Course Operating Systems

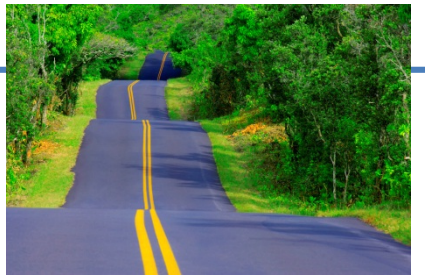
Lecture 7:
a. Landmarks on Synchronization
b. RA with OS as arbitrator (avoid/recover from deadlocks)

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

Roadmap



Understanding Synchronization better: some landmark methods/problems

- **Lamport's Bakery algorithm**

Understanding synchronization better

Critical Section for n threads

One idea: Before entering its critical section, each thread gets a number. Holder of the smallest number enters the critical section.

Lamport's Bakery Algorithm

Critical section for n threads using R/W variables

Idea: Implement "nummerlappar" using read/write variables only:

- De-centralized numbering scheme:
  - Each thread: read the others' numbers and choose the max+1 as your "nummerlapp"
  - Wait for the smaller numbers and then enter CS

That simple?! Can it work?

Note: the decentralized scheme may generate numbers in non-decreasing order of enumeration; i.e., 1,2,3,3,3,3,4,5

If threads Pi and Pj choose the same number:

if i <j, then Pi goes first; else Pj goes first.

I.e. we need to use thread_id's to break ties

Lamport's Bakery Algorithm

pseudocode thread i

```
Shared var choosing: array [0..n – 1] of boolean (init fasle);
           number: array [0..n – 1] of integer (init 0);
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] := false;
    for j := 0 to n – 1 do begin
        while choosing[j] do [nothing]; //spin
        while number[j] ≠ 0 and (number[j],j) < (number[i], i) do [nothing]; //spin
    end;
        critical section
    number[i] := 0;
        remainder section
until false;
```

*This is a more **decentralized** method than e.g. Peterson's: no variable is "writ-able" by all threads*

Why does it satisfy the 3 conditions:

Mutex (no 2 threads A and B in CS concurrectly): Consider the time between A's decision step and A's entry to CS; A decided to move because:

- B had higher number: when B checks , it will wait for A since A has smaller number
- or B was not interested; when B gets interested, it will choose a number > A's number, hence it will wait

Progress: the thread with the smaller number can proceed

Fairness: If A waits for B and B exits and wants to enter CS again, if A still waits, B will choose a number > A's, number, hence B cannot bypass A

To elaborate/think: homework

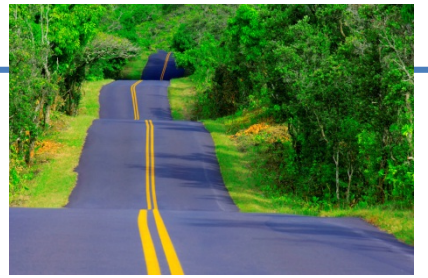
- Explain why Bakery algorithm satisfies the 3 conditions for critical section problem, following the diagrams and the description as we did for Peterson's algo

- Bakery algorithm idea not tied with atomic R/W variables, can work with weaker primitives:
  - Read/watch Lamport's Turing award talk (links @ reading instructions)
    - Bakery algo + concurrent readers/writers gave rise to the research for lock-free synchronization

Practice further on synchronization constructs, e.g write algorithms for implementing:

- counting semaphores from binary ones
- semaphores using mutual exclusion solutions e.g. Peterson's, Lamport's methods, the TAS or CAS methods
- *a ticket-based (a la Bakery) method using TAS, CAS, …*

Roadmap



Understanding Synchronization better: some milestone steps /landmark methods

- Lamport's Bakery algorithm
- **Readers-Writers problem**

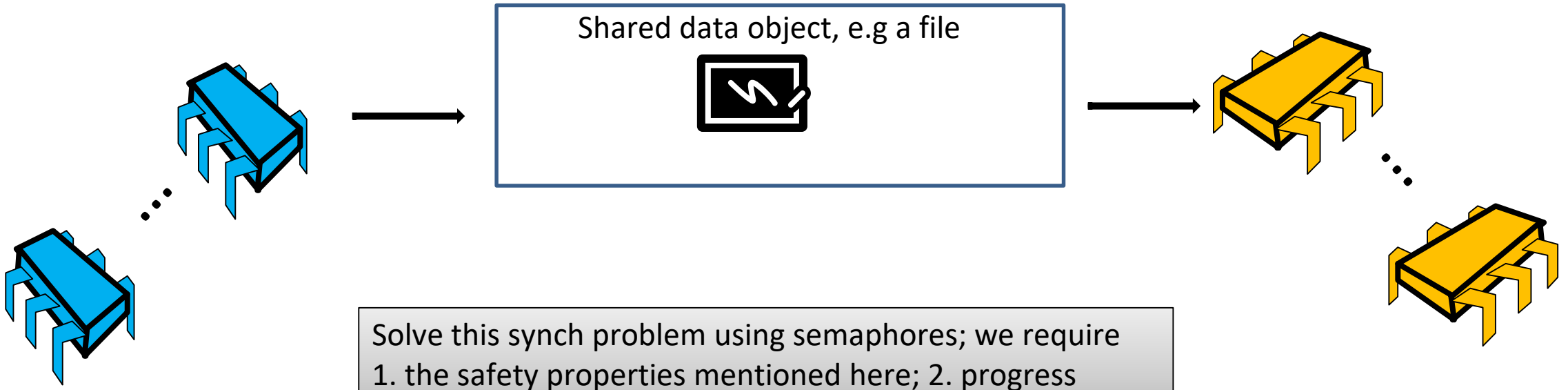
Readers-writers problem

Writers write;
must wait if some writer or reader
is accessing the shared data object

(ie *only one writer is allowed to write at a time*)

Readers read;
must wait if some writer
is accessing the shared data object

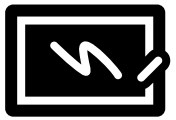
(ie *multiple readers are allowed to read at a time but not concurrently with any writer*)



Solve this synch problem using semaphores; we require
1. the safety properties mentioned here; 2. progress

Readers-writers problem

Shared data object, e.g a file

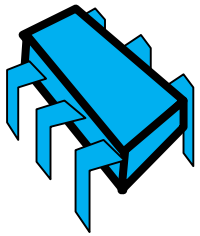


Writers write;
must wait if some writer or reader
is accessing the shared data object

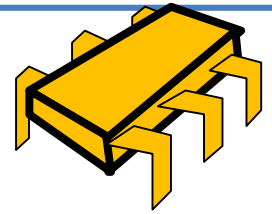
(ie only one writer is allowed to write at a time)

Readers read;
must wait if some writer
is accessing the shared data object

(ie multiple readers are allowed to read at a time
but not when some writer writes)



Writer



Reader

if I am first in a batch of readers to check

wait until no one accesses the object

// WRITE

no longer / no one accessing the object

wait until no one accesses the object
(else others read, hence go on)

// READ

inform: no longer accessing the object

(… + if I am last in a batch of readers who read,

tell that no one accesses the object …)

