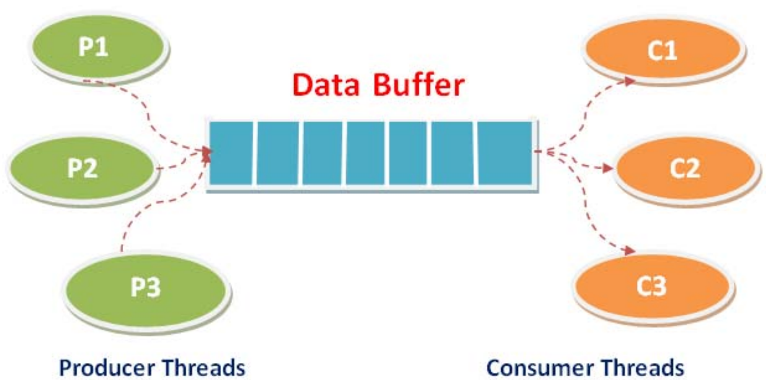
Lecture 6:

Inter-process Communication and Synchronization :

Part 2: Classical Synchronization Problems



Outline

- **Classical Synchronization Problems**
  - Producer-Consumer Problem
  - Dining Philosophers Problem
- **A Formal Model of Deadlock**
  - Resource Allocation Graphs (RAG)

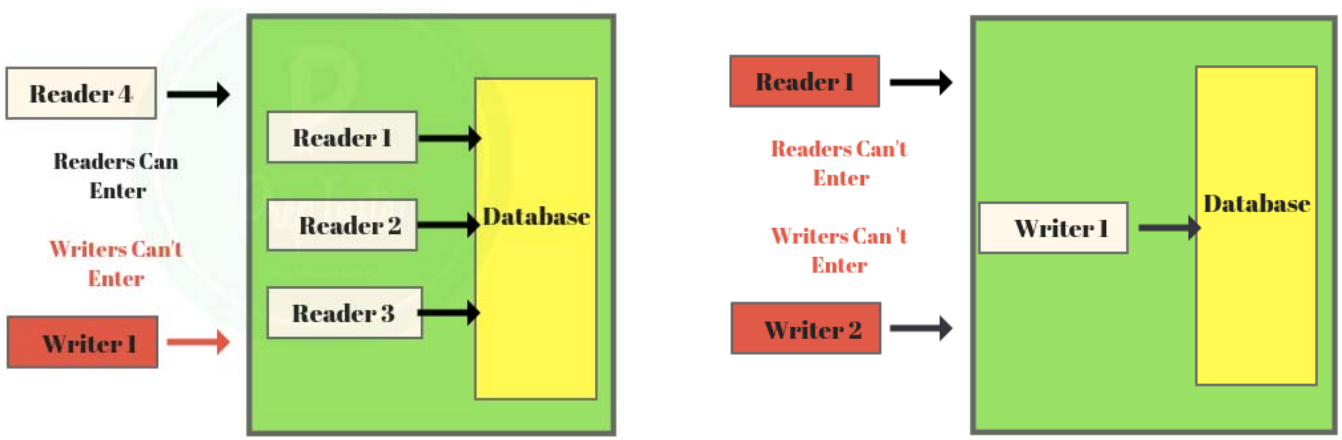
Classical Synchronization Problems

- **Producer-Consumer Problem**
  - also known as the **bounded-buffer problem**
- **Dining Philosophers Problem**
  - allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner
- **Readers and Writers Problem** (Not discussed)
  - Database: multiple threads read/update
- **Sleeping Barber Problem** (Not discussed)

For Your Own Interest

Readers Writers Problem

- Any number of readers can <u>read</u> from the shared resource simultaneously, but only <u>one writer</u> can write to the shared resource.
- When a writer is writing data to the resource, no other process can access the resource.
- A writer cannot write to the resource if there are <u>non zero number of readers</u> accessing the resource at that time.

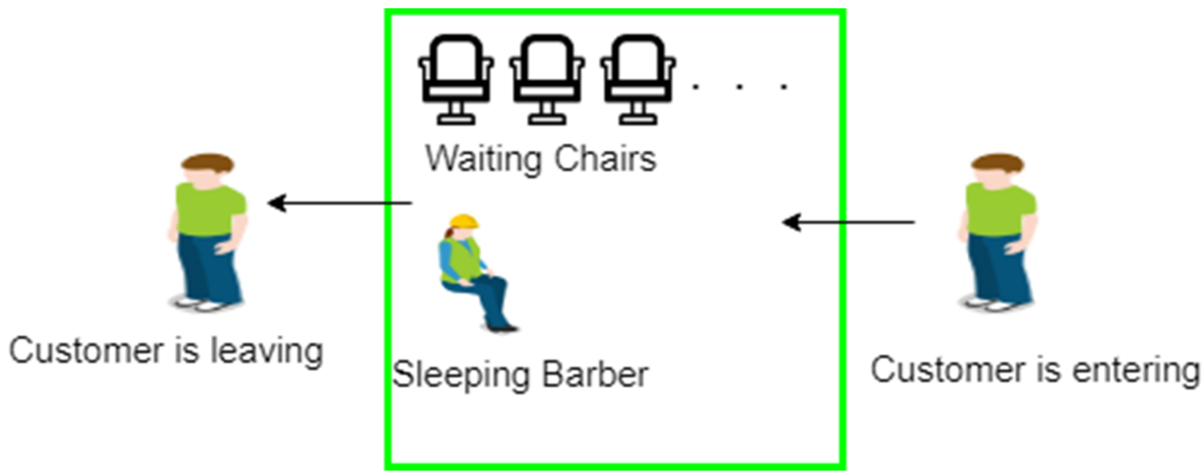


https://prepinsta.com/operating-systems/readers-writers-problem/
https://course.ccs.neu.edu/cs3650sp20/Labs/7/reader_writer.c
https://shivammitra.com/reader-writer-problem-in-c/#

For Your Own Interest

Sleeping Barber problem

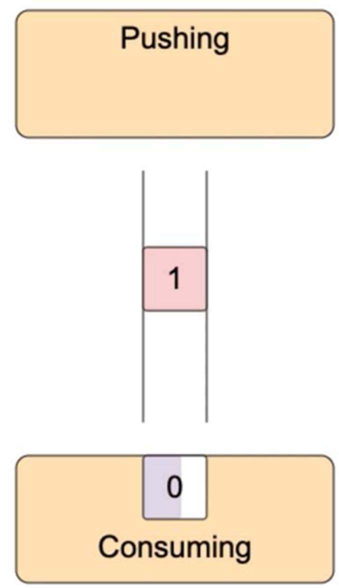
- Barber shop with one barber, **one barber chair** and **N chairs to wait in**.
- When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in.
- When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy.



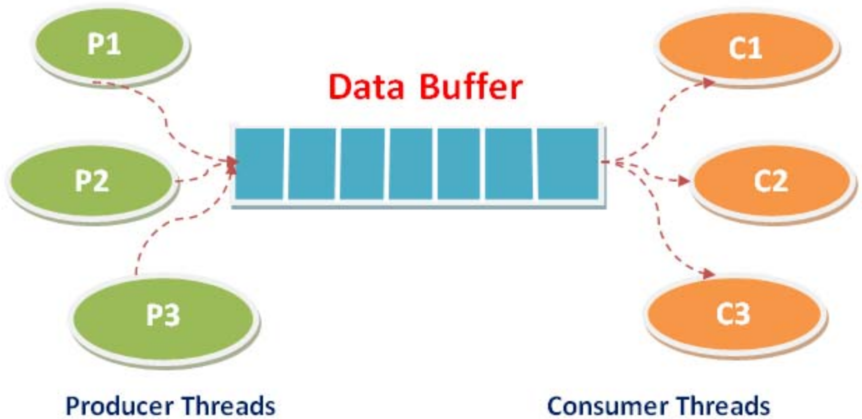
https://github.com/tonyjmartinez/Sleeping-Barber/blob/master/mybarber.c
https://learningcomputersciencemadeeasy.wordpress.com/2017/04/08/sleeping-barber-problem-code-in-c/

Producer-Consumer Problems (bounded-buffer problem)

- The problem is to make sure that the **producer** won't try to add data into the buffer if it's full and that the **consumer** won't try to remove data from an empty buffer.
  - Use pthread_mutex_lock( ) and pthread_mutex_unlock() to enforce **mutual exclusion** for the critical sections modifying the queue.
  - Use two counting semaphores **"full"** and **"empty"** to keep track of the current number of full and empty buffers respectively.

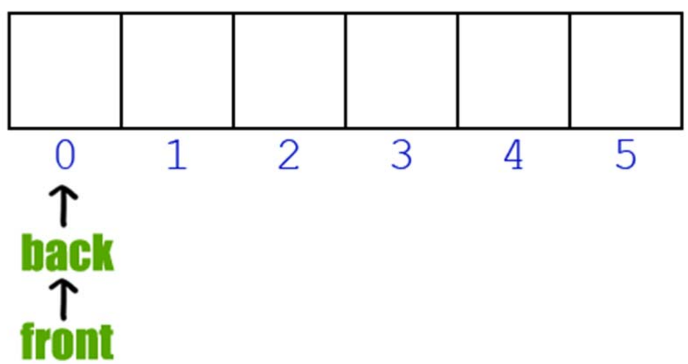


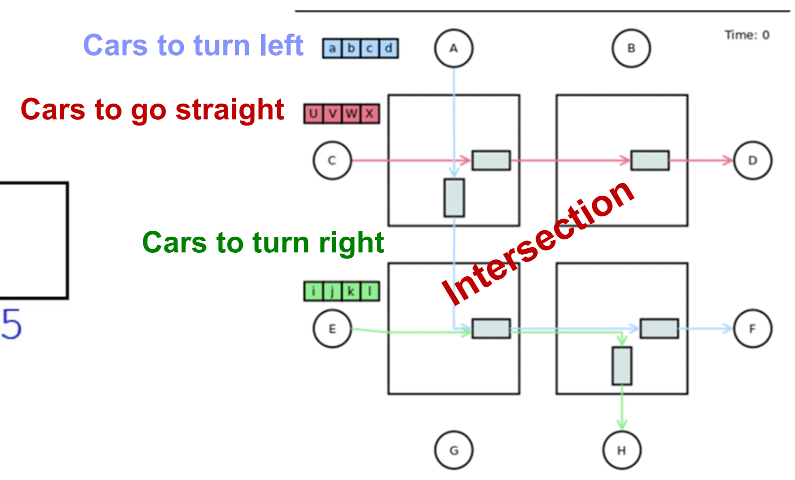
Producer-Consumer Problems
(bounded-buffer problem)



FIFO Buffer

Circular Buffers

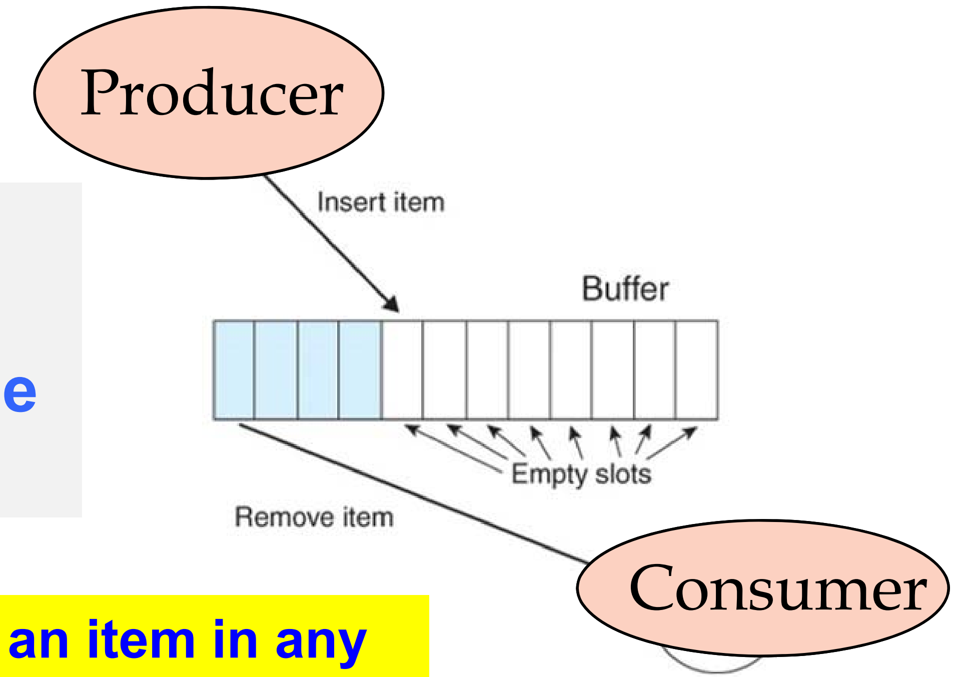




Traffic control

Bounded-Buffer Problem using Counting Semaphores

- **Producer**: keep moving items to the buffer.
- **Consumer**: keep removing items from the buffer.
- **Buffer**: *N* entries.

