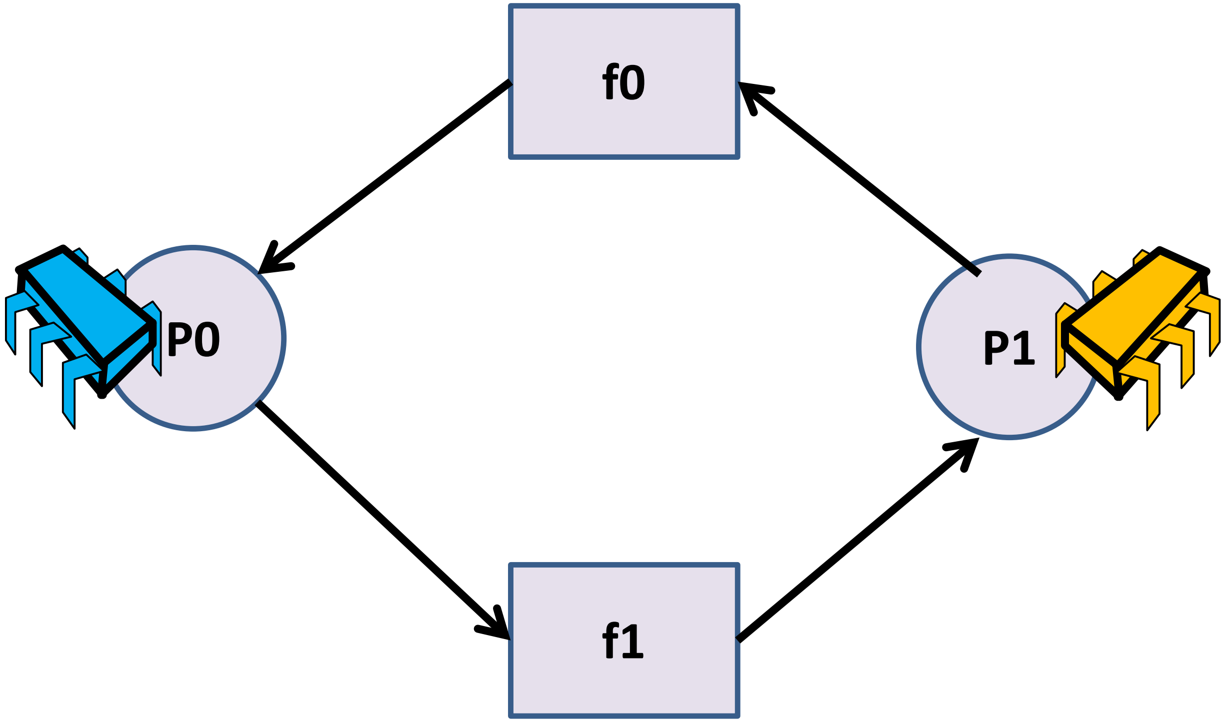
Shared **var** *f[0..n-1]: bin-semaphore*
// one for each fork; init all 1

```
P_i:
do
  Wait f[i];   // pick left fork;
  Wait f[(i+1) mod n];  // pick right fork
        //  Eat
  Signal f[i];  // Leave left fork
  Signal f[(i+1) mod n];  leave right fork
        // Think
forever
```



Does the “pick-left-pick-right-fork” method satisfy the progress property?

Can it deadlock?



Shared **var** *f[0..n-1]: bin-semaphore*
// one for each fork; init all 1

P_i:

do

Wait f[i];   // pick left fork;

Wait f[(i+1) mod n]; // pick right fork

// Eat

Signal f[i]; // Leave left fork

Signal f[(i+1) mod n]; leave right fork

// Think

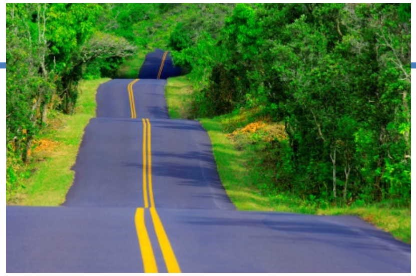
forever

Yes, example deadlock with 2 philosophers and 2 forks

Think of:

- Mutual exclusion
- Hold&wait
- No preemption
- Cyclical wait

Roadmap



The bounded buffer producer-consumer problem

Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- … and how to help them
- **Well, we failed; let's try to eliminate the deadlock's necessary conditions**
  - **First the cyclical wait**
  - Then the no-preempt
  - Now the hold-and-wait

Pick one fork at a time, & fight the circular wait:

Shared var f[0..n-1]: bin-semaphore //init all 1

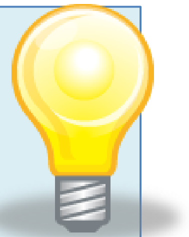
P_i: (i ≠n-1)
do
- **Wait(f[i]);**
- **Wait(f[(i+1)mod n]);**
- // Eat
- **Signal(f[(i+1)mod n])**
- **Signal(f[i])**
- // Think

forever

P_{n-1}
do
- **Wait(f[(i+1)mod n])** //ie wait(f[0])
- **Wait(f[i])** //ie wait(f[n-1])
- // Eat
- **Signal(f[i])**
- **Signal(f[(i+1)mod n])**
- // Think

forever

Idea:
- use ordering of resources
- Proc's request their needed resources in **increasing order**



Does it solve the problem?

Does it fight the circular wait?

Key idea: Follow the waiting chains (directed paths in the RA graph): always the requested resource with max-id is the end of it, thus preventing circle

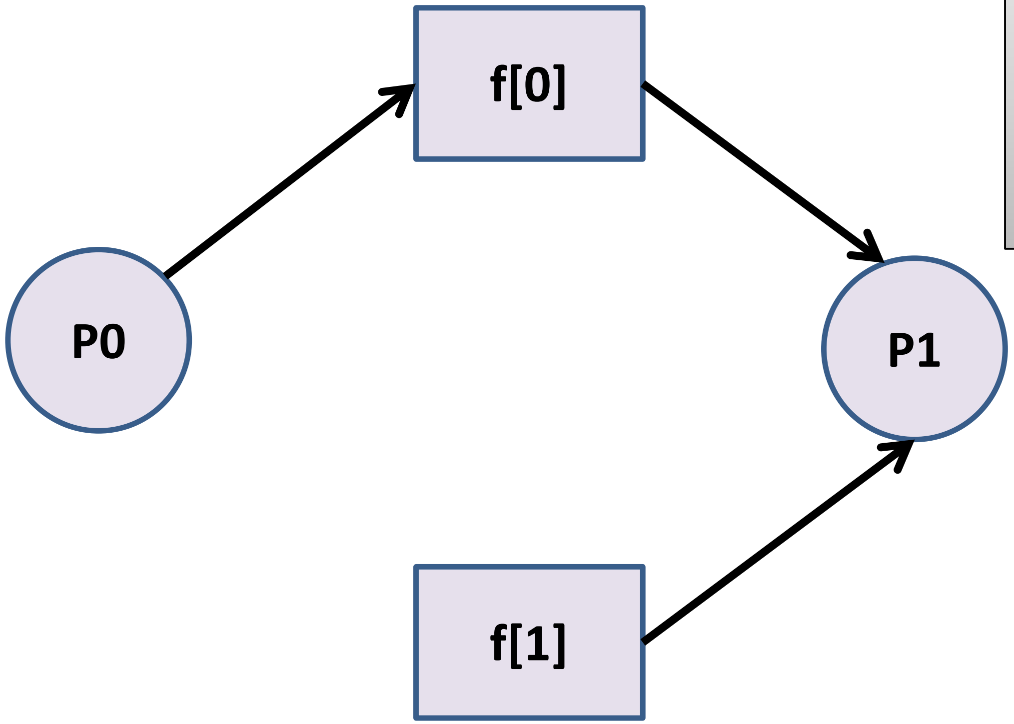
Correctness argument:
How is circular wait prevented with the “request-in-resource-order” algo?

