Synchronization - Part 2
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF
GOTHENBURG

What to read (Main textbook) - Both for part 1 and part 2

- Chapter 2.3.1-2.3.6, 2.5.1-2.5.2, 6.1-6.2, 6.5-6.6, 6.7.3-6.7.4;
- Quicker reading, for awareness, of sections 2.3.7-2.3.10, 6.3

  (facultative reading: sections 6.1-6.7, 6.9, 7.1-7.5, 7.6-7.8 from OS Concepts by Silberschatz et-al).

Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

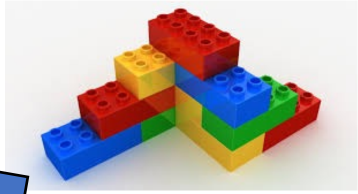
Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport's bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

the Bounded buffer producer-consumer problem

Using: primitives

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



Construct: objects / solve specific synchronization problems

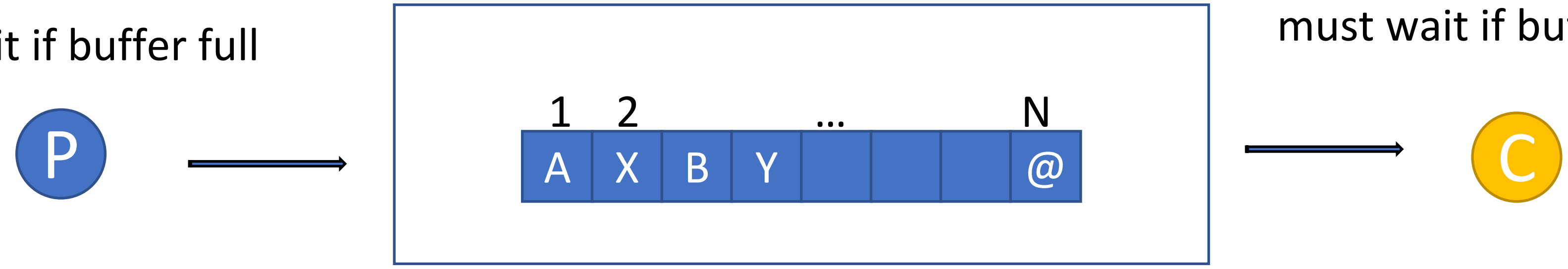
- 2 thread CS, n-thread-CS
- Semaphores, mutex-locks, …
- Producer-consumer (bounded buffer)
- Dining philosophers
- Transactions
- …

the Bounded buffer producer-consumer problem

Buffer with **space for N** items; accessing common entries is a **critical section**

Producer inserts items;
must wait if buffer full

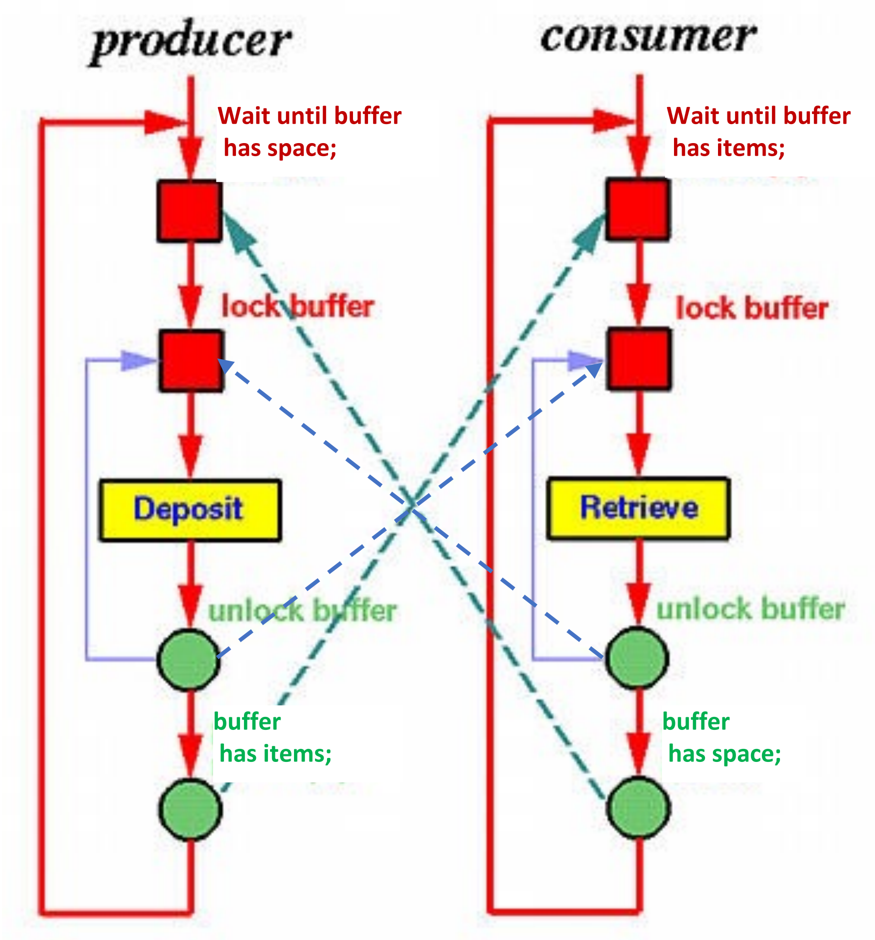
Consumer removes items;
must wait if buffer empty