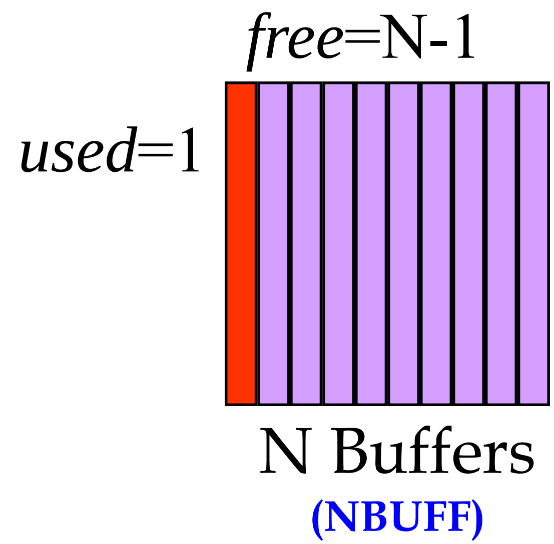


Because the buffer has a maximum size, this problem is often called the *bounded buffer problem*.

Our assumption: you can put an item in any empty slot, or remove item from any slot.)

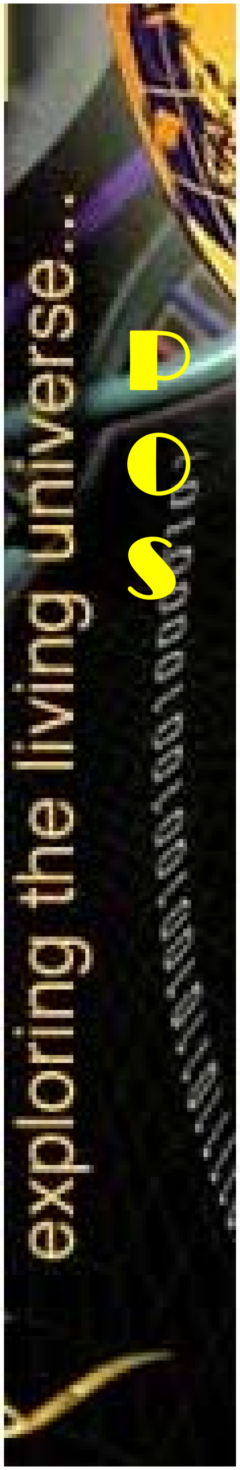
Bounded-Buffer Problem
(with one producer + one consumer)

- **Binary semaphore:**
  - ***mutex*** -- for mutual exclusion.
  - Initially ***mutex = 1***
- **Counting semaphores:**
  - ***free*** -- no. of empty buffers.
    - Initially, ***free = N***
  - ***used*** --- no. of used buffers.
    - Initially, ***used = 0***



/* initialize semaphores in a *Pthreads* program*/

```
/* initialize three semaphores */
sem_init (&shared.mutex, 0, 1);
sem_init (&shared.nempty, 0, NBUFF);
sem_init (&shared.nstored, 0, 0);
```



Bounded-Buffer Problem
(code analysis)

Code for producer

repeat

…

Produce an item in *nextp*;

…

P(*free*); /* synchronization */

P(mutex)

…

Add *nextp* to buffer;

…

V(mutex);

V(*used*);

until false;

Code for consumer

repeat

P(*used*); /* synchronization */

P(mutex)

…

Remove one item from buffer to *nextc*;

…

V(mutex);

V(*free*);

…..

consume the item in *nextc*;

until false;

Bounded-Buffer Problem
(code analysis)

Code for producer

repeat

...

Produce an item in nextp;

...

P(free); /* synchronization */

P(mutex)

...

Add nextp to buffer;

...

V(mutex);

V(used);

until false;

Critical section

Code for consumer

repeat

P(used); /* synchronization */

P(mutex)

...

Remove one item from buffer to nextc;

...

V(mutex);

V(free);

.....

consume the item in nextc;

until false;

Bounded-Buffer Problem
(code analysis)

Initially, free = 4, used = 0

Code for producer

```
repeat
    ...
    Produce an item in nextp;
        ...
    P(free);        /*          -1
      synchronization */
    P(mutex)
        ...
    Add nextp to buffer;
        ...
    V(mutex)
    V(used);                    +1
until false;
```

synchronization

Code for consumer

```
repeat
    P(used); /*                 -1
      synchronization */
    P(mutex)
        ...
    Remove one item from buffer
      to nextc;
        ...
    V(mutex)
    V(free);                    +1
        .....
    consume the item in nextc;
until false;
```

Bounded-Buffer Problem

(when to stop producer?)

Code for producer

```
repeat
    ...
    Produce an item in nextp;
        ...
    P(free); /* synchronization */
    P(mutex)
        ...
    Add nextp to buffer;
        ...
    V(mutex);
    V(used);
until false;
```

If **free=0** → "no more empty space" in buffer
Then wait !!

One more produced, who is wanting for it

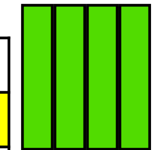
In V(used),
used=used+1;
Wakeup one "consumer" process who was blocked because
Buffer is empty (i.e., used=0)

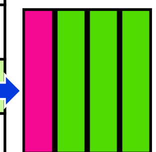
The bounded buffer problem: single CPU

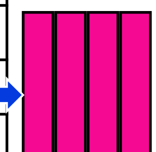
A buffer with 4 entries (free=4, used=0 initially)

Consumer thread action	Producer thread action	Free	Used
Thread starts	Thread inactive (not scheduled)	4	0
P(Used) blocks. Suspended.		4	0
wait in semaphore queue (Used)	**P(Free)** flows through	**3**	0
wait in semaphore queue (Used)	**Item Added. V(Used)**	3	**1**
Consumer wakes up → Ready queue	**P(Free)** flows through	**2**	1
ready queue (not running)	**Item Added. V(Used)**	2	**2**
ready queue (not running)	**P(Free)** flows through	**1**	2
ready queue (not running)	**Item Added. V(Used)**	1	**3**
ready queue (not running)	**P(Free)** flows through	**0**	3
ready queue (not running)	**Item Added. V(Used)**	0	**4**
ready queue (not running)	**P(Free)** blocks. Suspended	**0**	4
Context switch→ P(Used) completes	wait in semaphore queue (Free)	0	3
Item Removed. **V(Free)**	wait in semaphore queue (Free)	***1***	3
P(Used) flows through	***Producer wakes up → Ready queue***	1	2
Item Removed. V(Free)	ready queue (not running)	2	2
P(Used) flows through	ready queue (not running)	2	1
Item Removed. V(Free)	ready queue (not running)	3	1
P(Used) flows through	ready queue (not running)	3	0
Item Removed. V(Free)	ready queue (not running)	4	0
P(Used) blocks. Suspended	ready queue (not running)	4	0

time







Producer-consumer in pthreads and semaphore

```
/* main.c */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include "buffer.h"

#define RAND_DIVISOR 100000000
#define TRUE 1

/* The mutex lock */
pthread_mutex_t mutex;

/* the semaphores */
sem_t full, empty;

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

/* buffer counter */
int counter;

pthread_t tid;       //Thread ID
pthread_attr_t attr; //Set of thread attributes

void *producer(void *param); /* the producer thread */
void *consumer(void *param); /* the consumer thread */

void initializeData() {

   /* Create the mutex lock */
   pthread_mutex_init(&mutex, NULL);

   /* Create the full semaphore and initialize to 0 */
   sem_init(&full, 0, 0);

   /* Create the empty semaphore and initialize to BUFFER_SIZE */
   sem_init(&empty, 0, BUFFER_SIZE);

   /* Get the default attributes */
   pthread_attr_init(&attr);

   /* init buffer */
   counter = 0;
}
```

Mutex

Semaphores

buffer

Use mutex lock to protect critical section

Semaphore initialization

Producer

```
/* Producer Thread */
void *producer(void *param) {
   buffer_item item;

   while(TRUE) {
      /* sleep for a random period of time */
      int rNum = rand() / RAND_DIVISOR;
      sleep(rNum);

      /* generate a random number */
      item = rand();

      /* acquire the empty lock */
      sem_wait(&empty);
      /* acquire the mutex lock */
      pthread_mutex_lock(&mutex);

      if(insert_item(item)) {
         fprintf(stderr, "
      }
      else {
         printf("producer
      }
      /* release the mute
      pthread_mutex_unlock
      /* signal full */
      sem_post(&full);
   }
}
```

Full=3
Empty=2

Blocked if empty=0

-1 empty

full +1

Consumer

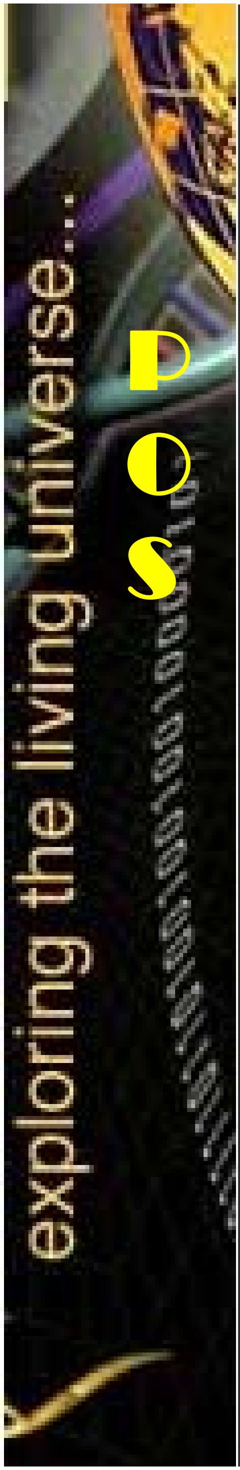
```
/* Consumer Thread */
void *consumer(void *param) {
   buffer_item item;

   while(TRUE) {
      /* sleep for a random period of time */
      int rNum = rand() / RAND_DIVISOR;
      sleep(rNum);

      /* aquire the full lock */
      sem_wait(&full);
      /* aquire the mutex lock */
      pthread_mutex_lock(&mutex);
      if(remove_item(&item)) {
         fprintf(stderr, "Consumer report error c
      }
      else {
         printf("consumer consumed %d\n", item);
      }
      /* release the mutex lock */
      pthread_mutex_unlock(&mutex);
      /* signal empty */
      sem_post(&empty);
   }
}
```