A solution for readers-writers

shared var:
noone_accesses, protect_check: binary semaphore; // initially 1
rc: int ; //active readers counter, init 0

Writer

Repeat
wait(*noone_accesses*);

//WRITE

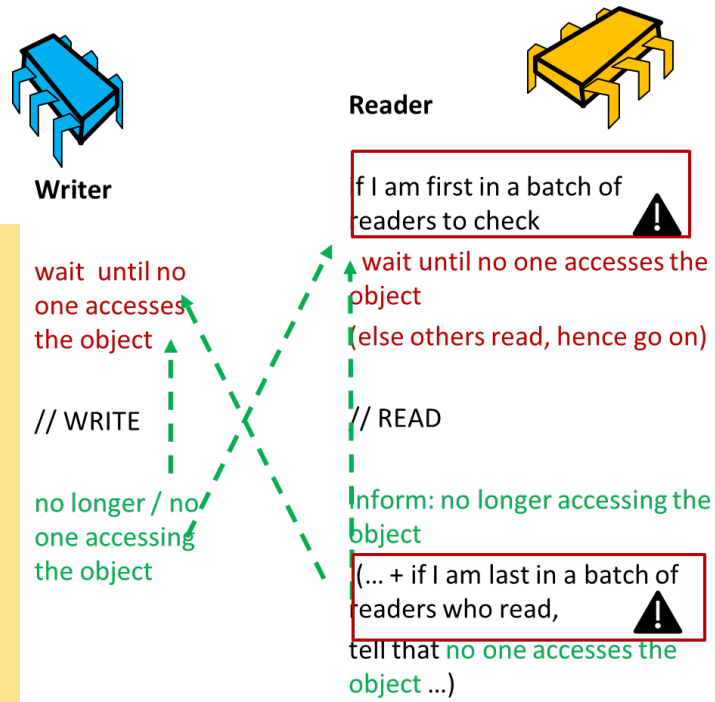
signal(*noone_accesses*)
forever

Reader

repeat
wait(*protect_check*); // CS to change and check rc variable
if *rc++* == 1 **then** wait(*noone_accesses*) **fi**
// "first" reader: block writers or wait if some of them writes
signal(*protect_check*);

// READ

wait(*protect_check*); // CS to change and check rc
if *rc--* == 0 **then** signal(*noone_accesses*) **fi**
// "last" reader: signals
signal(*protect_check*)
forever

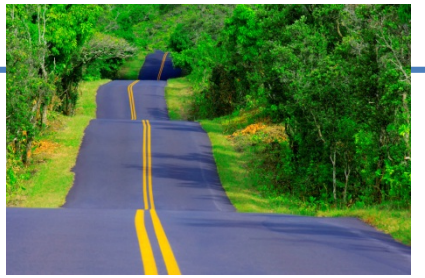


Homework:
1. argue about correctness wrt requirements (safety, progress);
2. What about fairness?
Show that the solution enforces that readers have "priority"…

The Readers/Writers Problem …

… paved the way to research on *lock/wait-free synchronization -- concurrent reading while writing (see also ptrs in Reading Instructions)*

Roadmap



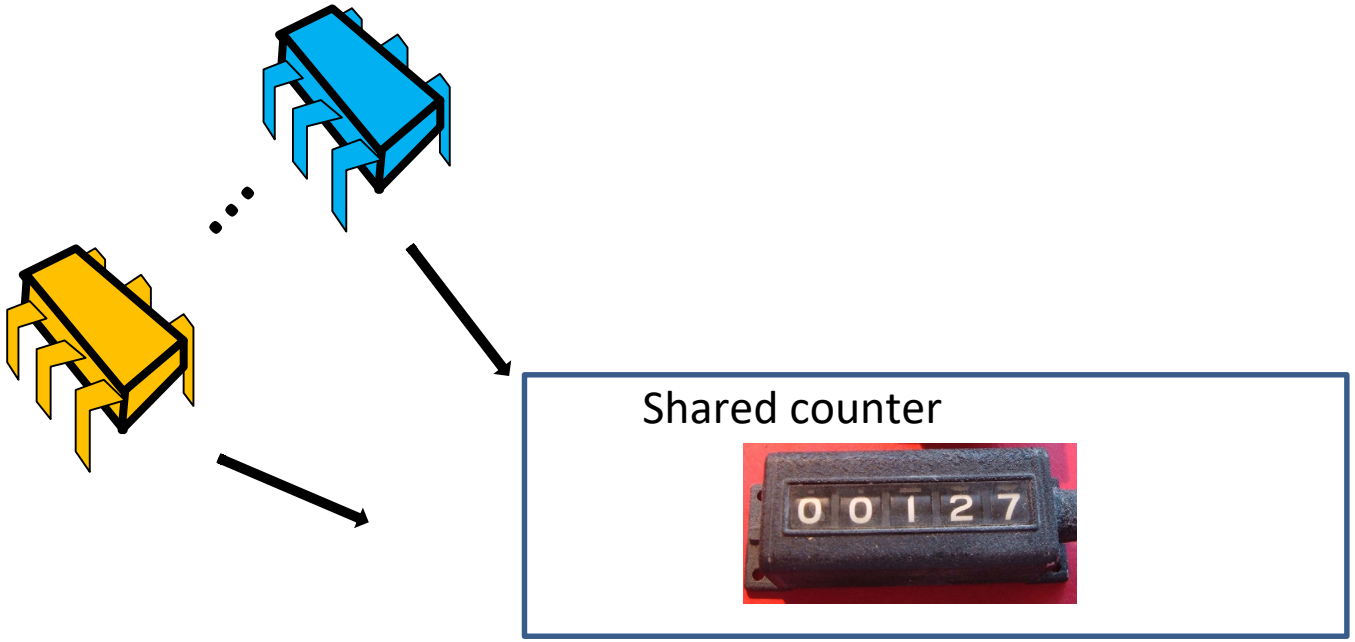
Understanding Synchronization better: some milestone steps /landmark methods

- Lamport's Bakery algorithm
- Readers-Writers problem
- **A touch of lock-free synchronization**

Shared counter problem

Increment

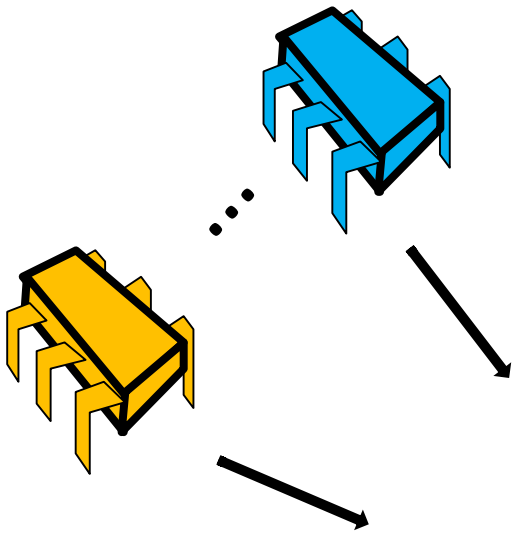
must increase the value of the counter by one