Solve this synch problem using semaphores

What synchronization 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**



a solution to the Bounded buffer producer-consumer problem

Synchronization variables:

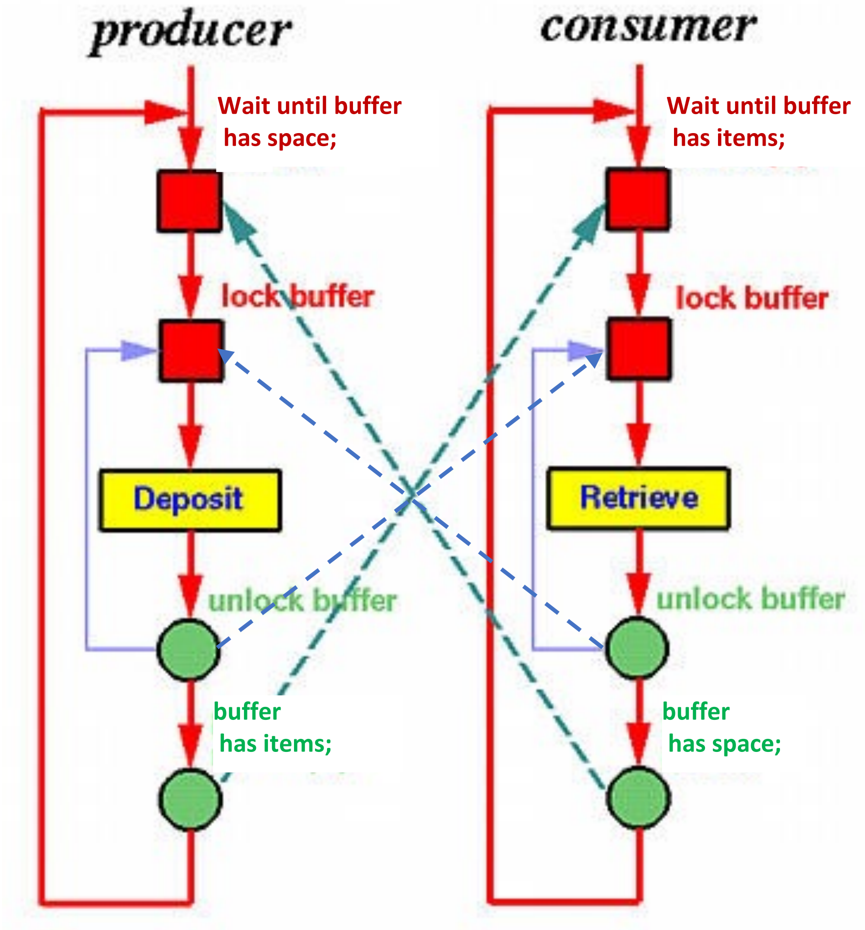
- Binary semaphore mutex_sem initialized to 1
- General semaphore buffer-has-items initialized to 0
- General semaphore buffer-has-space initialized to N

```
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);
```



Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Resource allocation

- Processes/threads need resources (e.g., memory pages, printer, access to parts of shared data structure, etc.)
  - Our focus: reusable resources
- a human analogy: process = go fishing; needed resources: boat, fishing-rod

Structure of process/thread *P*

Repeat

Request resources (entry section)

critical section

Release resources (exit section)

remainder section

Forever

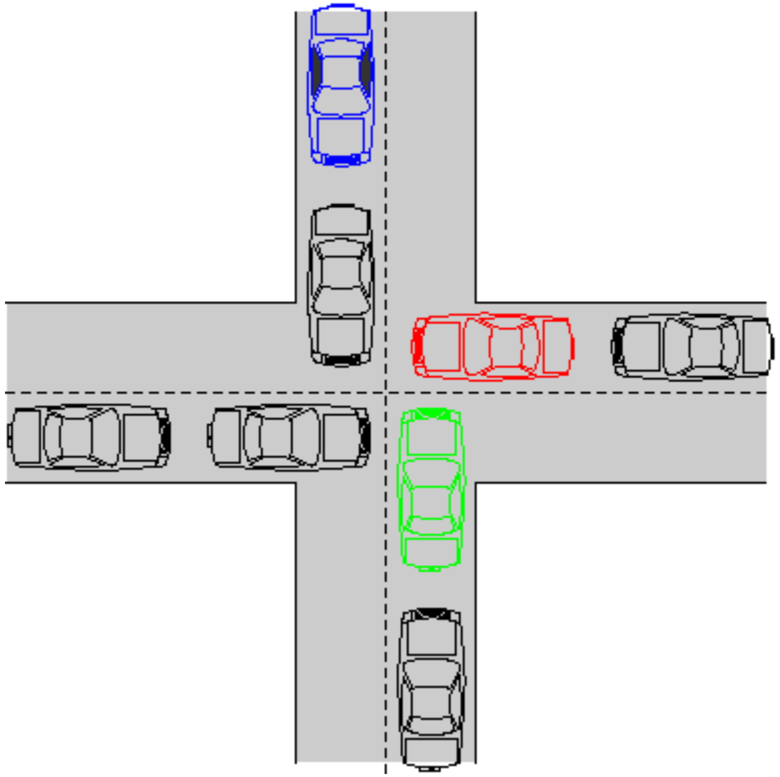
Problem formulation:

A solution must provide the entry and exit sections

It must ensure:

1. **Acquire/Release all the needed resources**
2. **Mutual Exclusion**.
3. **Progress: <u>no deadlock</u>**
4. **Fairness (e.g., Bounded Waiting , no starvation**)

What is a deadlock?

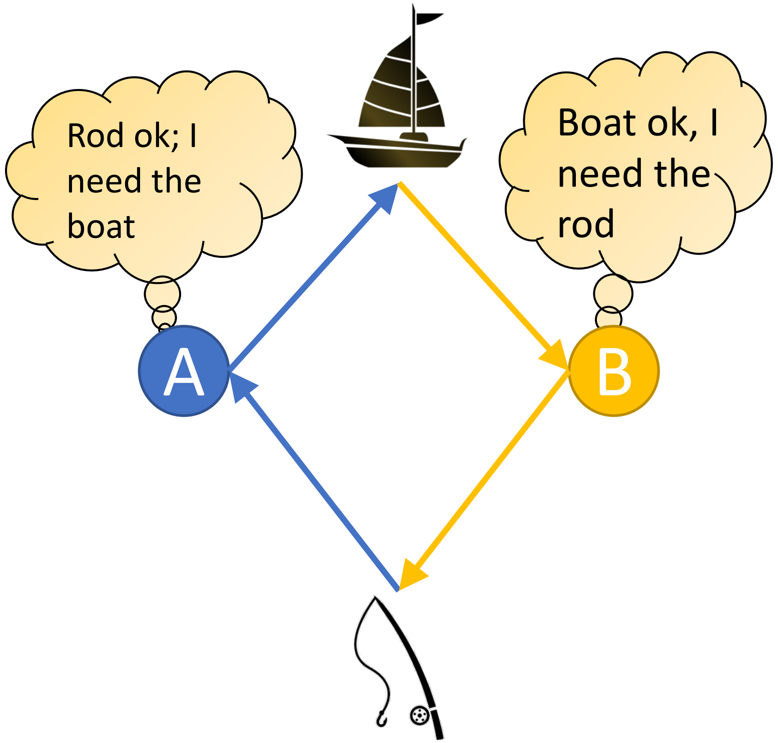


A set of processes/threads blocking each-other so that none of them can proceed:

How can it occur?

4 necessary conditions for Deadlock [Coffman et al 1971]

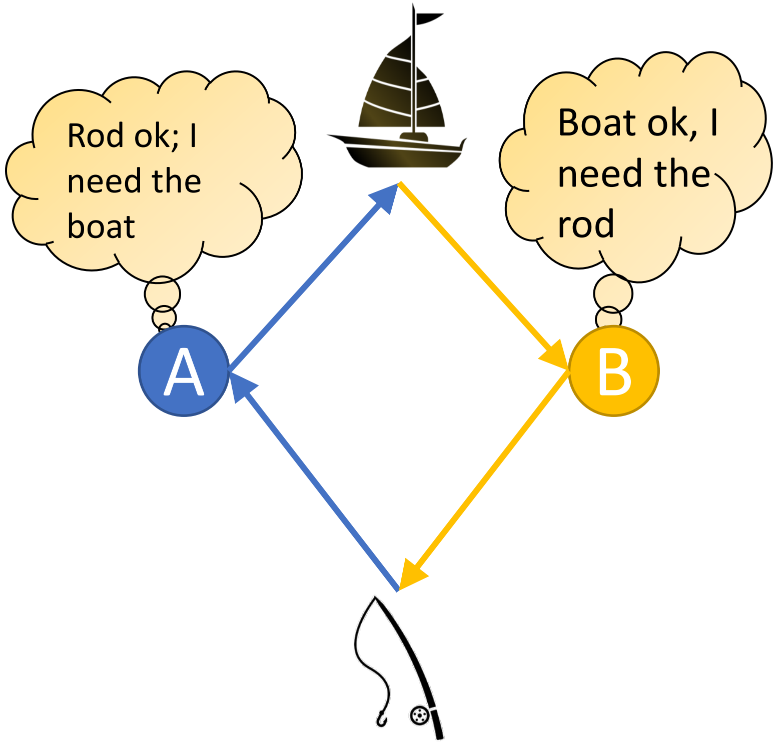
Theorem: all 4 conditions hold simultaneously when a deadlock occurs:



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

4 necessary conditions for Deadlock [Coffman et al 1971]