Start simple, consider 2 processes (P0, P1)
Assume, towards a contradicton that deadlock can happen,
i.e. there exists a circle…

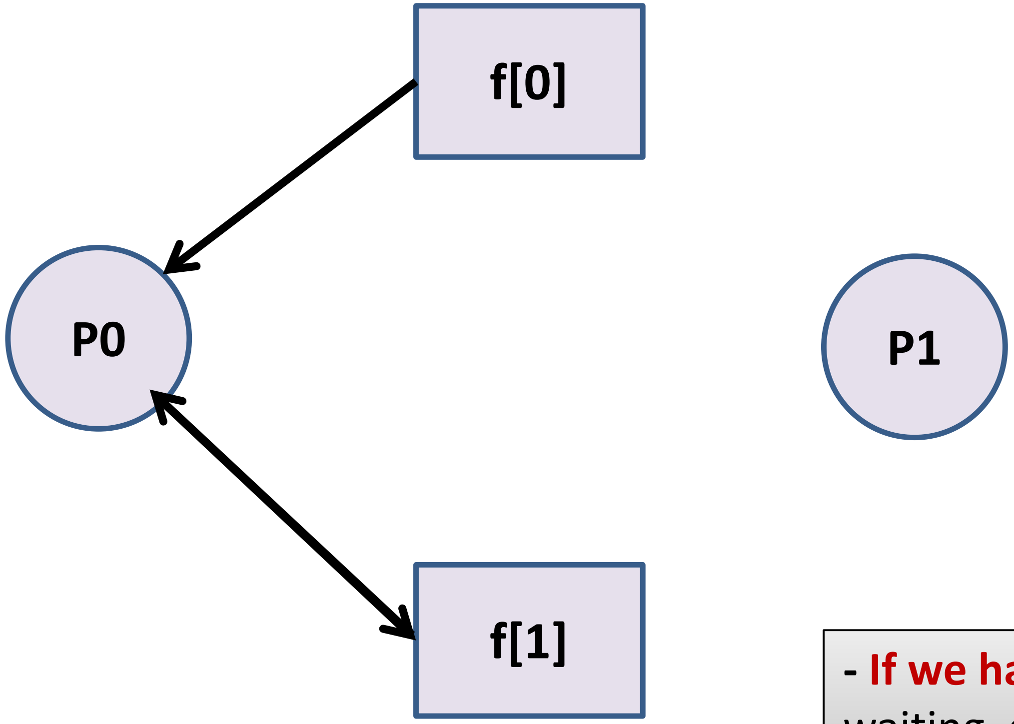
Without loss of generality (wlog), consider P0:

Case 1: if P0 waits for f[0]:

- P1 must have it, hence it can get f[1] (i.e. **max-id resource**) and eat; i.e. no circle (i.e. contradiction of the assumption that the wrong thing can happen, in case 1)



Correctness argument: How is circular wait prevented with the "request-in-resource-order" algo? (cont)



Case2: if P0 waits for f[1]:

- P0 must have f[0], hence f[1] (i.e. **max-id resource**) is available and P0 can eat; i.e. again no circle (i.e. contradiction of the assumption that the wrong thing can happen, in case 2)

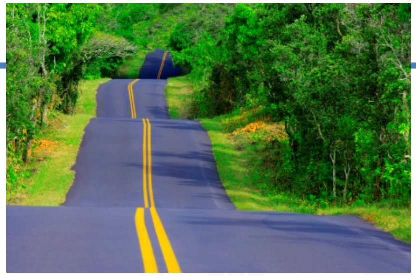
- If we have more processes and resources, follow the waiting chain: **always the max-id resource is the end of the waiting chain,** thus preventing the circle, QED

Fairness property of the “request-in-resource-order” algo?

It depends directly on the fairness guarantees of the underlying semaphore’s implementation.

@home: Show in timelines that:
if the semaphores do not guarantee fairness, then the “request-in-resource-order” algo can be unfair e.g given 2 threads A and B, *A can by-pass B many times* while B is not able to go beyond the wait of their common-fork’s semaphore. You may consider a simple system with just 2 philosophers.