Blocked if full =0

full -1

Use mutex lock to protect critical section

+1 empty




```
yczhong@workbench:~/snippets$ gcc T4-pcp-2.c -o pcp -pthread
yczhong@workbench:~/snippets$ ./pcp
Producer 1: Insert Item 1804289383 at 0
Producer 2: Insert Item 846930886 at 1
Producer 3: Insert Item 1681692777 at 2
Consumer 4: Remove Item 1804289383 from 0
Producer 5: Insert Item 1957747793 at 3
Producer 1: Insert Item 424238335 at 4
Consumer 3: Remove Item 846930886 from 1
Producer 4: Insert Item 1714636915 at 0
Consumer 1: Remove Item 1681692777 from 2
Consumer 2: Remove Item 1957747793 from 3
Producer 3: Insert Item 1649760492 at 1
Consumer 5: Remove Item 424238335 from 4
Consumer 4: Remove Item 1714636915 from 0
Producer 5: Insert Item 596516649 at 2
Producer 2: Insert Item 719885386 at 3
Consumer 3: Remove Item 1649760492 from 1
Consumer 2: Remove Item 596516649 from 2
Producer 1: Insert Item 1189641421 at 4
Producer 4: Insert Item 1025202362 at 0
Consumer 1: Remove Item 719885386 from 3
Producer 3: Insert Item 1350490027 at 1
Producer 2: Insert Item 1102520059 at 2
Consumer 3: Remove Item 1189641421 from 4
Consumer 4: Remove Item 1025202362 from 0
Producer 5: Insert Item 783368690 at 3
Producer 5: Insert Item 304089172 at 4
Consumer 3: Remove Item 1350490027 from 1
Producer 1: Insert Item 2044897763 at 0
Consumer 5: Remove Item 1102520059 from 2
Consumer 4: Remove Item 783368690 from 3
Producer 2: Insert Item 1540383426 at 1
Producer 4: Insert Item 1967513926 at 2
Producer 3: Insert Item 1365180540 at 3
Consumer 5: Remove Item 304089172 from 4
Consumer 4: Remove Item 2044897763 from 0
Producer 5: Insert Item 1303455736 at 4
Producer 1: Insert Item 35005211 at 0
Consumer 5: Remove Item 1540383426 from 1
Consumer 5: Remove Item 1967513926 from 2
Producer 2: Insert Item 521595368 at 1
Consumer 2: Remove Item 1365180540 from 3
Consumer 1: Remove Item 1303455736 from 4
Producer 4: Insert Item 294702567 at 2
Producer 4: Insert Item 336465782 at 3
Consumer 1: Remove Item 35005211 from 0
Consumer 2: Remove Item 521595368 from 1
Producer 3: Insert Item 1726956429 at 4
Consumer 1: Remove Item 294702567 from 2
Consumer 3: Remove Item 336465782 from 3
Consumer 2: Remove Item 1726956429 from 4
yczhong@workbench:~/snippets$
```

/* COMP3230 Sample Code: producer-consumer problem using mutex and counting semaphore with BufferSize=5, 5 producer, 5 consumers
File name: T4-pcp-2.c */

```
#define MaxItems 5 // Max
#define BufferSize 5 // S

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;
```

```
void *producer(void *pno)
{
    int item;
    for(int i = 0; i < MaxItems; i++) {
        item = rand(); // Produce an random item
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Producer %d: Insert Item %d at %d\
        in = (in+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}
```

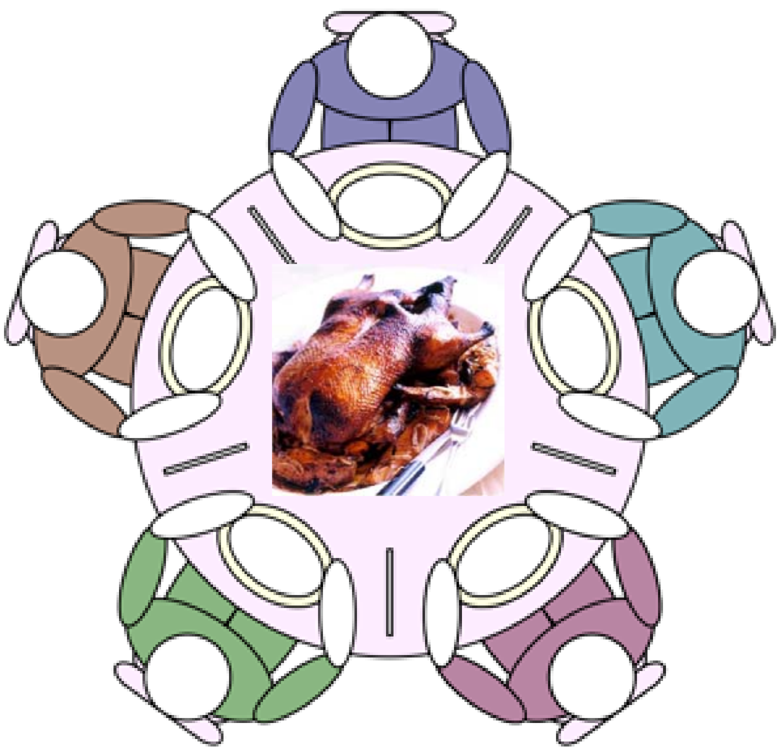
```
void *consumer(void *cno)
{
    for(int i = 0; i < MaxItems; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d: Remove Item %d from
        out = (out+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}
```

```
int main()
{
    pthread_t pro[5],con[5];
    pthread_mutex_init(&mutex, NULL
    sem_init(&empty,0,BufferSize);
    sem_init(&full,0,0);

    int a[5] = {1,2,3,4,5}; //Just used for numbering the producer and consumer
    for(int i = 0; i < 5; i++) { pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]); }
    for(int i = 0; i < 5; i++) { pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]); }
    for(int i = 0; i < 5; i++) {pthread_join(pro[i], NULL); }
    for(int i = 0; i < 5; i++) {pthread_join(con[i], NULL); }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

You can try this on workbench

Dining Philosophers Problem

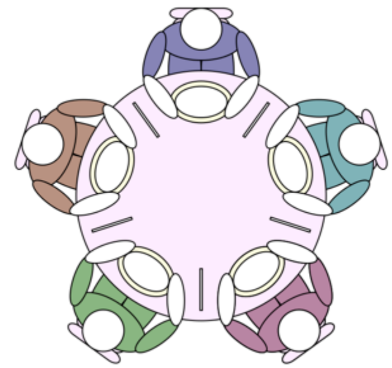
- 5 philosophers seated around a circular table. There is one chopstick between each philosopher. A philosopher can eat if he can pickup the two chopsticks adjacent to him. The major issues are ***deadlock*** and ***starvation***



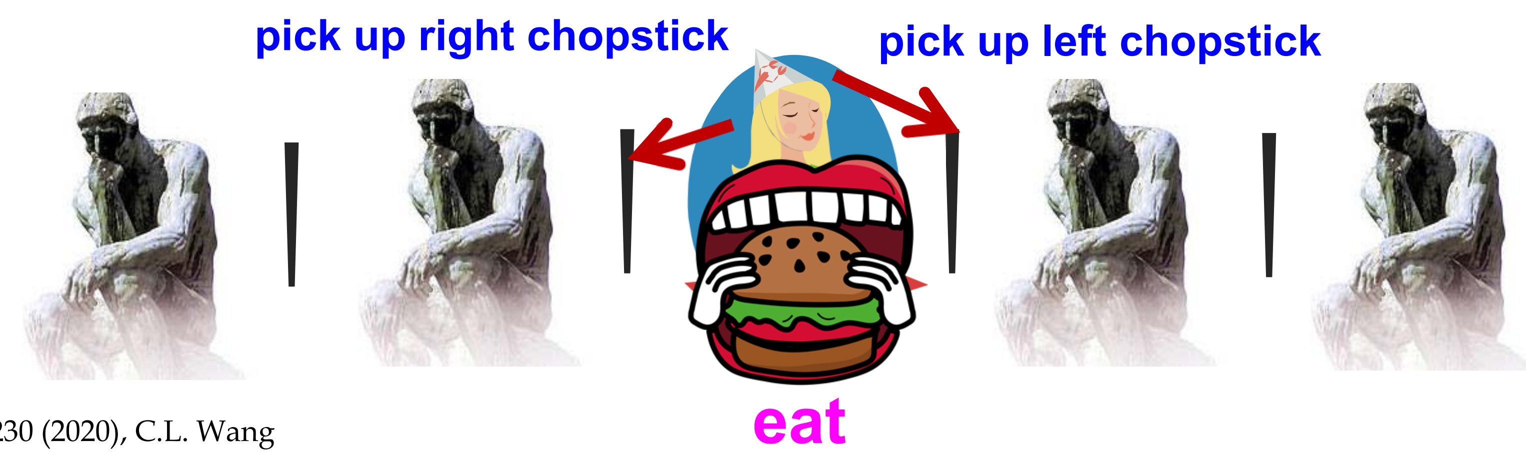
1. **Philosophers sit around a circular table.**
2. **Each philosopher spends his life alternatively <u>thinking</u> and <u>eating</u>.**
3. **Only able to eat <u>when</u> he holds <u>both</u> the left and right chopsticks.**

Solution 1

- **Each philosopher performs repeatedly;**
  - **Think;**
  - **pick up one (right) chopstick;**
  - **pick up another (left);**
  - **eat;**
  - **put down one (right);**
  - **putdown another (left);**

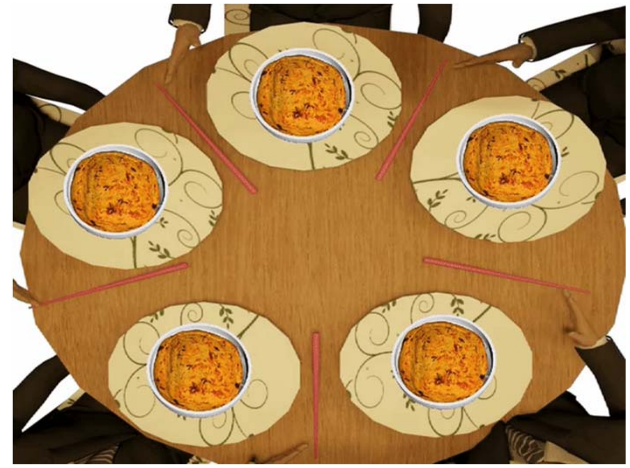


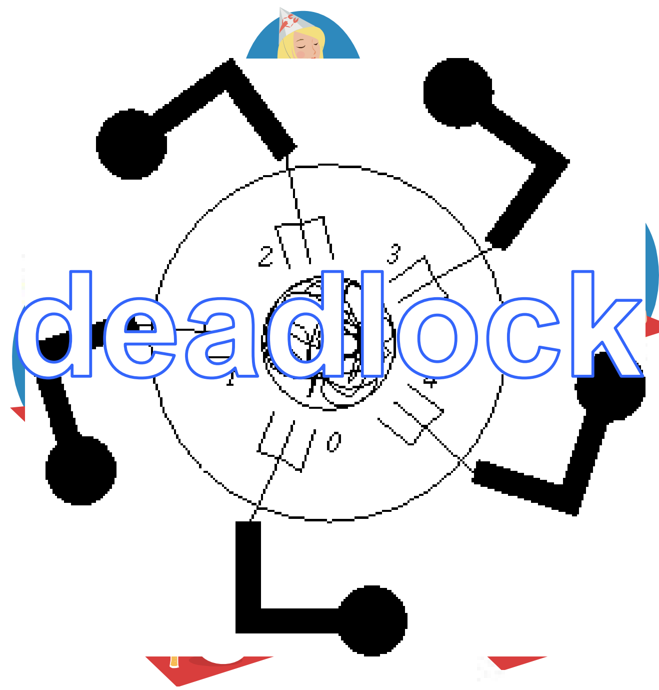
- **Rule: when a philosopher cannot pick up a chopstick, he/she must wait until the chopstick becomes available.**



Problem: Deadlock

- Each philosopher performs repeatedly;
  - Think;
  - pick up one (right) chopstick;
  - pick up another (left);
  - eat;
  - put down one (right);
  - putdown another (left);





Problem: When every philosopher picks up one (left) chopstick at the same time, the **deadlock** occurs.

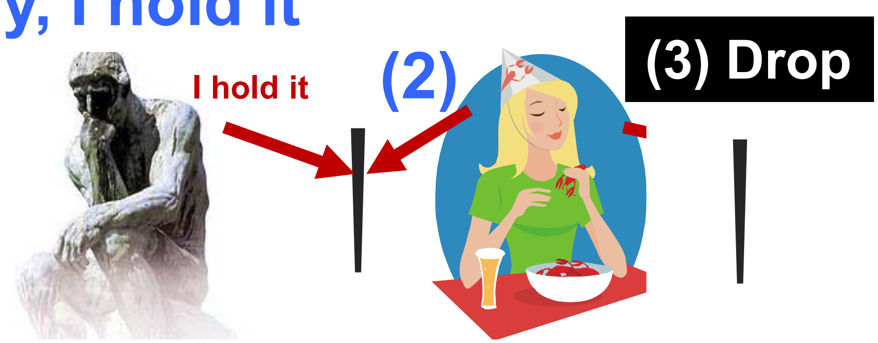
Solution 2

- **Each philosopher performs repeatedly;**

think;
try: **pick up the *left* chopstick;**
if the *right* chopstick is available
then pick up the *right*;
else <u>put down the *left*</u> chopstick ;
wait for some time;
go to try;
end if;
<u>**eat;**</u>

put down *left*;
put down *right*;

sem_trywait()



sem_trywait in Tutorial 4

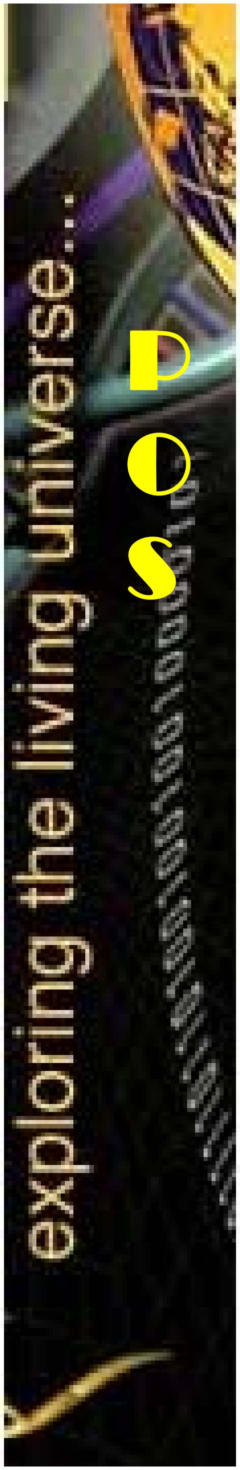
(resolve deadlock, but how about starvation?)