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



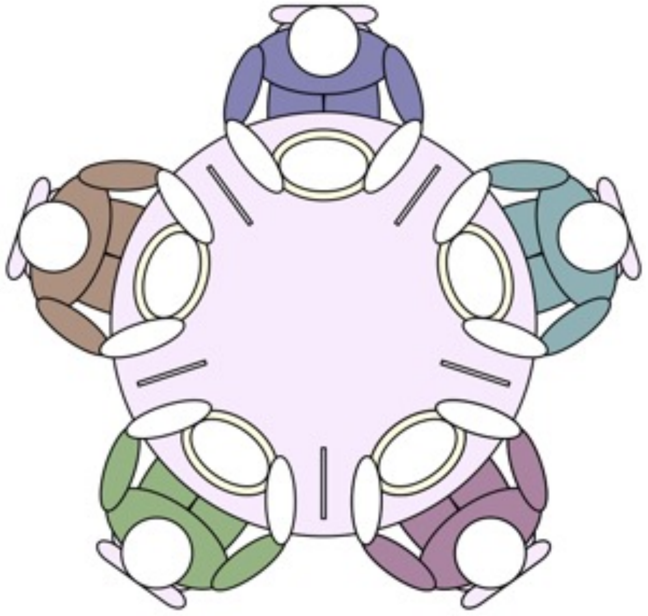
1. **Mutual exclusion:** only one process at a time can use a resource.
2. **Hold and wait:** a process holding some resource can request additional resources and wait for them if they are held by other processes. → GET ALL RESOURCES AT ONCE
3. **No preemption:** a resource can only be released by the process holding it, after that process has completed its task. → THREADS RELEASE RESOURCES IF THEY MANAGE TO GET SOME BUT NOT ALL
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. → ACQUIRE RESOURCES IN A CERTAIN ORDER

Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Dining philosophers [Dijkstra65]

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



Structure of process/thread *P*

Repeat

Request resources (left + right chopstick)

eat

Release resources (left + right chopstick)

think

Forever

Dining philosophers: pick-left-then-right approach

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

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

Dining philosophers: pick-left-then-right approach

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

Hold and wait

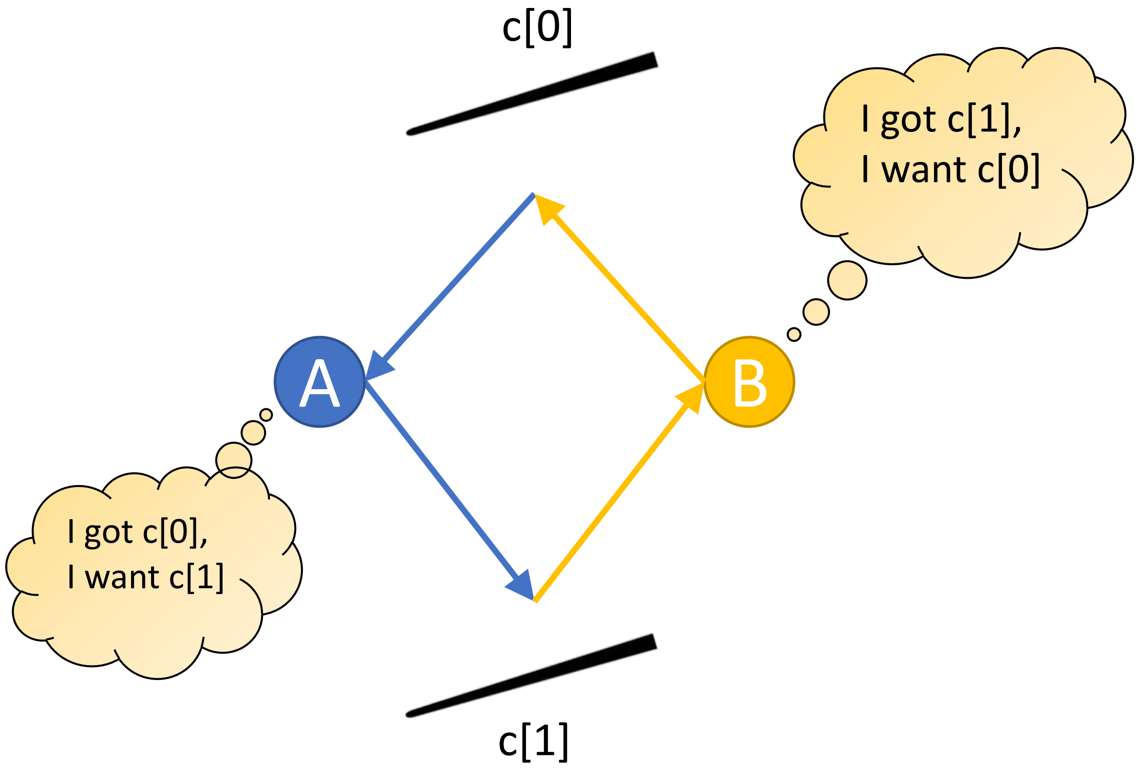
No preemption

Dining philosophers: pick-left-then-right approach

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

Hold and wait

No preemption



Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Dining philosophers: pick-one-at-a-time without circular wait