Shared counter problem: lock—based method (Critical Section)

Increment

must increase the value of the counter by one



Standard CS-based method
shared var: mutex: binary semaphore // init 1
A: int // holds counter's value, init eg 0

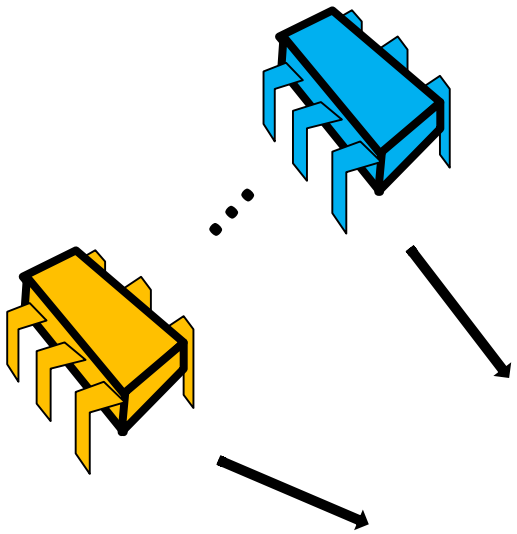
Increment(A)
wait(mutex)
  tmp := A
  tmp++
  A := tmp
signal(mutex)

Shared counter

Shared counter problem – can we do without locking?

Increment

must increase the value of the counter by one



Standard CS-based method
shared var: mutex: binary semaphore
// init 1
A: int // holds counter value, init eg 0

Increment(A)
wait(mutex)
  tmp := A
  tmp++
  A := tmp
signal(mutex)

Shared counter

Lock-free (no CS-based) method
shared var:
A: int // holds counter's value

Increment(A)
repeat
  tmp := A
until CAS(&A, tmp, tmp+1)

Recall:
CompareAndSwap (aka CAS) Instruction
Definition:

```
int CompareAndSwap(int *V, int exp_v, int
   new_v) {
  boolean effect := false ;
  if (*V == exp_v) then
     *V := new_v; effect := true; fi
  return effect}  // Executed atomically in HW
```

Lock-free synchronization

Goal:

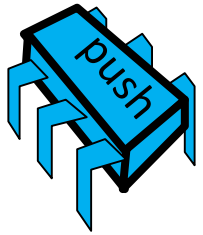
- allow more parallelization AND
- achieve the same consistency ***as if*** CS *of different threads are not overlapping* in time
- Possible through fine-grain synchronization, allowing a fail-retry-loop, indicated here using this symbol :

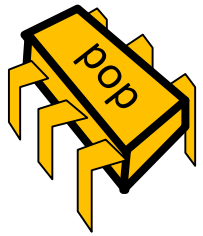
Another example:

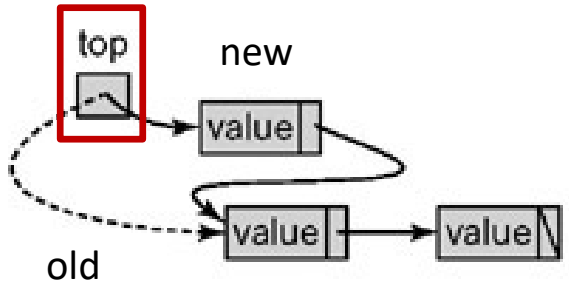
Non-blocking stack [Treiber '86]

```
proc push(new)
do
  old = top
  new.next = old
while not CAS(top, old, new)
end

proc pop
do
  old = top
  return null if old == null
  new = old.next
while not CAS(top, old, new)
return old
end
```







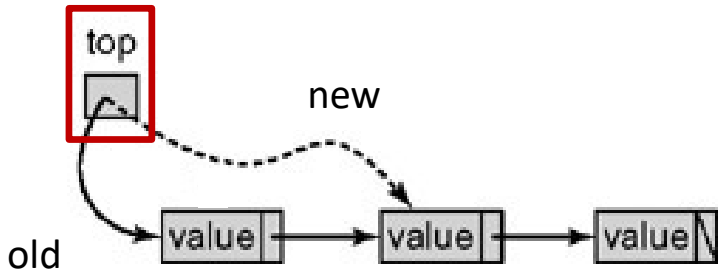


Fig based on related figure from
The Art of Multiprocessor Programming,
by Herlihy & Shavit

Whys and hows?

- Why lock-free synchronization?
  - Efficiency, free from: convoy effects, deadlocks, priority inversion
  - Better utilization of parallel HW
    - cf TBB, Boost libraries

How to argue about correctness?

- Safety condition: Linearizability [Herlihy&Wing 81]
- Progress: no livelock
- Fairness: starvation possible; tougher here to argue about fairness; ongoing research
  - Work by our group shows that lock-free methods balance throughput/fairness trade-offs

  [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, http://www.computer.org/csdl/proceedings/ipdps/2013/4971/00/4971b309-abs.html]

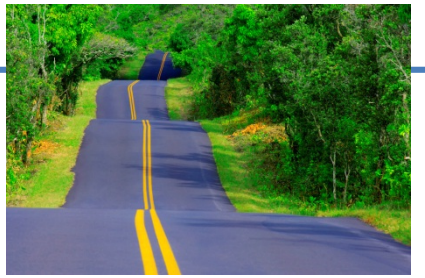
To elaborate/think:

- Bakery algorithm idea not tied with atomic R/W variables, can work with even weaker primitives:
  - Read/watch Lamport's Turing award talk
    - Commun. ACM 58, 6 (May 2015), 71-76. DOI= http://dx.doi.org/10.1145/2771951
    - Bakery algo + concurrent readers/writers and related work gave rise to the research for lock-free synchronization

- Remember "helping" from dining philosophers "prevent the hold&wait" method?
  - "Helping" is used a lot in lock-free methods

- *Large variety of synch methods: how to think/decide? Cf also eg:*
  - M. Herlihy&Shavit, The Art of Multiprocessor Programming (ebook),
    - Lectures: http://cs.brown.edu/courses/cs176/lectures.shtml
  - TBB, Boost libraries
  - A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems; D. Cederman et-al., 27th IEEE International Parallel & Distributed Processing Symposium, http://www.computer.org/csdl/proceedings/ipdps/2013/4971/00/4971b309-abs.html

Roadmap



Understanding Synchronization better: some milestone steps /landmark methods

- Lamport's Bakery algorithm
- Readers-Writers problem
- A touch of lock-free synchronization

Encore about resource allocation and deadlocks:

(last lecture we discussed deadlock prevention, i.e. methods for how threads request&acquire resources so that deadlock cannot occur)

Now: using the OS as arbitrator … Deadlock avoidance: Dijkstra's Banker algo

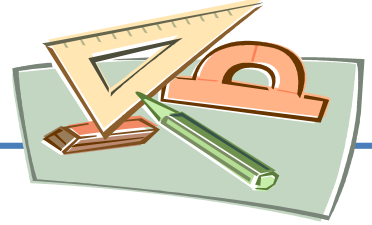
Deadlock avoidance using the OS as arbitrator