```
while(isFull == 0)
{
        sem_wait(&left_chopstick_sem);
        printf("Philosopher(%d) picks up left chopstick\
        fflush(stdout);
        if(sem_trywait(&right_chopstick_sem) == 0)
        {
                t = time(NULL);
                printf("Philosopher(%d) picks up right chopstick\n", tid);
                fflush(stdout);
                printf("\tPhilosopher(%d): eating... \n", tid);
                fflush(stdout);
                sleep(rand() % 15);  // randomly sle
                printf("\tPhilosopher(%d): Yummy yum
                fflush(stdout);
                isFull = 1;
                sem_post(&right_chopstick_sem);
                printf("Philosopher(%d) puts down right chopstick\n", tid);
                fflush(stdout);
        }
        sem_post(&left_chopstick_sem);
        printf("Philosopher(%d) puts down left chopstick\n", tid);
        fflush(stdout);
}
```

sem_trywait returns 0 if the semaphore value is **currently positive → pick up right chopstick → start eating!**

sem_trywait returns immediately without blocking **if unsuccessful** (right chopstick has been taken) → put down **left chopstick (by sem_post())**

https://github.com/utkusen/dining-philosophers/blob/master/philutku2.c



Solution 2
Problem: Starvation

With little of bad luck, all philosophers could start simultaneously:

1. picking up their left chopstick;
2. seeing right chopstick is not available;
3. <u>putting down</u> their left chopstick;
4. <u>waiting for the same amount of time</u>;
5. picking up their left chopstick simultaneously;

……..

Repeat above steps forever …

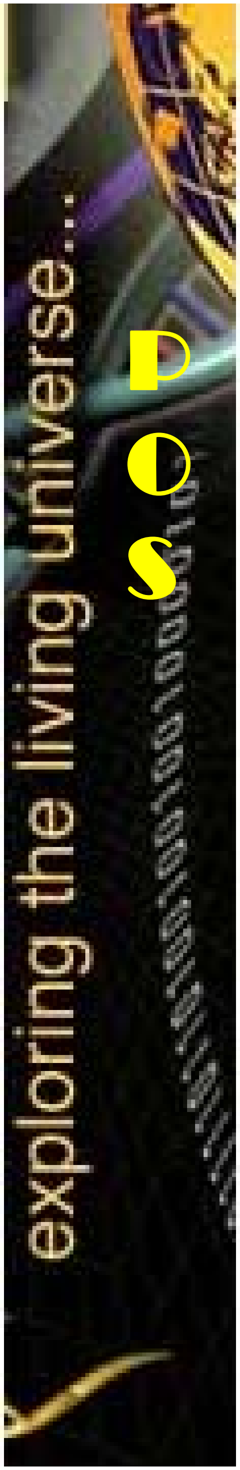
(All threads are still busily running but make no progress!)

“I'm starved”

饿死了

“I'm starved, too”

饿死了



Solution 3

- Each philosopher performs repeatedly:

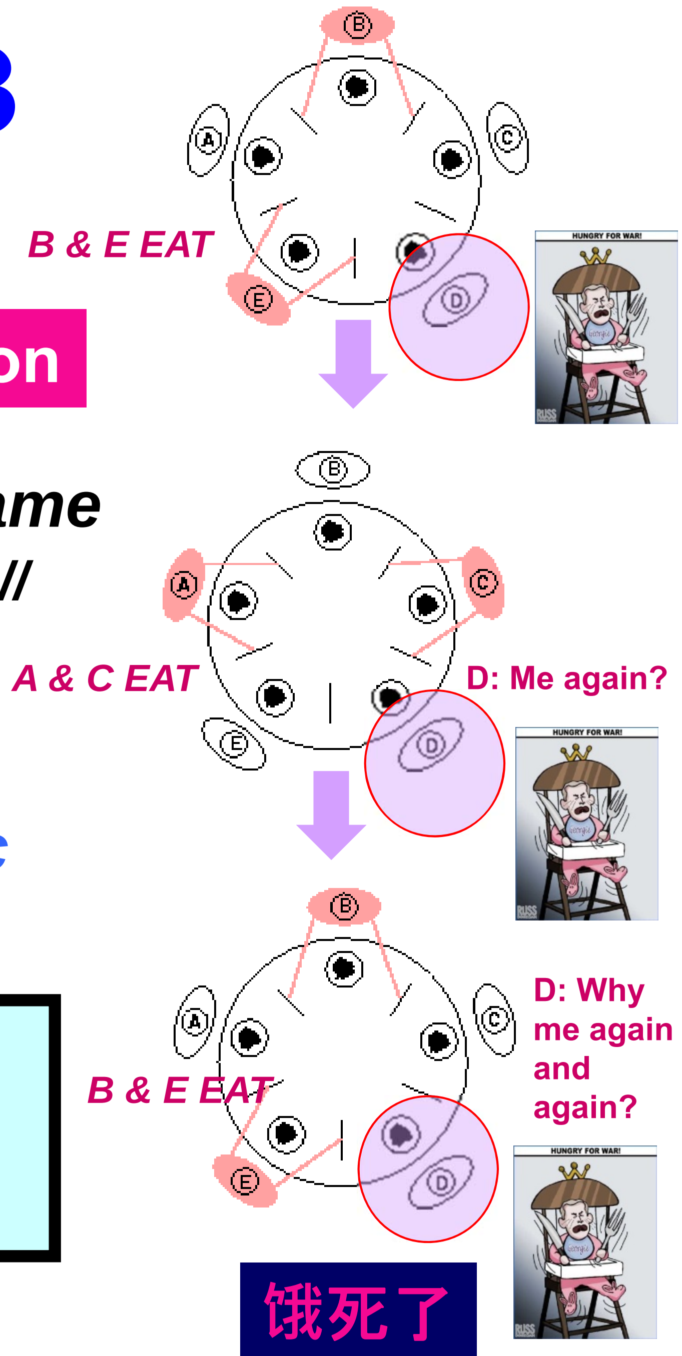
  Atomic operation

  think;

  pick up both chopsticks at the same time; // *make it an Atomic operation* //

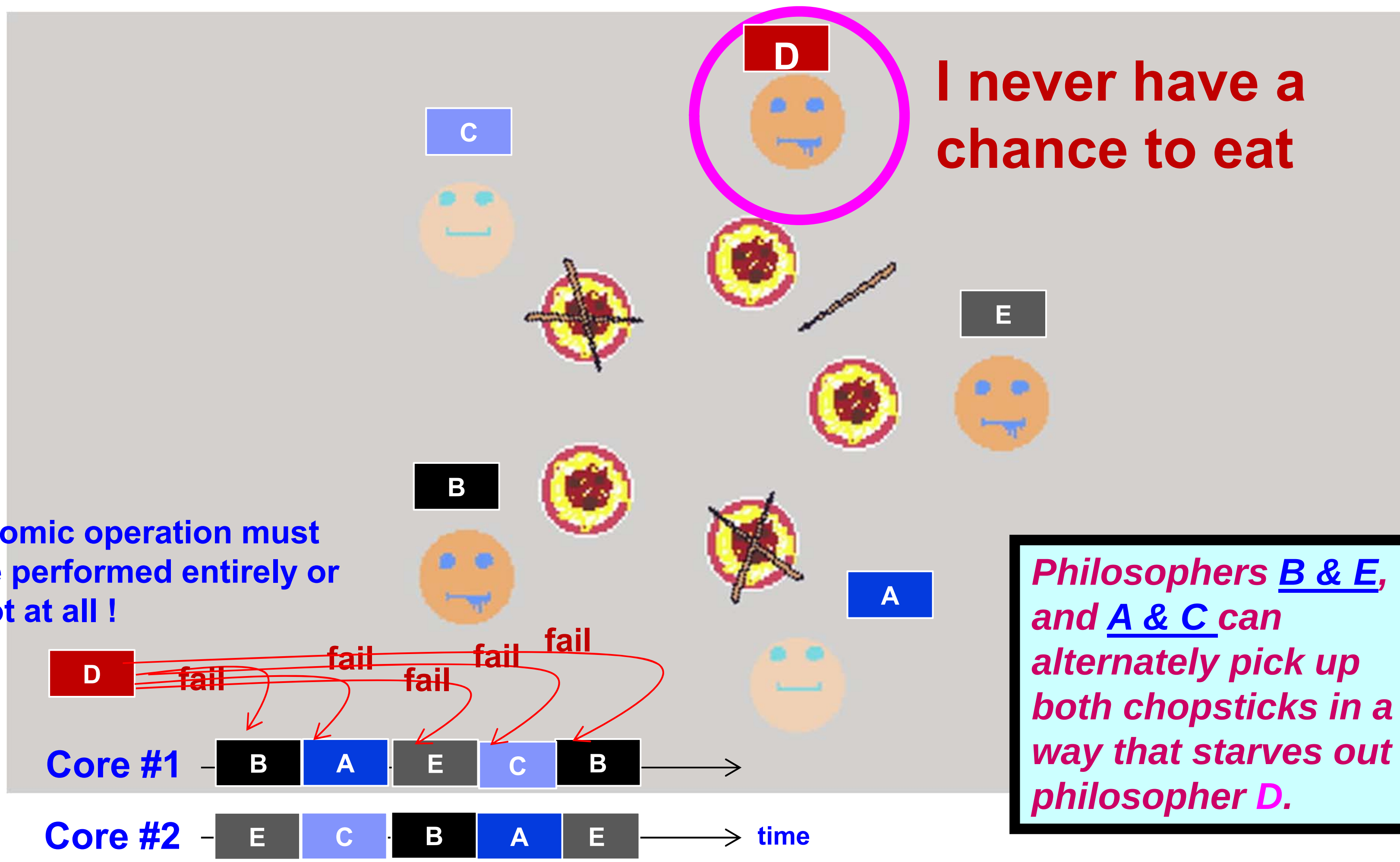
  eat;

  put down both chopsticks at the same time; // *make it an Atomic operation* //



Still Problem: ***Philosophers B & E, and A & C can alternate in a way that starves out philosopher D.***

Solution 3: Starvation (animation)



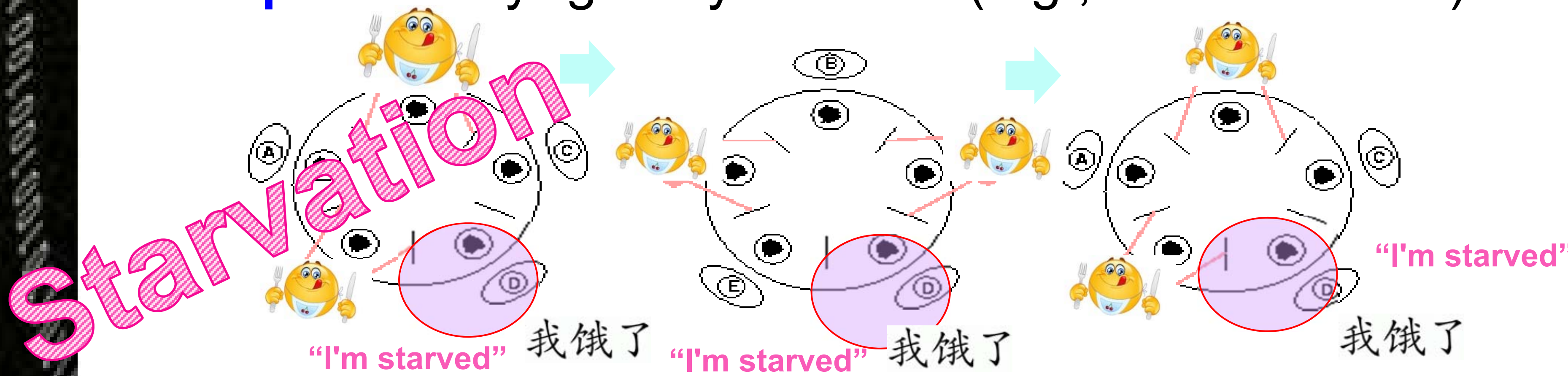
Atomic operation must be performed entirely or not at all !

Philosophers __B & E__, and __A & C__ can alternately pick up both chopsticks in a way that starves out philosopher D.

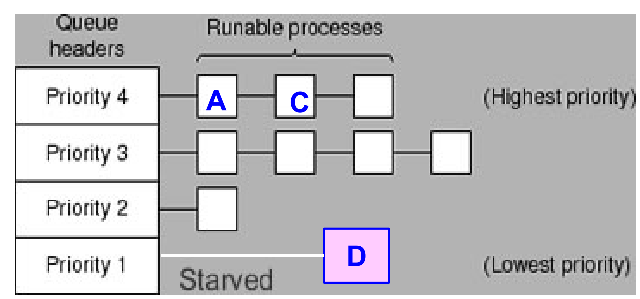
Source: https://multicus.com/posts/dining-philosophers-simulation/

Starvation ≠ Deadlock

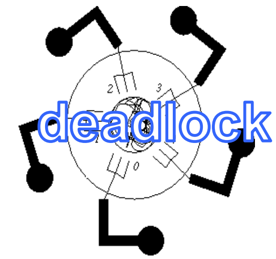
- ***Starvation***, **also called** ***Indefinitely postponed.***
  - A situation in which a runnable process is overlooked indefinitely **by the scheduler**; although it is able to proceed, it is **never chosen**. This happens when shared resources are made unavailable **for long periods** by "greedy" threads (e.g., **A+C** and **B+E**).



Example: Philosophers B & E, and A & C can alternate in a way that starves out philosopher D (i.e., D is Indefinitely postponed).



Deadlock vs. Starvation



- **Deadlock** refers to the situation when threads/processes **are stuck in *circular waiting* for the resources**.
- **Starvation** occurs when a thread/process waits for a resource **indefinitely**.
  - E.g., due to the CPU scheduling algorithms (low priority processes not being scheduled to run) or your S.L implementation (e.g., using **last-in-first-out**)
- **Deadlock implies starvation but starvation does not imply deadlock.**

Solution 4: Break circular waiting by restricting resource access

- Each **odd-numbered** ph performs repeatedly:

  think;

  *pick up the **left***

  *pick up the **right**;*

  eat

  put down the left

  put down the right;

- Each **even-numbered** ph performs repeatedly:

  think;

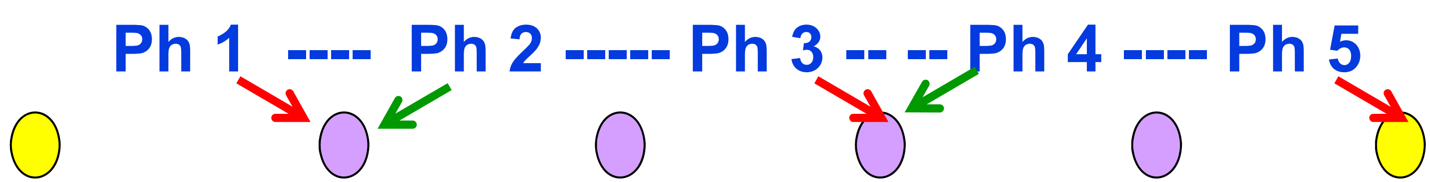
  *pick up the **right***

  *pick up the **left**;*

  eat

  put down the right

  put down the left;



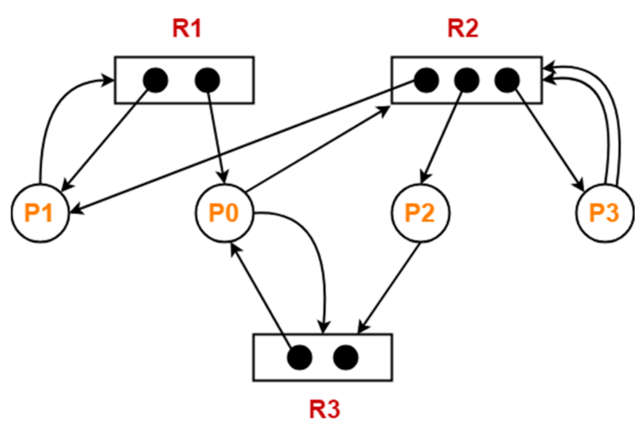
OK. No deadlock, No starvation

exploring the living universe...

OS

A Formal Model of Deadlock

Resource Allocation Graphs (RAG)

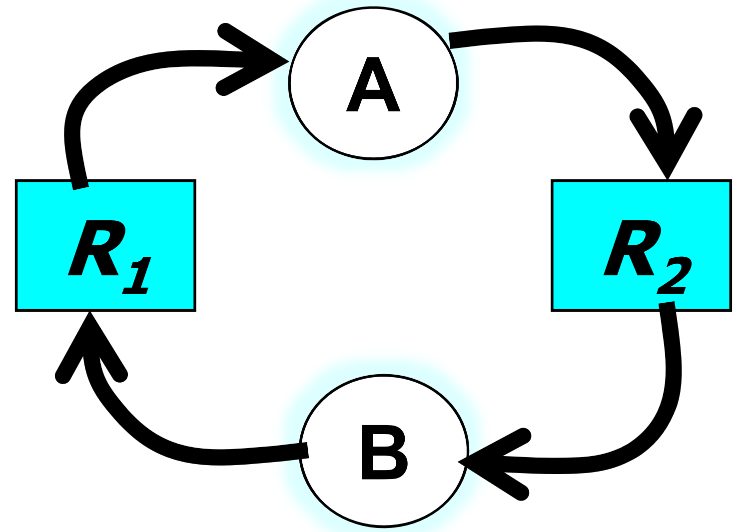


Resource Allocation Graphs (RAG)

- To represent resource allocation states

R_j is allocated to P1

P0 is requesting R_i



resource process **Deadlock State (a cycle)**

Deadlock description in terms of resource allocation graph (V, E), with V partitioned into two types of vertices:

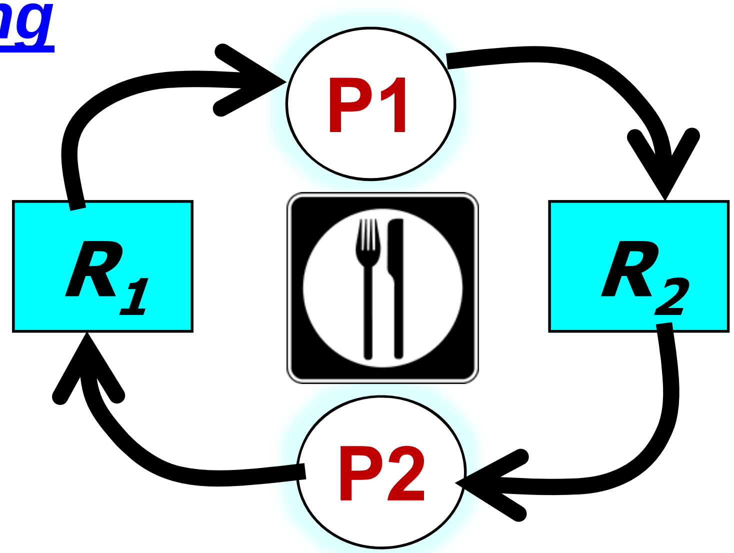
$P = P_1, P_2, P_3, .. P_n$ - set of active processes, and
$R = R_1, R_2, .. R_m$ - set of all resource types.

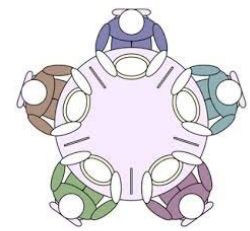
A directed edge $P_i \rightarrow R_j$ is called **request edge**.
A directed edge $R_j \rightarrow P_i$ is called **assignment edge**.

Deadlock Detection in RAG

- ***A situation in which two or more processes are unable to proceed because <u>each is waiting for one of the others</u> to do something.*** **<u>An example:</u>**
  - **Process P1** requires additional resource **R2** and is in possession of resource **R1**,
  - **P2** requires additional resource **R1** and is in possession of **R2**; neither process can continue

<u>Circular Waiting</u>

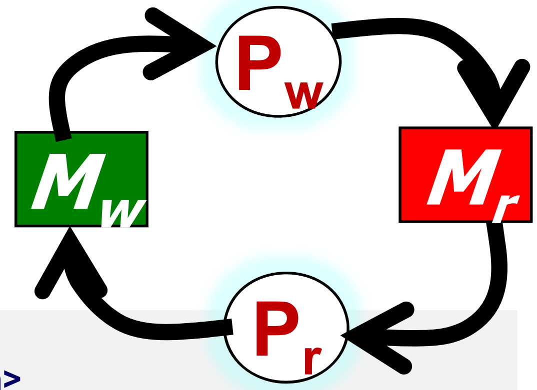




exploring the living universe...

DOS