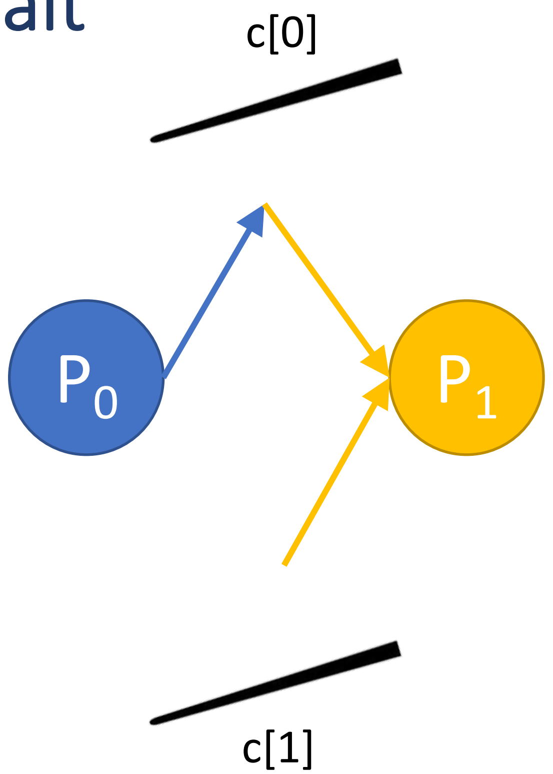
```
Shared var c[0..n-1]: bin-semaphore // one for
each chopstick; init all 1
Pi (except Pn-1):
do
        Wait c[i];   // pick left chopstick
        Wait c[(i+1) mod n];  // pick right chopstick
        // Eat
        Signal c[(i+1) mod n];  leave right chopstick
        Signal c[i];  // leave left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

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

Dining philosophers: pick-one-at-a-time without circular wait

Shared var c[0..n-1]: bin-semaphore // one for each chopstick; init all 1
P_i (except P_{n-1}):
do
  Wait c[i]; // pick left chopstick
  Wait c[(i+1) mod n]; // pick right chopstick
  // Eat
  Signal c[(i+1) mod n]; leave right chopstick
  Signal c[i]; // leave left chopstick
  // Think
forever

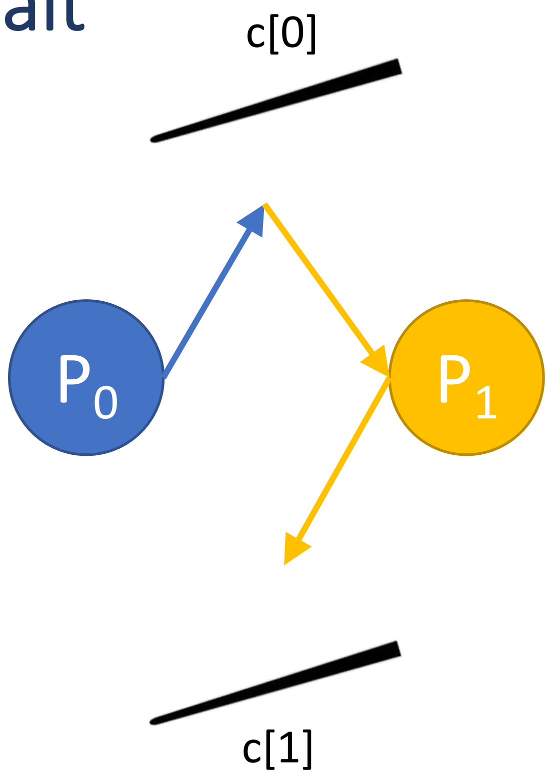
Shared var c[0..n-1]: bin-semaphore // one for each chopstick; init all 1
P_{n-1}:
do
  Wait c[(i+1) mod n]; // pick right chopstick
  Wait c[i]; // pick left chopstick
  // Eat
  Signal c[i]; // leave left chopstick
  Signal c[(i+1) mod n]; leave right chopstick
  // Think
forever



Dining philosophers: pick-one-at-a-time without circular wait

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

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



Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport's bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Dining philosophers: Fight the no-preemption

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

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

```
Pi:
   local var holding_both_chopsticks: boolean;
   repeat
      while (not holding_both_chopsticks) {
         lock(c[i])
         if !trylock(c[(i+1)%n]) then release(c[i])
         else holding_both_chopsticks := true }
      // Eat
      release(c[i])
      release(c[(i+1)%n])
      holding_both_chopsticks := false
      // Think
forever
```

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

```
lock(c : chopstick_structure):
repeat
until (trylock(c))
```

```
release(c : chopstick_structure):
wait(c.s)
c.available := true
signal(c.s)
```

Dining philosophers: Fight the no-preemption

- Mutual exclusion: ok
- Progress: no deadlock … but livelock is possible!
- Fairness: a process can starve…

```
Pi:
  local var holding_both_chopsticks: boolean;
  repeat
    while (not holding_both_chopsticks) {
      lock(c[i])
      if !trylock(c[(i+1)%n]) then release(c[i])
      else holding_both_chopsticks := true }
    // Eat
    release(c[i])
    release(c[(i+1)%n])
    holding_both_chopsticks := false
    // Think
forever
```

Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport's bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Dining philosophers: Fight the hold-and-wait

Idea: philosophers agree on who eats next instead of who gets a chopstick