Resource request & allocation is managed by the OS

Deadlock *avoidance*:

- deadlock *might* possible if resources are granted arbitrarily,
- but OS uses extra info to grant requests and schedule processes s.t. it avoids deadlock
- **Banker's algorithm[Dijkstra]**

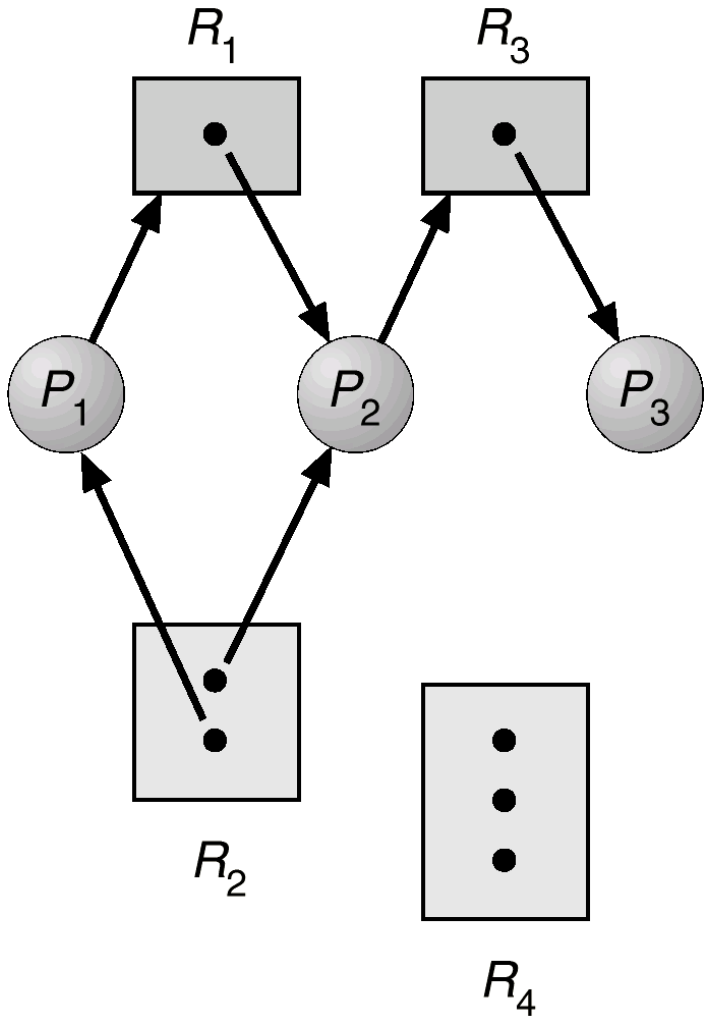
Graphs are useful tools: System Model



- Resource types $R_1, R_2, \ldots, R_m$
  - *e.g. CPU, memory space, I/O devices, files*
  - each resource type R_i has W_i instances.

Resource-Allocation Bipartite Graph G(V,E)

- nodes:
  - $P = \{P_1, P_2, \ldots, P_n\}$ the set of processes
  - $R = \{R_1, R_2, \ldots, R_m\}$ the set of resources types
- edges:
  - request edge: $P_i \rightarrow R_j$
  - assignment edge: $R_j \rightarrow P_i$

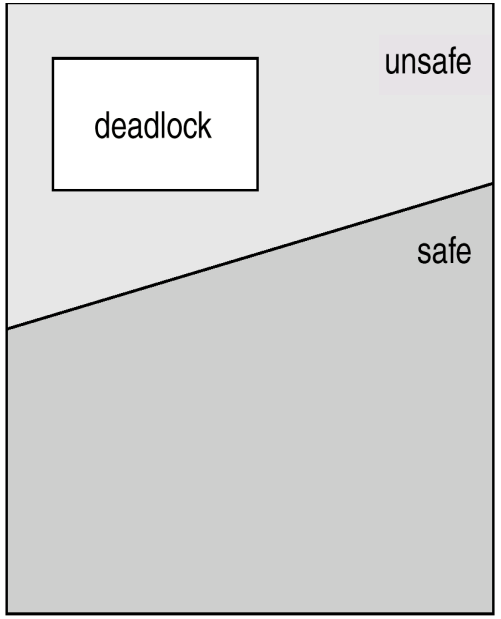


Resource Allocation with Deadlock Avoidance

Resource request & allocation is managed by the OS

Requires *a priori information* available.

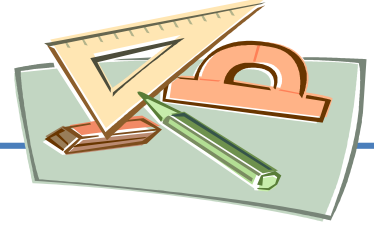
- e.g.: each *process* declares *maximum number* of resources of each type that it *may need* (e.g memory/disk pages).