```
/* COMP3230 Deadlock
   File name deadlock.c
*/
```




```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
pthread_mutex_t read_mutex;
pthread_mutex_t write_mutex;

void * write(void *temp) {
char *ret;
FILE *file1;
char *str;
pthread_mutex_lock(&write_mutex);        Mw
sleep(5);
pthread_mutex_lock(&read_mutex);         Mr
printf("\nFile locked, please enter the message \n");
str=(char *)malloc(10*sizeof(char));
file1=fopen("temp","w");
scanf("%s",str);
fprintf(file1,"%s",str);
fclose(file1);
pthread_mutex_unlock(&read_mutex);
pthread_mutex_unlock(&write_mutex);
printf("\nUnlocked the file you can read it now \n");
return ret;
}
```

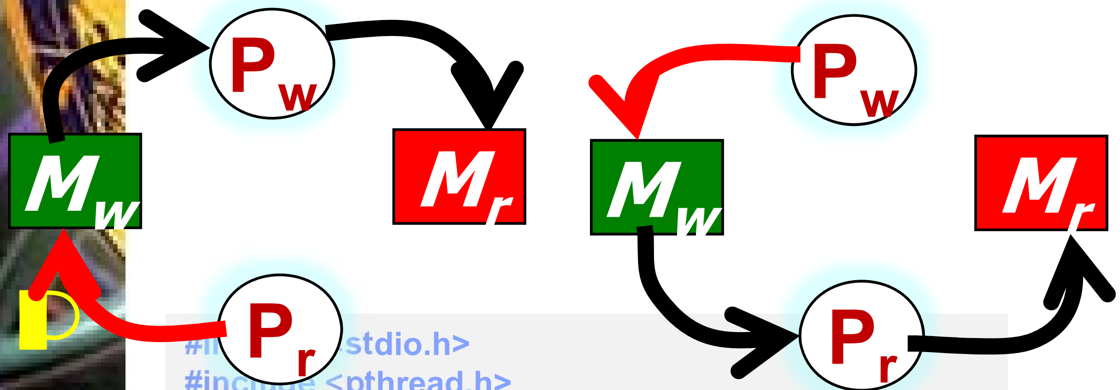
Deadlock!