```
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) // i.e., get both chopsticks
   // eat
   exitCS(i) // i.e., leave chopsticks
forever
```

```
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]) }
```

```
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)
```

Dining philosophers: Fight the no-preemption

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

- If you are interested in a solution with fairness too: https://dl.acm.org/doi/pdf/10.1145/62546.62567?casa_token=4uO24jxkwEAAAAAA:jJlAILeISZe5Uu2ERv6O-dTq_0LbSmRpv0beOZ_3vDi50otRS-_HqB30GoWDib1zVQ9jjrhx4w0

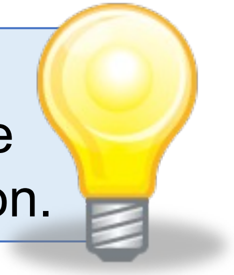
Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Lamport’s bakery algorithm

Critical section for n threads:

Idea: before entering its critical section, each thread gets a number. Holder of the smallest number enters the critical section.



Note: the decentralized scheme generates numbers in non-decreasing order; e.g., 1, 2, 3, 3, 4, 5, 5, 5, 6

If threads P_i and P_j choose the same number:
if $i<j$, then P_i goes first; else P_j goes first.
I.e., we use threads’ ids to break ties

Lamport's bakery algorithm

```
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;
```

Why does it satisfy the 3 conditions:

Mutex (no 2 threads A and B in CS concurrently): 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

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

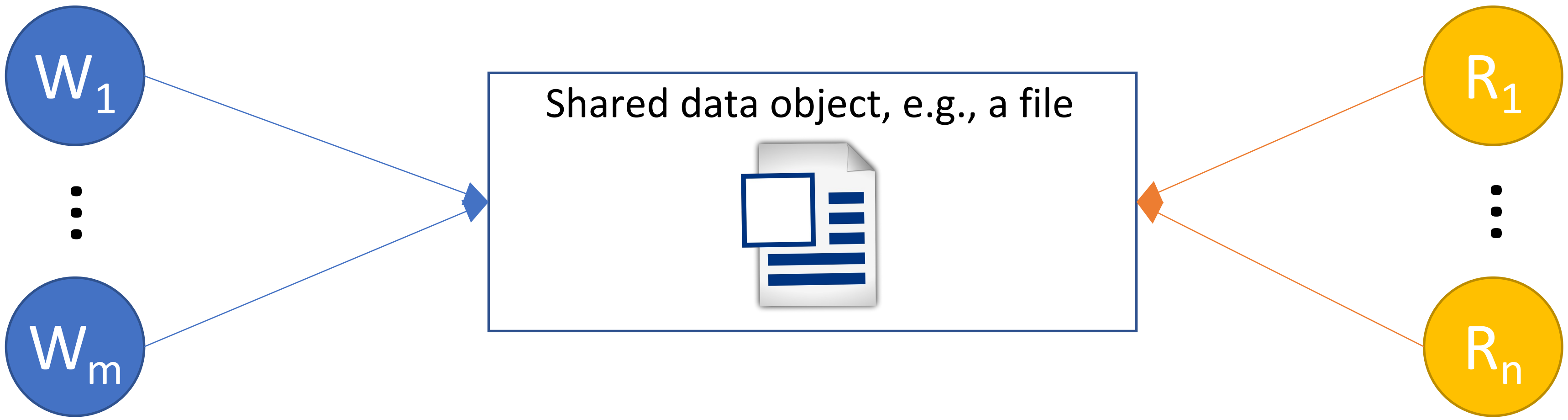
Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Readers/Writers problem

Writers write; must wait if some writer or reader is accessing the shared data object (i.e., *only one writer is allowed to write at a time*)

Readers read; must wait if some writer is accessing the shared data object (i.e., *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

Reader R_j

if I am first in a batch of readers to check

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 …)

Writer W_i

wait until no one accesses the object

// WRITE

no longer / no one accessing the object

Readers/Writers problem

W_i R_j

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

W_i

```
Writer

Repeat
   wait(noone_accesses);
   // WRITE
   signal(noone_accesses)
forever
```

R_j