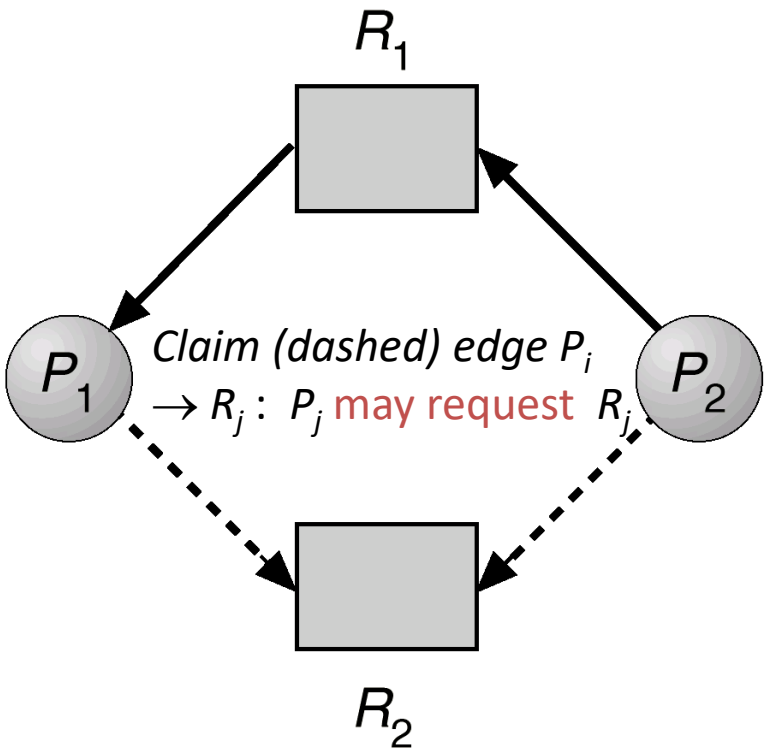


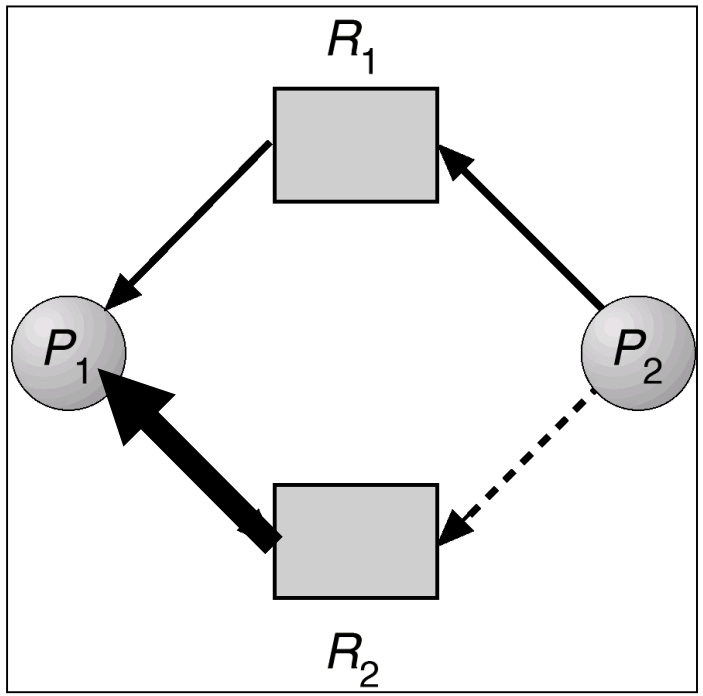
Deadlock-avoidance algo, run by OS:

- examines the resource-allocation state…
  - *Available, allocated resources*
  - *maximum possible demands* of the processes.
- …to ensure there is *no potential deadlock*:
  - unsafe state ⇒ deadlock might occur (i.e. later, if all procs request their maximun and no-one can be granted)
- *Avoidance* = ensure that *system will not enter an unsafe* state, by *suspending processes with risky requests*, until enough resources are freed.

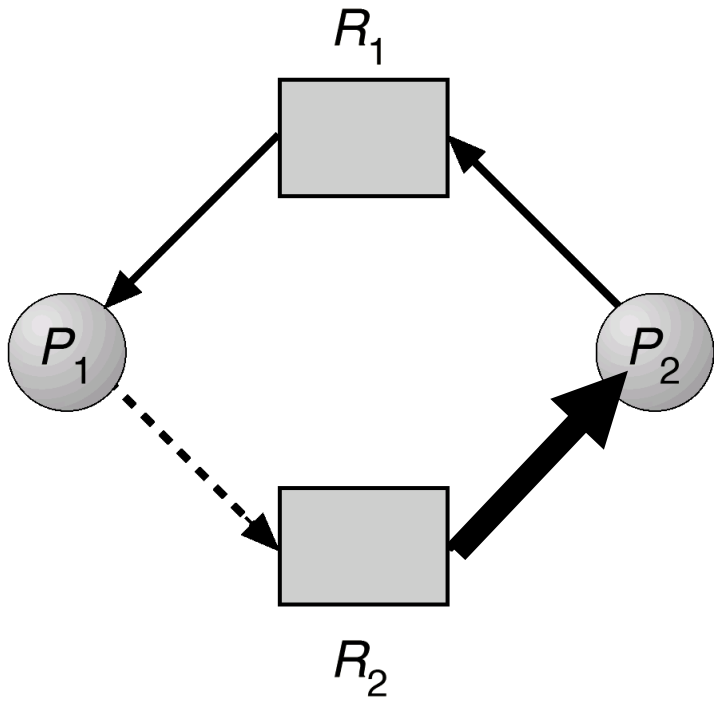
Enhanced Resource-Allocation Graph For Deadlock Avoidance:







Safe State



Example Safe State

Example Unsafe State:
ie granting the P2-R2 request should not be made until P1 finishes

Banker's Algorithm for Resource Allocation with Deadlock Avoidance [Dijkstra]

Max [i,j] = k:
P_i may request max k instances of R_j.

Allocation[i,j] = k:
Pi holds k instances of Rj

Available [j] = k :

k instances of Rj are still available.

Avoidance = ensure that *system will not enter an unsafe* state.

Idea:

If *potentially satisfying a request* can result in an *unsafe state* // *i.e. bank will have not enough to let its customers finish and return their loans in case someone requests its max needs*

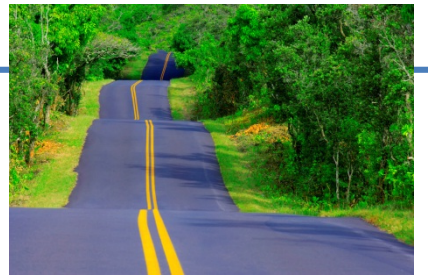
then the *requesting process is* ***suspended*** // *temporarily frozen, not scheduled*

until *enough resources are free-ed* // *by processes that will terminate in the meanwhile*

How to do the safety check efficiently?

Banker's algo gives criterion that can be checked in linear time using the Max, Allocation, Available matrices (check the algo data structures @book and @extra-slides, as homework)

Roadmap



Understanding Synchronization better: some milestone steps /landmark methods

- Lamport's Bakery algorithm
- Readers-Writers problem
- A touch of lock-free synchronization

Encore about resource allocation and deadlocks:

(last lecture we discussed deadlock prevention, i.e. methods for how threads request&acquire resources so that deadlock cannot occur)

- using the OS as arbitrator… **Deadlock avoidance: Dijkstra's Banker algo**
- **Deadlock detection and recovery**

Deadlock detection and recovery

- If OS grants requests without checking safety upon every request
- It can allow a deadlock state and when detected (eg through detection of cyclical waits), *recover*

Recovery from Deadlock

(1) Process Termination: Abort all or some deadlocked processes until deadlock is eliminated.

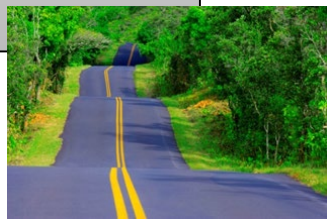
(2) Resource Preemption: Select victim and rollback – return to some safe state, restart process from that state

Must decide on selection criteria (cost, starvation risks, …)

Recovery is pretty expensive as a method ….