```
void * read(void *temp) {
char *ret;
FILE *file1;
char *str;
pthread_mutex_lock(&read_mutex);         Mr
sleep(5);
pthread_mutex_lock(&write_mutex);        Mw
printf("\n Opening file \n");
file1=fopen("temp","r");
str=(char *)malloc(10*sizeof(char));
fscanf(file1,"%s",str);
printf("\n Message from file is %s \n",str);

fclose(file1);

pthread_mutex_unlock(&write_mutex);
pthread_mutex_unlock(&read_mutex);
return ret;
}
```

```
yczhong@workbench:~/snippets$ gcc deadlock.c -o deadlock -pthread
yczhong@workbench:~/snippets$ ./deadlock

^C
```

```
main() {

pthread_t thread_id,thread_id1;
pthread_attr_t attr;
int ret;
void *res;
ret=pthread_create(&thread_id,NULL,&write,NULL);
ret=pthread_create(&thread_id1,NULL,&read,NULL);
printf("\n Created thread");
pthread_join(thread_id,&res);
pthread_join(thread_id1,&res);
}
```

There will be no output as both threads endup in a deadlock state. To get out of the execution use “CTRL + c".

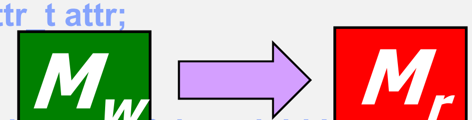



```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
pthread_mutex_t read_mutex;
pthread_mutex_t write_mutex;

void * write(void *temp) {
char *ret;
FILE *file1;
char *str;
pthread_mutex_lock(&write_mutex);
sleep(5);
pthread_mutex_lock(&read_mutex);
printf("\nFile locked, please enter the message \n");
str=(char *)malloc(10*sizeof(char));
file1=fopen("temp","w");
scanf("%s",str);
fprintf(file1,"%s",str);
fclose(file1);
pthread_mutex_unlock(&read_mutex);
pthread_mutex_unlock(&write_mutex);
printf("\nUnlocked the file you can read it now \n");
return ret;
}
```

Enforce resource access order: every thread should lock write_mutex first and then try to lock read_mutex

```c
void * read(void *temp) {
char *ret;
FILE *file1;
char *str;
pthread_mutex_lock(&write_mutex);
sleep(5);
pthread_mutex_lock(&read_mutex);
printf("\n Opening file \n");
file1=fopen("temp","r");
str=(char *)malloc(10*sizeof(char));
fscanf(file1,"%s",str);
printf("\n Message from file is %s \n",str);

fclose(file1);

pthread_mutex_unlock(&read_mutex);
pthread_mutex_unlock(&write_mutex);
return ret;
}