```
Reader

repeat
   wait(protect_check); // CS to change and check rc variable
   if rc++ == 1 then wait(noone_accesses) // "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
```

Agenda

- Bounded-buffer producer/consumer
- Resource allocation, deadlocks, and necessary conditions for deadlocks
- Dining philosophers
  - without circular wait
  - without no-preemption
  - without hold-and-wait
- Lamport’s bakery algorithm
- Readers/Writers problem
- OS as arbitrator for deadlock avoidance/recovery

Deadlock avoidance using the OS as arbitrator

Resource request & allocation is managed by the OS

Deadlock *avoidance*:

- deadlock *might be* possible if resources are granted arbitrarily,
- but OS uses extra info to grant requests and schedule processes so that it avoids deadlock
- **Banker's algorithm[Dijkstra]**

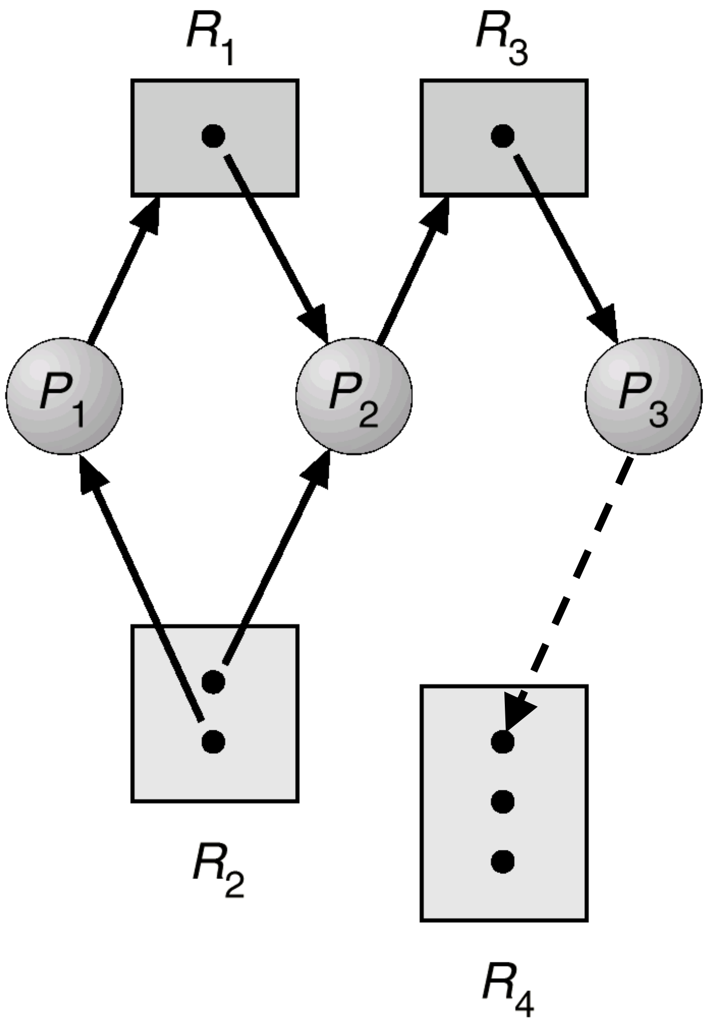
Deadlock avoidance by the OS: system model

Resource types $R_1, R_2, \ldots, R_m$

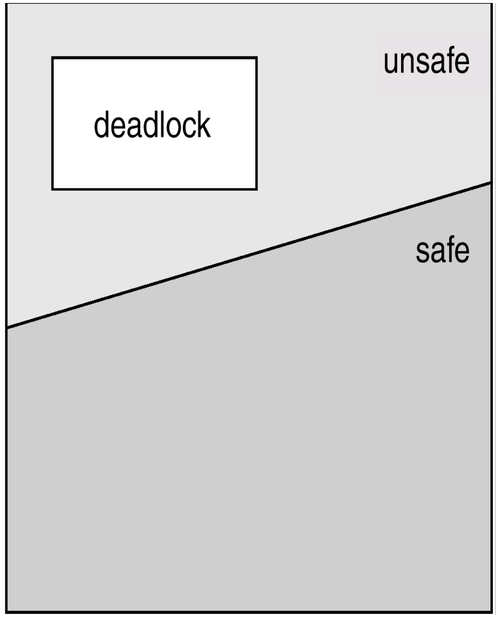
- *e.g., CPU, memory space, I/O devices, files*
- each resource type R_i has W_i instances.

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: $P_i \rightarrow R_j$
    - If dotted: potential request