Summary

- Landmarks on synchronization problems: Bakery algo, Readers-writers, a glimpse on lock-free synchronization
- Resource-allocation&deadlocks
  - Avoiding or recovering from deadlock with OS as arbitrator
- **We saw a lot of synchronization methods and examples**
  - **and a lot of homework tips**
- **Lab 2-3: holistic training on scheduling & synchronization together**



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 et-al 27th IPDPS. http://www.computer.org/csdl/proceedings/ipdps/2013/4971/00/4971b309-abs.html

3.M. Herlihy&Shavit, The Art of Multiprocessor Programming,

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

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

5. Efficient Data Streaming Multiway Aggregation through Concurrent Algorithmic Designs and New Abstract Data Types. ACM Trans. Parallel Comput. 4, 2017 https://doi.org/10.1145/3131272

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)

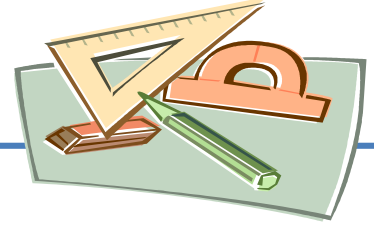
-Matching review questions at
http://codex.cs.yale.edu/avi/os-book/OS9/review-dir/index.html

EXTRA SLIDES/NOTES

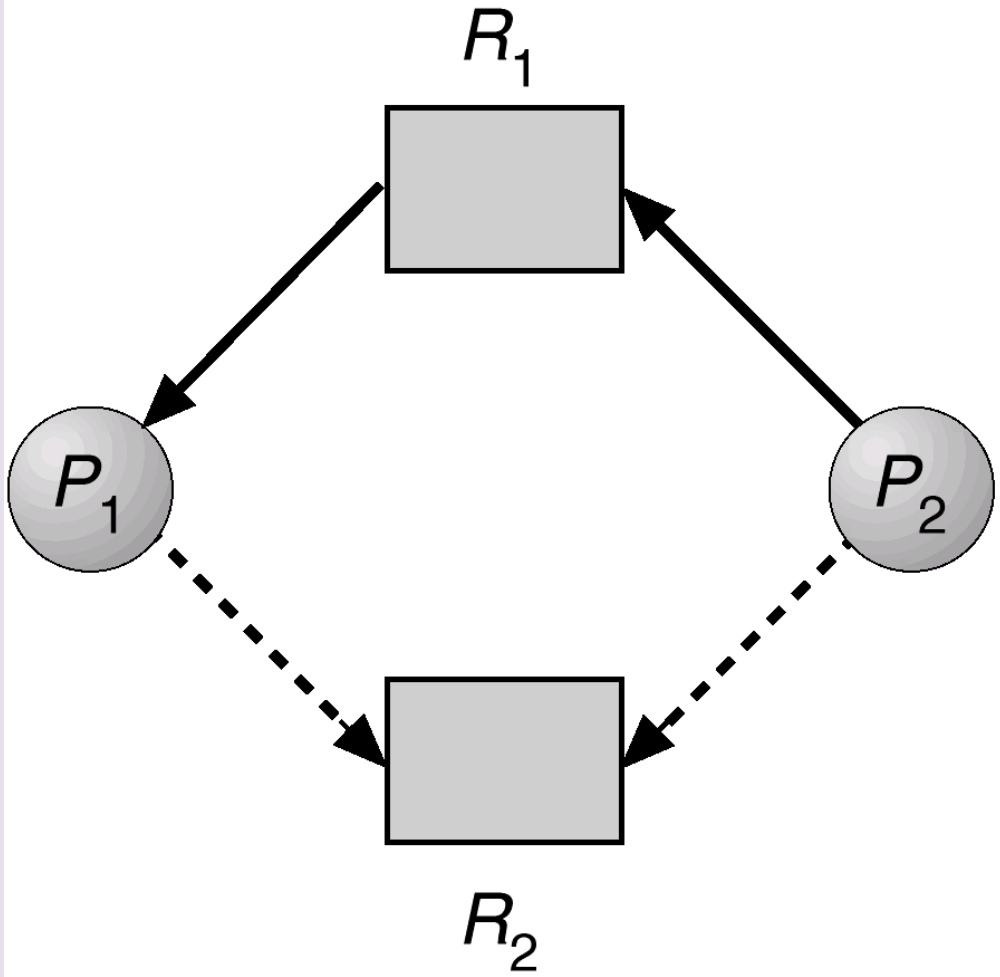
- Part 1: Complement: Banker's algo for deadlock avoidance: data structures and safety check

- Part 2: Use of Banker's algo idea to do deadlock detection

Enhanced Resource-Allocation Graph For Deadlock Avoidance:



- *Claim (dashed) edge* $P_i \rightarrow R_j$: P_j may request R_j
- Claim edge converts to request edge when the process requests the resource.
- When the resource is released by the process, assignment edge reconverts to a claim edge.

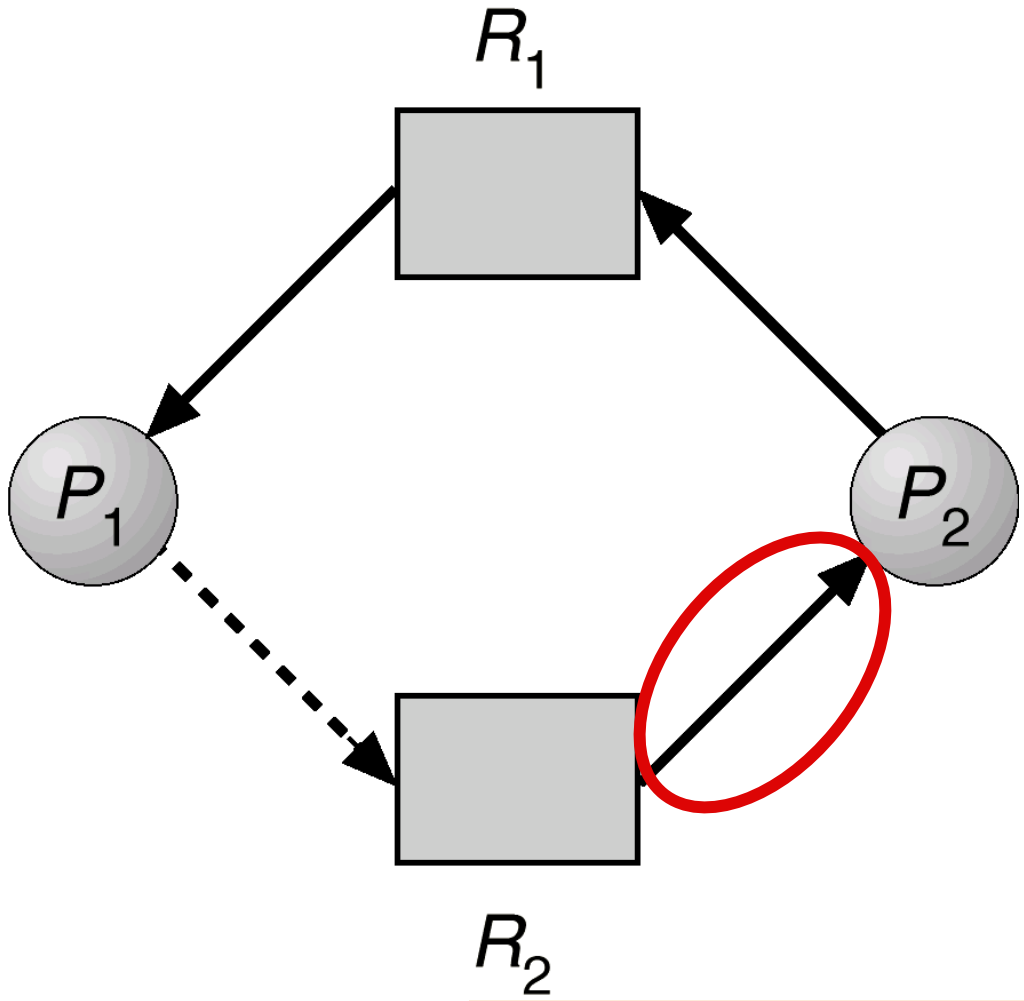


Example Safe State

Example Resource-Allocation Graph For Deadlock Avoidance:

Q: What if P2 makes that request and the system allocates it?