main() {

pthread_t thread_id,thread_id1;
pthread_attr_t attr;
int ret;
void *res;
ret=pthread_create(&thread_id,NULL,&write,NULL);
ret=pthread_create(&thread_id1,NULL,&read,NULL);
printf("\n Created thread");
pthread_join(thread_id,&res);
pthread_join(thread_id1,&res);
}
```

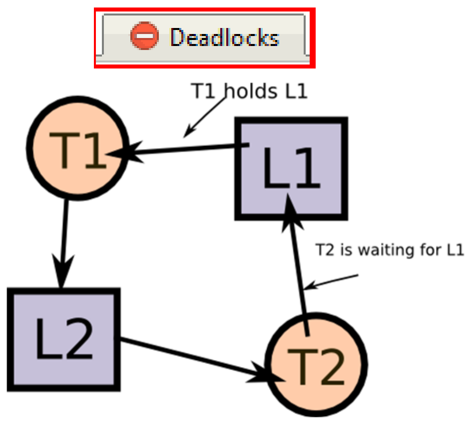
No Deadlock!

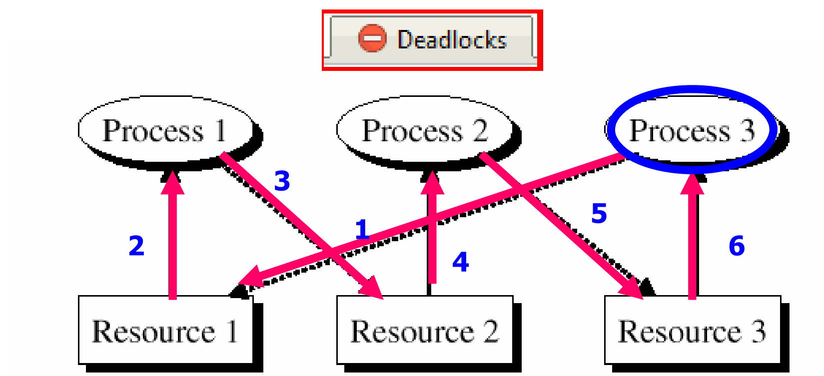


Enforce Access Order

RAG and Deadlock

- If each resource type has **exactly one instance**, a cycle in a RAG, <u>**if and only if**</u> deadlock occurs.



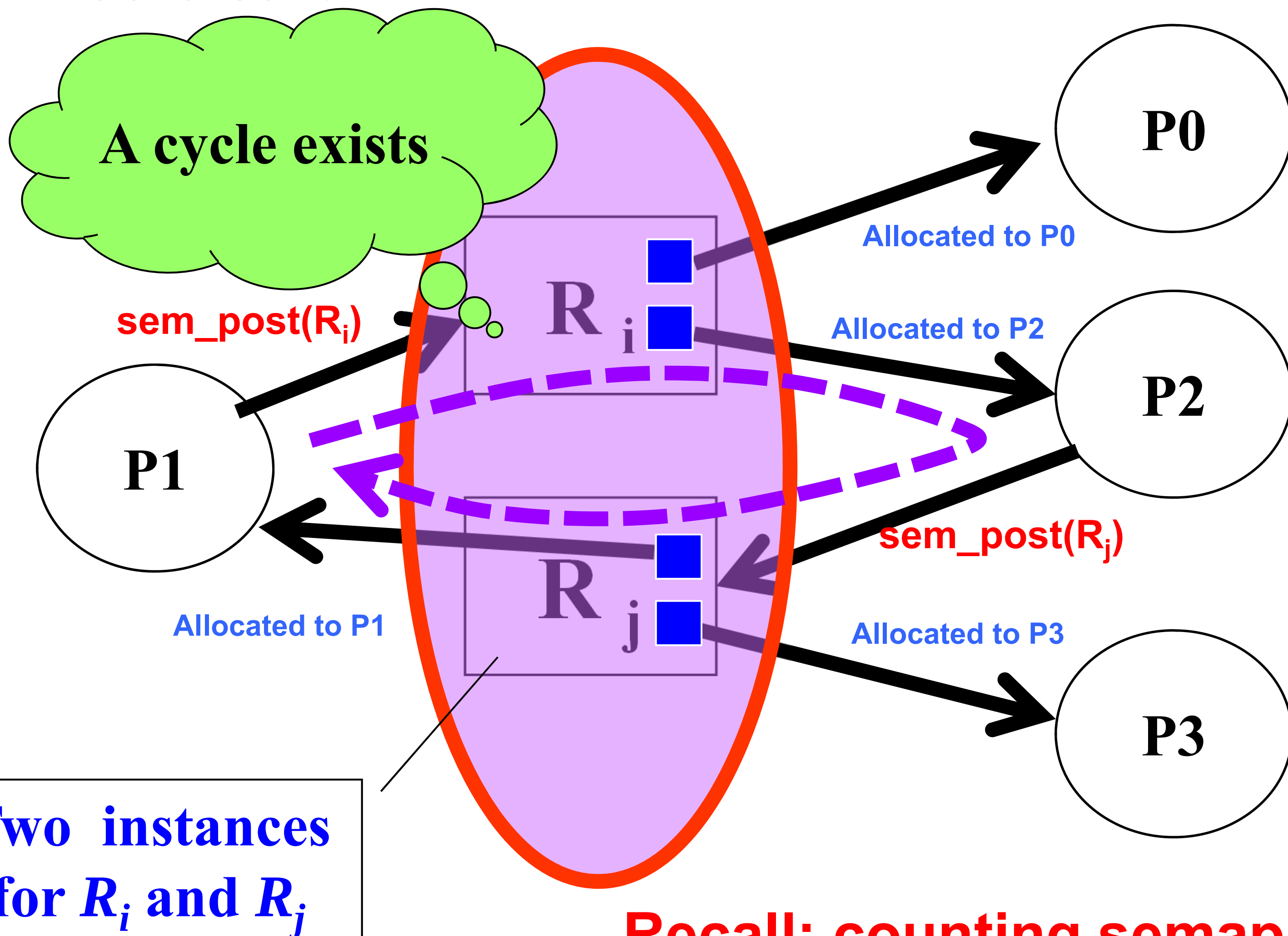


(Only one instance per resource)

- If each resource type has **multiple instances**:
  - **A cycle in RAG is a necessary, but not sufficient, condition for deadlock, i.e.:**
    - **If** a deadlock occurs, you must find a cycle in a RAG.
    - If a cycle found in a RAG, there is not necessary a deadlock!!

Resource with multiple instance

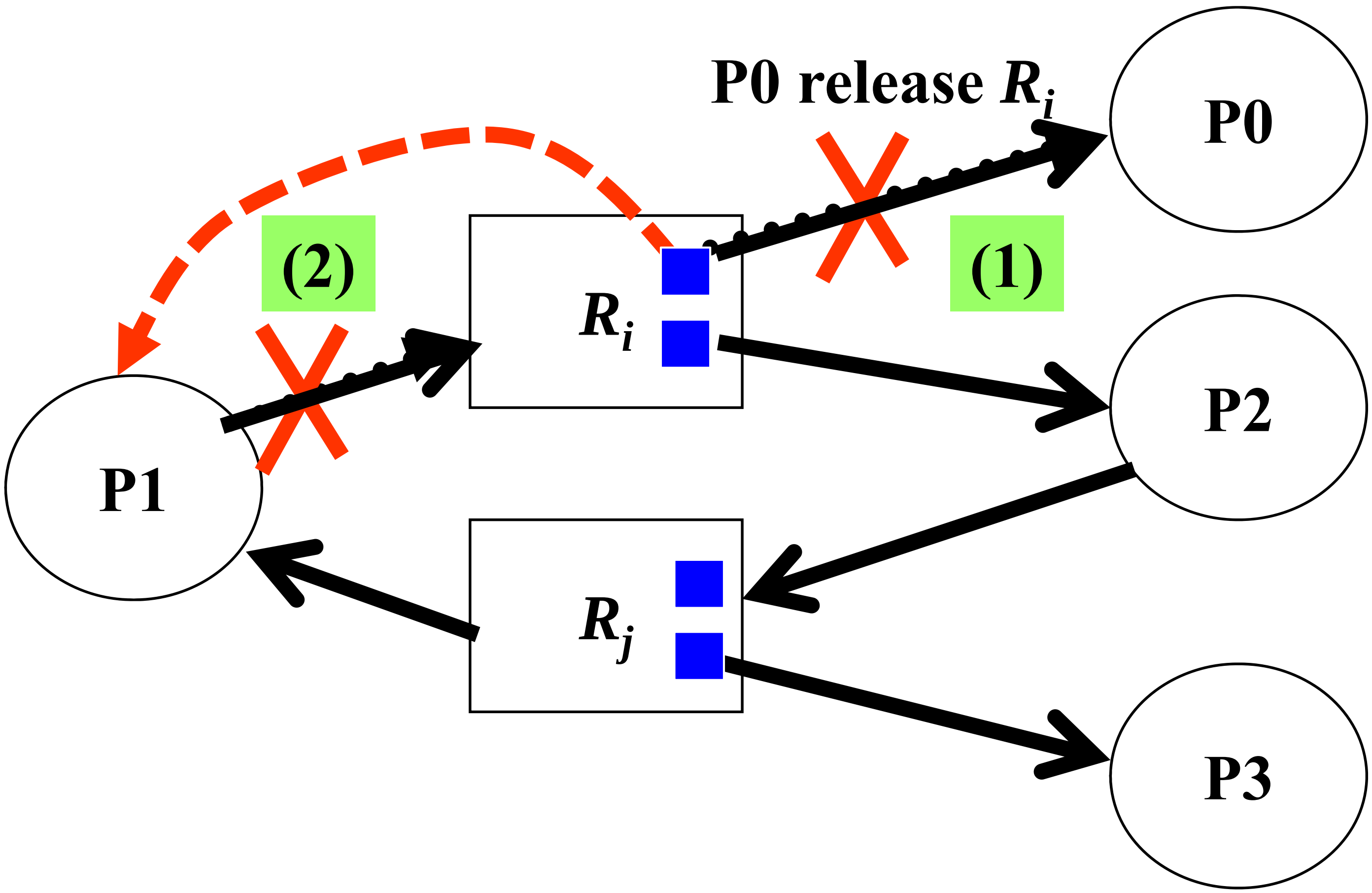
- Example: a `cycle` found in a RAG, **but is there a deadlock?**



Recall: counting semaphores

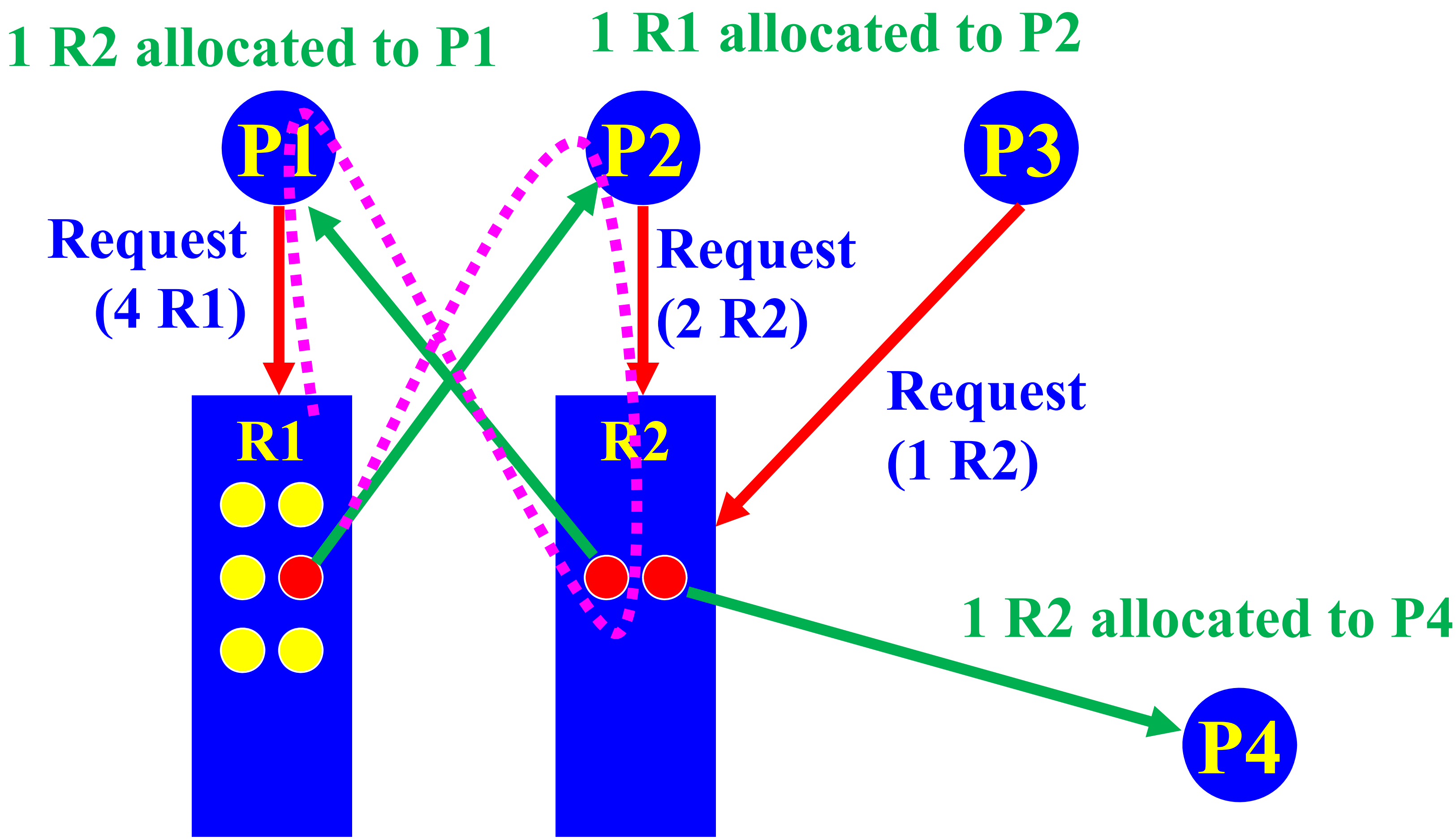
Why no deadlock?

- Why there is no deadlock ?



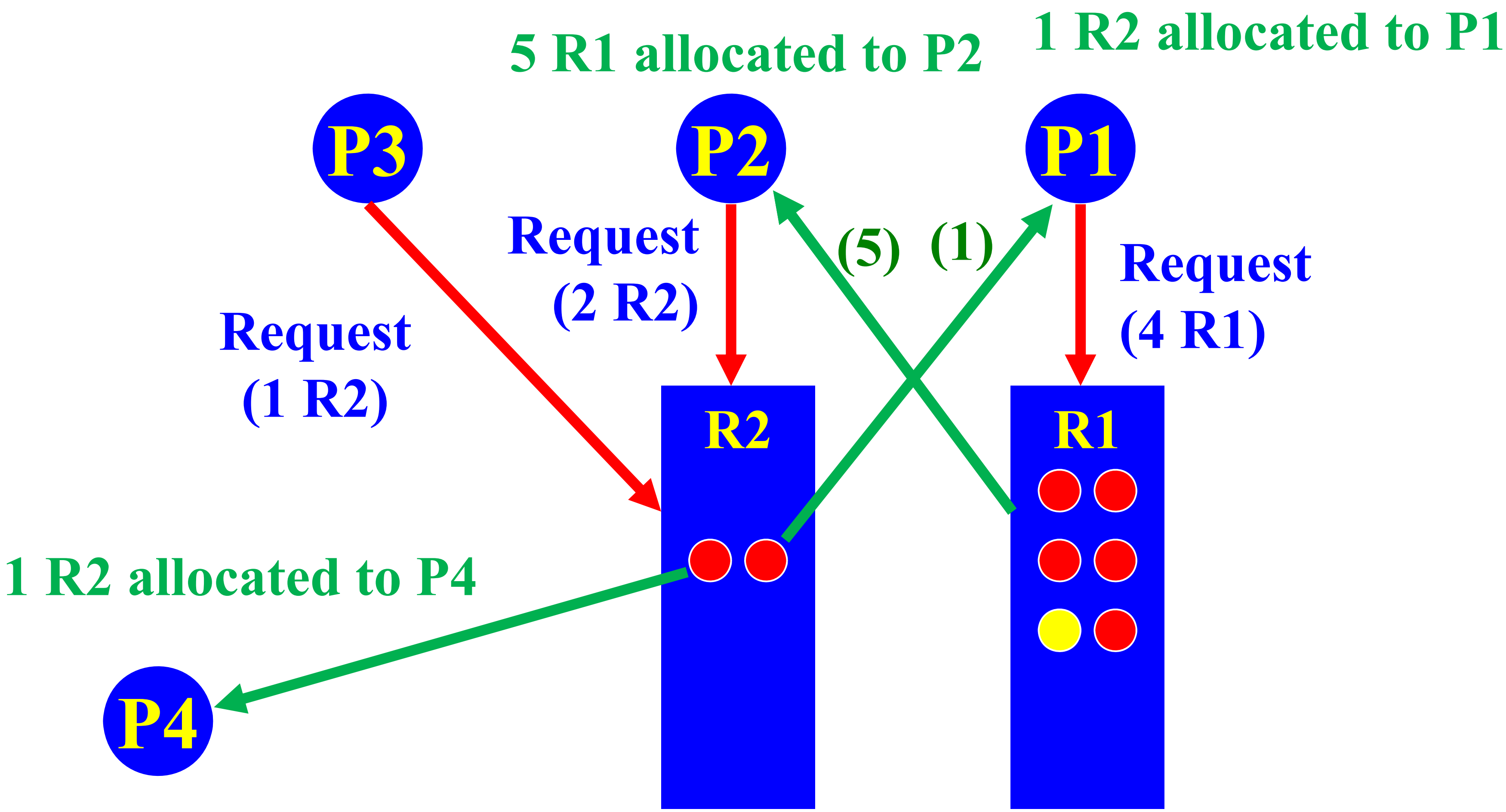
P0 releases one R_i, then the released R_i instance can be assigned to P1
P3 eventually finished and R_j can be assigned to P1 → P1 can complete

If the graph is "reducible"→ System not deadlocked



Can you find a cycle above? YES. But is it a deadlock?

Not reducible → system deadlocked



Can you find a cycle above? Try it yourself!

Reference (YouTube)

- **Pthreads: Introduction**
  - https://www.youtube.com/watch?v=ynCc-v0K-do
  - https://www.youtube.com/watch?v=GXXE42bkqQk&list=RDCMUCMnSzocL2Z5hhOJglLQF9oA&index=2
- **Mutex Synchronization in Linux with Pthreads**
  - https://www.youtube.com/watch?v=GXXE42bkqQk
- **Bounded Buffer Demonstration**
  - https://www.youtube.com/watch?v=RWyv14K1DpE
  - https://www.youtube.com/watch?v=NuvAjMk9bZ8
- **Dining philosopher problem**
  - https://www.youtube.com/watch?v=c99S9vkuN24
  - https://www.youtube.com/watch?v=0b0tEmRQJx0
  - https://www.youtube.com/watch?v=wD9nM7loabA
  - https://www.youtube.com/watch?v=rCiQc3ife90

Pthreads Sample Programs

- **A simple pthreads program**
  - http://gauss.ececs.uc.edu/Courses/c4029/code/pthreads/01-thread.c
- **Multithreading in C**
  - http://www.geeksforgeeks.org/multithreading-c-2/
- ***Multithreading in C, POSIX style***
  - http://softpixel.com/~cwright/programming/threads/threads.c.php
- **pthread_mutex_lock:**
  - http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html#thread_mutex_whatis
- **Pthreads and Semaphores**
  - http://condor.depaul.edu/glancast/374class/docs/pthreads.html
- **Implementing Semaphores Using pthreads**
  - **http://www.cs.ucsb.edu/~rich/class/cs170/notes/Semaphores/**

Past Exam Questions

- (X) **All the shared variables** modified within a critical section are **not readable** by other threads until the thread leaves the critical section.
- (X) Semaphores (sem_wait/post()) are mechanisms provided by OS kernels, while pthread_mutex_lock/unlock() are library functions built **on top of** sem_wait() and sem_post().
- (X) An ***atomic*** instruction, such as **testAndSet,** can ensure mutual exclusion **by itself alone**, which are more powerful than those pure software-based solutions.
- (O) Critical section is **a piece of code** that only one thread can execute at a time, otherwise a race condition could happen.
- (X) An example of critical section is the code segment executed **within** the **test_and_set** operation.
- (O) A ***race condition*** is a situation in which more than one process or thread access a ***shared resource*** concurrently, and the result depends on the order of execution. (Sure, this is the definition)
- (O) A good solution to the critical section problem must satisfy three conditions: mutual exclusion, progress and bounded waiting.

6. (   ) Which of the following statements is **INCORRECT**? (2)

(1) A binary semaphore is equivalent to a lock.
(2) A semaphore implementation must use busy waiting (i.e. spinlock).
(3) A counting semaphore is initialized to an integer value k > 1.
(4) When a semaphore is probed, the testing and decrementing operations must be done atomically.
(5) None of the above (i.e., (1)-(4) are all incorrect).

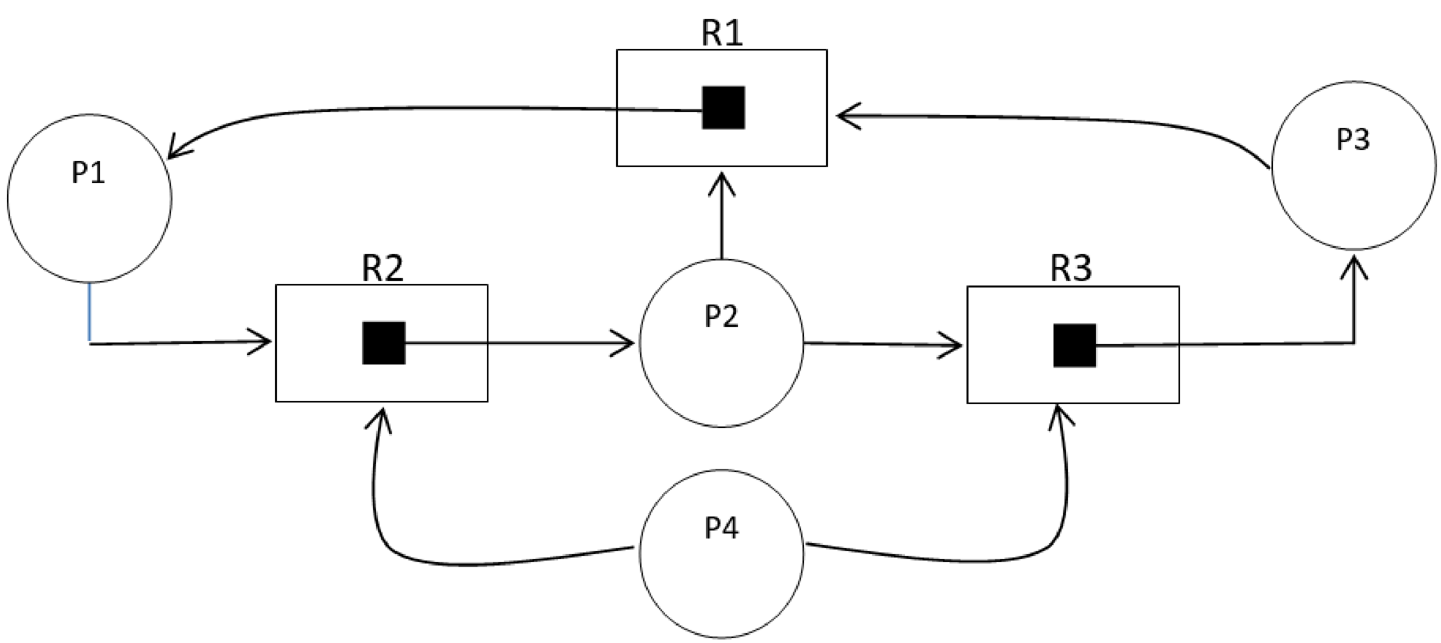
7. (   ) One solution to the Dining Philosophers problem which avoids deadlock is: (**4**)

(1) Non-preemptive scheduling.
(2) Ensuring that all philosophers pick up their left fork before they pick up their right fork.
(3) Ensuring that all philosophers pick up their right fork before they pick up their left fork.
(4) Ensuring that odd philosophers pick up their left fork before they pick up their right fork and even philosophers pick up their right fork before they pick up their left fork.
(5) None of the above (i.e., (1)-(4) are all incorrect.)

Past Exam Questions

Problem 4. Deadlock (10%)

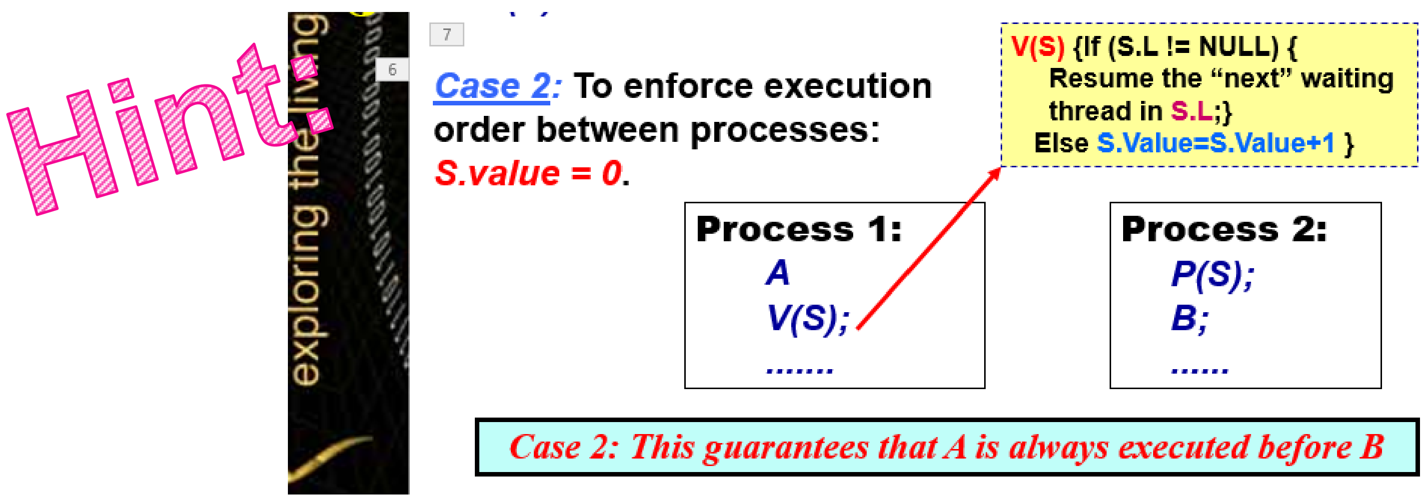
(a) Given the following resource allocation diagram. Is it in a deadlocked state? (**2%**)
(b) If one more instance of resource **R1** is added, will it be in a deadlocked state? Explain why? (**4%**) [**Note:** You need to show the graph reduction steps to justify your answer.]
(c) What if the newly added instance is **R3**? Is it in a deadlocked state? Explain why? (**4%**)



Problem 3. Process Synchronization (10%)

There are four processes (A, B, C, and D) running in a system. Process A should finish before process B starts, and process B should finish before either of processes C or D start. The *P()* and *V()* operations are given below. Show how these processes can use **TWO** *semaphores* to provide the necessary synchronization. Note that you are required to set the initial values of the two semaphores.

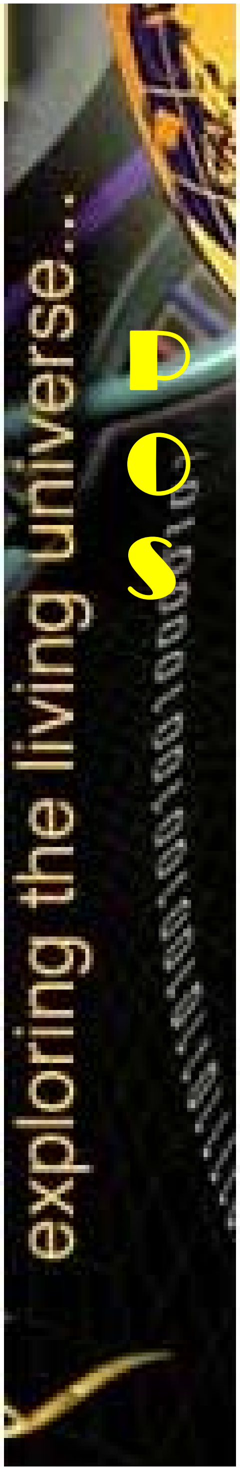
```
P(S) {   S.value--;
         if (S.value<0) {
               add this process to the waiting queue
               Sleep( ); }}