  - assignment edge: $R_j \rightarrow P_i$



Deadlock avoidance using the OS as arbitrator

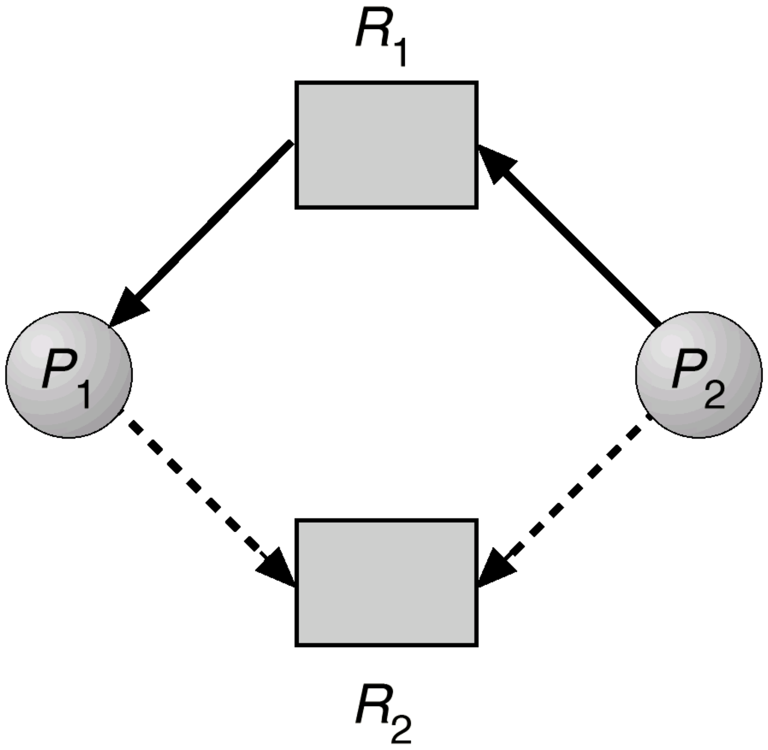


Deadlock-avoidance algo, run by OS:

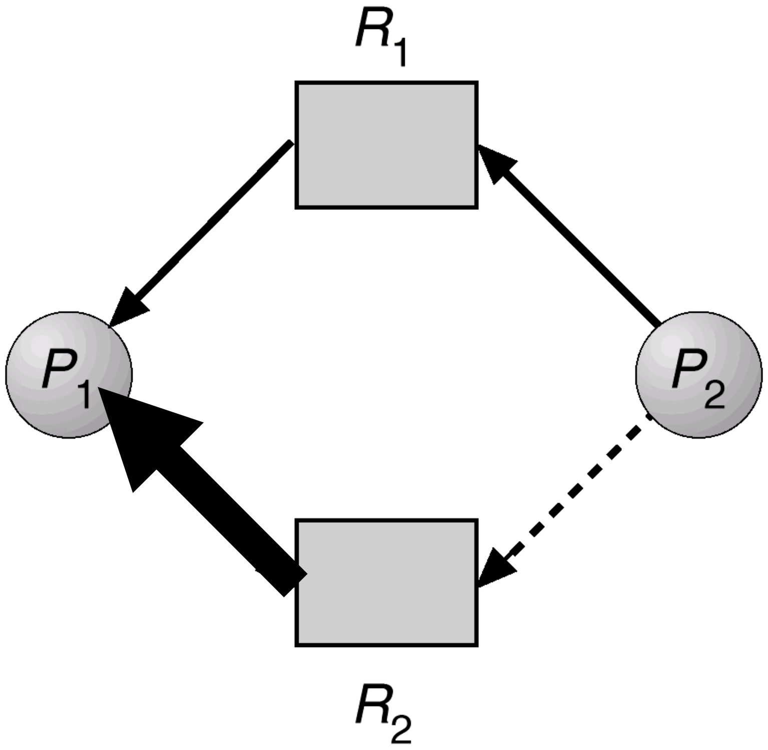
- examines the resource-allocation state…
  - *Available, allocated resources*
  - *maximum possible demands* of the processes.
- …to ensure there is *no potential deadlock*:
  - unsafe state ⇒ deadlock might occur (i.e., later, if all procs request their maximum and no-one can be granted)

- *Avoidance* = ensure that *system will not enter an unsafe* state, by *suspending processes with risky requests*, until enough resources are freed.

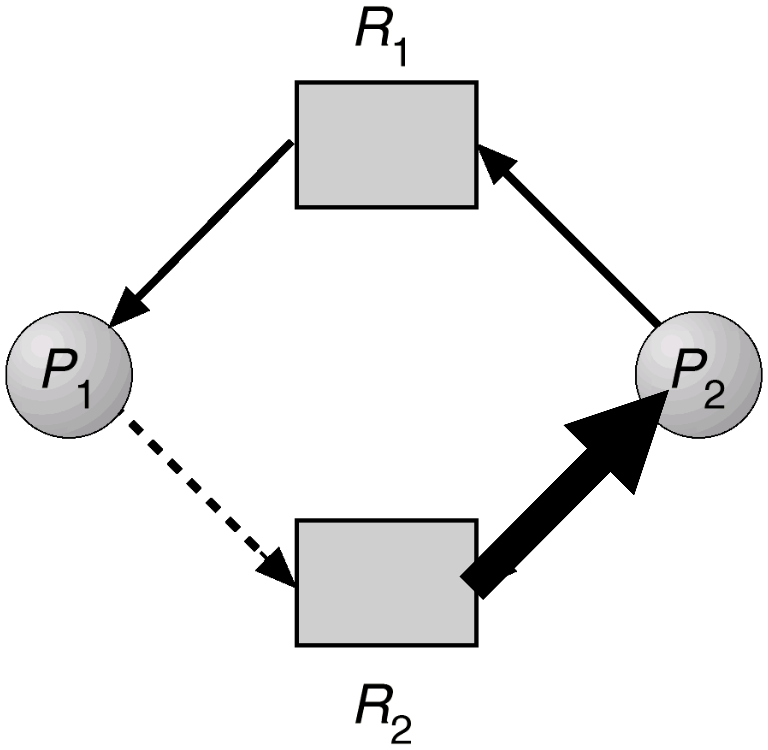
Deadlock avoidance by the OS: example



Example Safe State



Safe State



Example Unsafe State:

i.e., granting the P2-R2 request should not be made until P1 finishes

Deadlock avoidance by the OS: banker's algorithm

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***

***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

Deadlock detection and recovery

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

(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 ….