A: Resulting is unsafe (not deadlocked; it will be deadlocked if P1 also makes the other request) To avoid deadlock, better postpone that allocation



Example UNsafe State

Banker's Algorithm for Resource Allocation with Deadlock Avoidance [Dijkstra]

Max [i,j] = k:
P_i may request max k instances of R_j.

Allocation[i,j] = k:
Pi holds k instances of Rj

Need [i,j] =
Max[i,j] – Allocation[i,j]:
potential max request by Pi for Rj

Available [j] = k :
k instances of
Rj are available.

RECALL: (1) *Avoidance* = ensure that *system will not enter an unsafe* state.)

(2) Idea:

If *satisfying a request* result in an *unsafe state*,

then the *requesting process is **suspended***

until enough resources are free-ed by processes that will terminate in the meanwhile.

Safety checking: More about Safe State

safe state = there exists a *safe sequence* $<P_1, P_2, \ldots, P_n>$ of terminating all processes:

for each P_i, the max requests that it can still make can be granted by available resources + those held by $P_1, P_2, \ldots, P_{i-1}$

i.e. the system (OS, imaginary banker) could safely allocate as follows:

- if P_i 's resource needs are not immediately available, then it can
  - wait until all $P_1, P_2, \ldots, P_{i-1}$ have finished
  - obtain needed resources, execute, release resources, terminate.
- then the next process can obtain its needed resources, and so on.

Banker's algorithm: Resource Allocation

For each new $Request_i$ do $Request_i[j] = k$:

// P_i wants k instances of R_j.
Check consequence
if request would be granted //

// **State S**: tentative changes, to check if safe i.e what might happen //

old resource-allocation state :=current resource-allocation state;

$Available := Available - Request_i;$

$Allocation_i := Allocation_i + Request_i;$

$Need_i := Need_i - Request_{i;;}$

If safety-check (S) OK ⇒ *the resources are allocated to* P_i.

Else (unsafe) ⇒

P_i *must wait (be blocked/suspended) and*

old resource-allocation state is restored;

Banker's Algorithm: safety check

Work and *Finish*: auxiliary vectors of length *m* and *n*, respectively.

- Init: *Work* := *Available*

  Finish [*i*] = *false* for *i* = 1,2, …, *n*.

//"work" has enough for proc i to get the max it might need//

- While there exists *i* such that both (a) *Finish* [*i*] = *false* (b) $Need_i \leq Work$ do

  Work := *Work* + $Allocation_i$

  Finish[*i*] := *true*

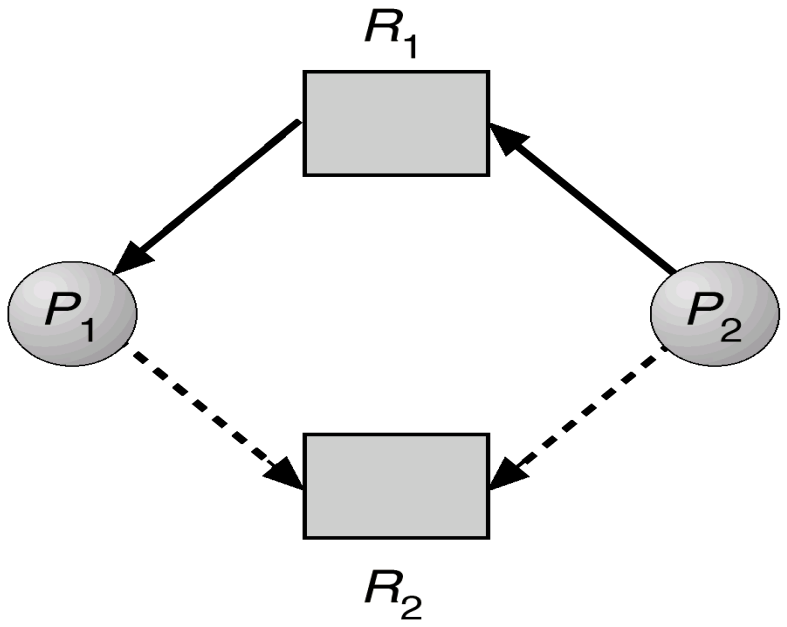
// "simulate" Pi's execution & termination //

- If *Finish* [*i*] = true for all *i*, then the state S in question (cf prev slide) is a safe one; else state is unsafe

Very simple example execution of Bankers Algo (snapshot 1)

	Allocation	*Max*	*Need*	*Available*
	R1 R2	*R1 R2*	*R1 R2*	*R1 R2*
P_1	1 0	1 1	0 1	0 1
P_2	0 0	1 1	1 1	

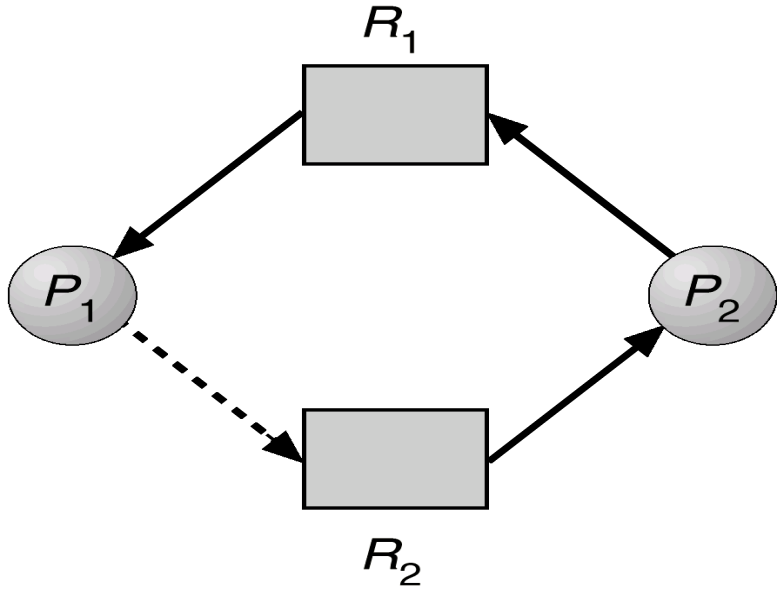
- The system is in a safe state since the sequence < P_1, P_2> satisfies safety criteria.