Roadmap



The bounded buffer producer-consumer problem

Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- … and how to help them
- Well, we failed; let's try to eliminate the deadlock's necessary conditions
  - First the cyclical wait
  - **Then the no-preempt**
  - Now the hold-and-wait

Fight the no-preemption

```
shared var f[0..n-1]: of type fork_structure { // one for each fork
      s: bin-semaphore  //init 1
      available: boolean //init true
}

P_i:
local var holding_both_forks: boolean;
repeat
      while (not holding_both_forks){
            lock(f[i])
            if !trylock(f[(i+1)modn]) then release(f[i])
            else holding_both_forks := true }
      // Eat
      release(f[i])
      release(f[(i+1)modn])
      holding_both_forks := false
      // Think
forever
```

```
trylock(fork: fork_structure):
      wait(fork.s)
      if fork.available then { fork.available := false ;
            ret:= true;
      }
      else ret:= false;
      signal(fork.s)
      return(ret)

lock(fork : fork_structure):
      repeat
      until (trylock(fork))

release(fork : fork_structure):
      wait(fork.s)
      fork.available := true
      signal(fork.s)
```

Properties?

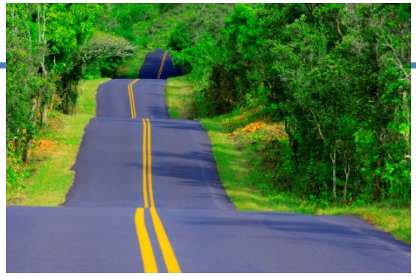
Idea: when the second resource is not available, release the first one and retry

Fight the no-preemption algo of the prev. slide: properties:

- Mutual exclusion: ok
- Progress: no deadlock …
- Fairness: a process can starve…

- Homework: put down the arguments for the above using our discussion and the methodology that we apply + (for the no-deadlock property) use the implication of Coffman’s thm

Roadmap



- The bounded buffer producer-consumer problem

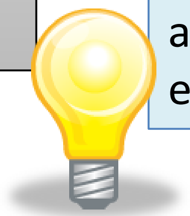
- Resource allocation (dining philosophers is such a problem)
  - Intro
  - We elaborate on deadlocks
- What is the problem with the Dining philosophers…
  - … and how to help them
  - Well, we failed; let's try to eliminate the deadlock's necessary conditions
    - First the cyclical wait
    - Then the no-preempt
    - **Now the hold-and-wait**

Fighting the hold and wait

```
shared var semaphore S[0 .. n-1] // init all 0
shared var semaphore mutex // init 1
shared var state[0 .. n-1] in {HUNGRY, THINKING,EATING}
Pi:
do
        // think
        enterCS(i) // ie get both forks
        // eat
        exitCS(i) // ie leave bothforks
forever
```

```
enterCS(i)
    wait(mutex)
    state(i) := HUNGRY
    help(i)
    signal(mutex)
    wait(S[i])
```

```
exitCS(i)
    wait(mutex)
    state(i) := THINKING
    help((i-1) mod n)
    help((i+1) mod n)
    signal(mutex)
```



Idea: ”eat” is mutually exclusive (ie CS) among each P_i and its neighbours, hence:
apply a CS algo in each neighbourhood, instead of for each fork (i.e. as if philosopher picks both forks at once)

```
help(k)
    if state[k] ==HUNGRY &&  state[(k-1) mod n] != EATING && state[(k+1) mod n] != EATING
    then {state(k) := EATING ; signal(S[k]) }
```

Properties?

Fight the no-hold-and-wait algo of the prev slide: properties:

- Mutual exclusion: ok
- Progress: no deadlock
- Fairness: a process can starve

- Homework: put down the arguments for the above using our discussion and the methodology that we apply + (for the no-deadlock property) use the implication of Coffman's thm

Link to solutions to the problem with fairness guarantees as well

Eugene Styer and Gary L. Peterson. 1988. Improved algorithms for distributed resource allocation. In Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC '88). Association for Computing Machinery, New York, NY, USA, 105–116. DOI: https://doi.org/10.1145/62546.62567

(direct link to pdf https://dl.acm.org/doi/pdf/10.1145/62546.62567?casa_token=4uO24jxkwEAAAAAA:jJlAILeISZe5Uu2ERv6O-dTq_0LbSmRpv0beOZ_3vDi50otRS-_HqB30GoWDib1zVQ9jjrhx4w0)

Roadmap

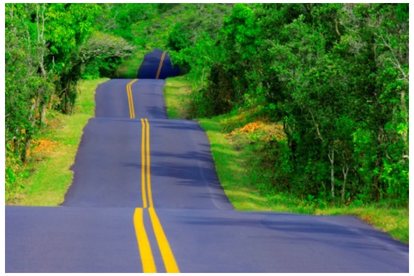
The bounded buffer producer-consumer problem

Resource allocation (dining philosophers is such a problem)

- Intro
- We elaborate on deadlocks

What is the problem with the Dining philosophers…

- … and how to help them
- Well, we failed; let's try to eliminate the deadlock's necessary conditions
  - First the cyclical wait
  - Then the no-preempt
  - Now the hold-and-wait

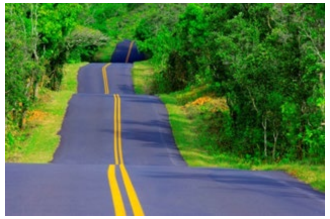


Summary

- Discussed the concept of building synch-objects from other synch objects
- Common synchronization problems: bounded buffer, dining philosophers
- Resource-allocation&deadlocks
  - Deadlock: 4 conditions necessary
  - Fighting deadlock: prevent (i.e. attack deadlock's necessary conditions)
- We saw several synchronization methods and examples
  - incl. helping, trylock implementation
- Shortly: narrow bridge & lab

Next lecture: more in-depth n-process mutual-exclusion and tools/methods/ properties

- Lamport's bakery algo + Turing award topic
- Readers/writers problems and a touch on lock-free synchronization
- One more way to deal with deadlocks (avoid, using with an arbitrator: Bankers algo by Dijkstra)



Reading instructions (on all the synchronization topics we discuss)

Modern OS by Tanenbaum-Bos:

- careful study of sections 2.3.1-2.3.6, 2.5.2
- Complement (Bakery alg.) through http://web.cs.iastate.edu/~chaudhur/cs611/Sp09/notes/lec03.pdf
- Quicker reading, for awareness, of sections 2.3.7-2.3.10

Alt. from OS Concepts: Silberschatz-et-al: Sections 6.1-6.7, 6.9

-Matching review questions at e.g.
http://codex.cs.yale.edu/avi/os-book/OS9/review-dir/index.html

Optional reading, other sources:

1. Leslie Lamport (recipient of ACM Turing award 2013). Turing lecture: The computer science of concurrency (with special mention to the Bakery algo)

   Commun. ACM 58, 6 (May 2015), 71-76. DOI= http://dx.doi.org/10.1145/2771951

2. Large variety of synch methods: how to think/decide? Cf also eg:

   A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems; D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, P Tsigas, 27th IEEE International Parallel & Distributed Processing Symposium, IPDPS 2013 http://www.computer.org/csdl/proceedings/ipdps/2013/4971/00/4971b309-abs.html

3.M. Herlihy&Shavit,"The art of Multiprogramming, By Herlihy & Shavit) (http://cs.brown.edu/courses/cs176/lectures.shtml)

4. P. Fatourou: Spin Locks and Contention https://www.csd.uoc.gr/~hy586/material/lectures/cs586-Section3.pdf

Reading instructions (include all deadlock-related parts of our discussions):

- **From Modern OS by Tanenbaum et-al:**

  Careful study 2.5.1, 6.1-6.2, 6.5-6.6, 6.7.3-6.7.4; quick reading 6.2-6.3
- **Alt. from OS Concepts by Silberschatz et-al:**

  Careful study 7.1-7.5, 7.8; quick reading 7.6-7.7
- In addition to the above:
  - Practice on the dining philosopher solutions described in the notes; understand why they work, try to argue about correctness as we did for Peterson's algo
  - Practice on homework hintsin the notes and on the exercises that will be discussed in class (several of them have been basis for exam questions)

-Matching review questions at

http://codex.cs.yale.edu/avi/os-book/OS9/review-dir/index.html