Very simple example execution of Bankers Algo (snapshot 2)

	Allocation	*Max*	*Need*	*Available*
	R1 R2	*R1 R2*	*R1 R2*	*R1 R2*
P_1	1 0	1 1	0 1	0 0
P_2	0 1	1 1	1 0	

- Allocating B to P_2 leaves the system in an unsafe state since there is no sequence that satisfies safety criteria (no *need* can be satisfied, since *Available* vector is 0).
- Hence OS must suspend P2 until P1 has finished and then allocate the resources to P2

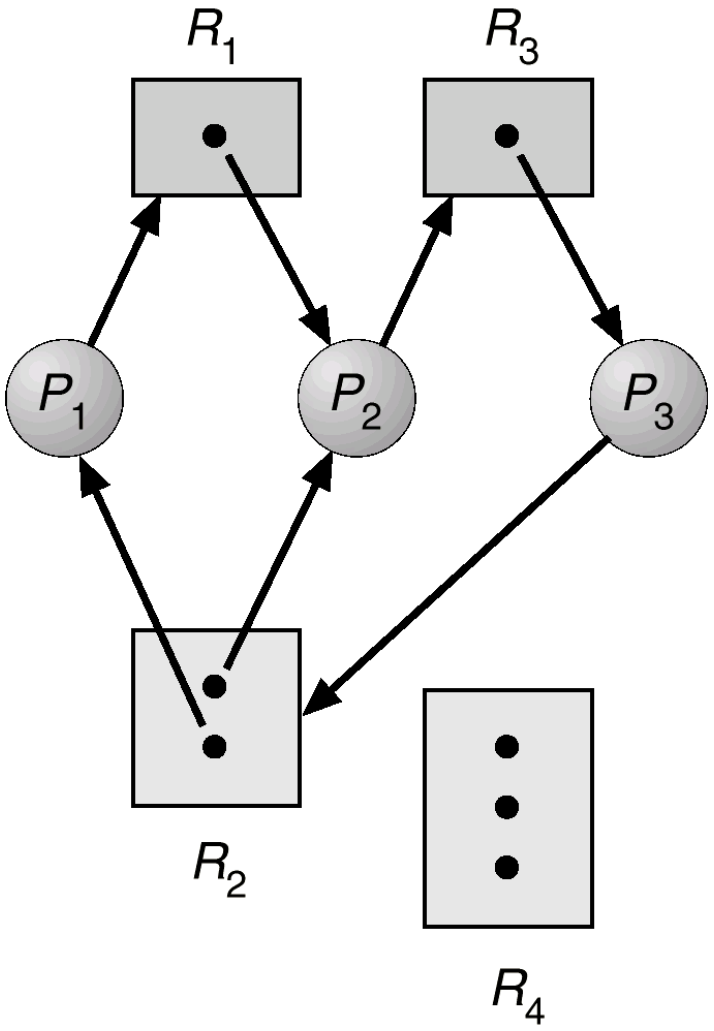


EXTRA SLIDES/NOTES

- Part 1: Complement: Banker's algo for deadlock avoidance: data structures and safety check

- Part 2: Use of Banker's algo idea to do deadlock detection

Another example of a Resource Allocation Graph With A Deadlock

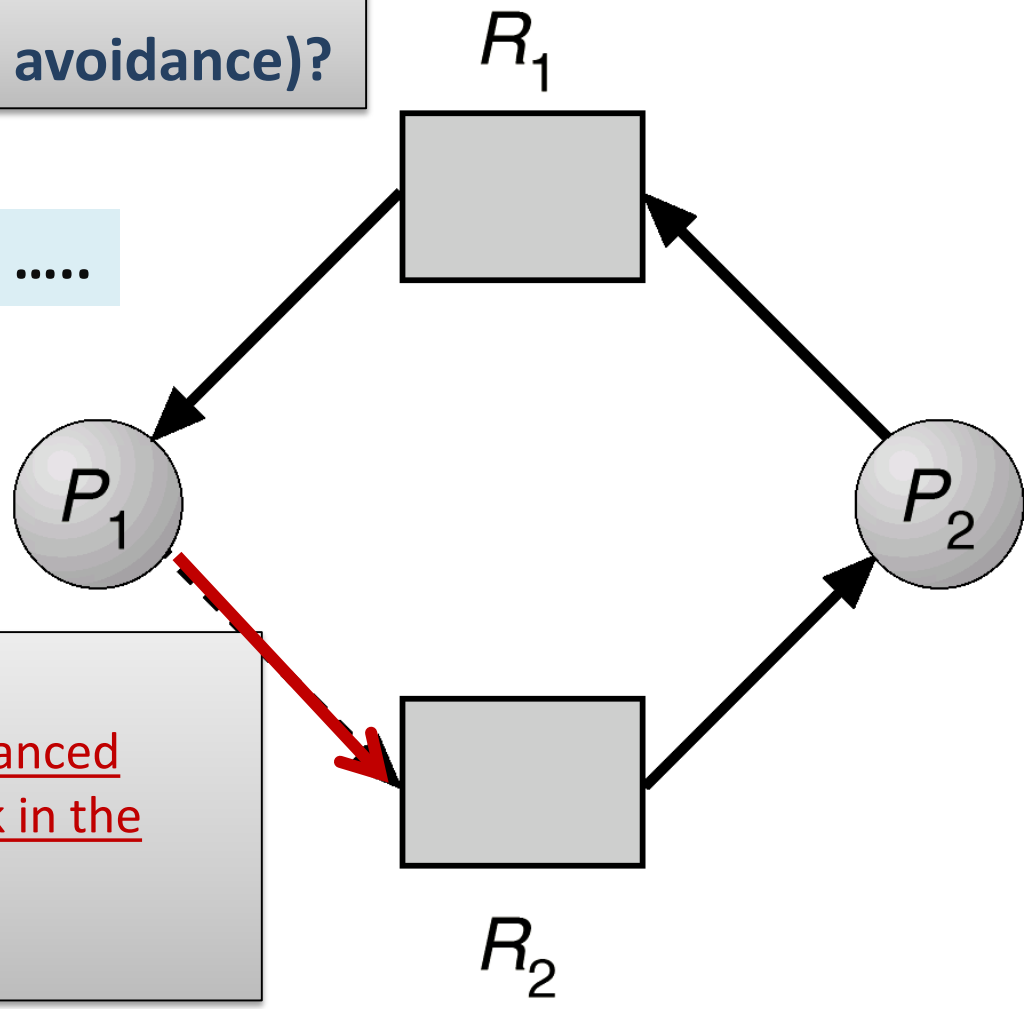


Observe for deadlock detection:

Recall Unsafe State
in the enhanced RA graph (for deadlock avoidance)?

In the actual RA graph it is …..

… deadlocked!!



Detection algorithm:

what we can do for checking safety in enhanced graph, can serve for checking no-deadlock in the resource allocation graph

Eg Using Banker's algo idea

Deadlock Detection

Note:

- similar as detecting unsafe states using Banker's algo
- Q: if they cost the same, why not use avoidance instead of detection&recovery?
  - Hint: think trade-off between checking cost and recovery cost

Data structures:

- *Available:* vector of length *m:* number of available resources of each type.
- *Allocation:* *n x m* matrix: number of resources of each type currently allocated to each process.
- *Request: n x m* matrix: **current request** of each process. *Request* [*ij*] = *k:* P_i is requesting *k* more instances of resource type R_j.

Detection-Algorithm Usage

- When, and how often, to invoke:
- We don't want to be too late to detect; think

  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?

- Reason: If algorithm is invoked arbitrarily rarely,
  - there may be many cycles in the resource graph ⇒ we would not be able to tell which of the many deadlocked processes "caused" the deadlock.