V(S) {   S.value++;
         if (S.value <= 0) {
               remove a process P from the waiting queue;
               Wakeup (P); }}
```



Self-test Questions

- Can a thread acquire more than one lock (Mutex)?
  - Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.
- Is it necessary that a thread must block always when resource is not available?
  - **Not necessary.** If the design is sure 'what has to be done when resource is not available', the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.
  - For example POSIX pthread_mutex_trylock() API. When mutex is not available the function returns immediately whereas the API pthread_mutex_lock() blocks the thread till resource is available.

Extra Slides



Linux: Counting Semaphores
(kernel code)

- ***A counting semaphore may be acquired 'n' times before sleeping.***

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

down=P()

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

down - acquire the semaphore. <u>Use of this function is deprecated,</u> use down_interruptible() or down_killable() instead

up()=V()

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

up - release the semaphore (Note: up() may be called from any context and even by tasks which have never called down().)

Source code: https://elixir.bootlin.com/linux/v4.3/source/kernel/locking/semaphore.c

Counting Semaphores implementation in Linux

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

acquire the semaphore

Critical section

*A counting semaphore may be acquired 'n' times before sleeping. The **count** variable represents how many more tasks can acquire this semaphore*

try to acquire the semaphore, without waiting

```
int down_trylock(struct semaphore *sem)
{
        unsigned long flags;
        int count;

        raw_spin_lock_irqsave(&sem->lock, flags);
        count = sem->count - 1;
        if (likely(count >= 0))
                sem->count = count;
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return (count < 0);
}
EXPORT_SYMBOL(down_trylock);
```

Critical section

disables interrupts

enables interrupts

exploring the living universe...

DOS

Linux: Counting Semaphores

```c
int down_interruptible(struct semaphore *sem)
{
        unsigned long flags;
        int result = 0;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                result = __down_interruptible(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return result;
}
```

***down_interruptible* - acquire the semaphore unless interrupted**

```c
int down_trylock(struct semaphore *sem)
{
        unsigned long flags;
        int count;

        raw_spin_lock_irqsave(&sem->lock, flags);
        count = sem->count - 1;
        if (likely(count >= 0))
                sem->count = count;
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return (count < 0);
}
```

***down_trylock* - try to acquire the semaphore, without waiting**

```c
int down_killable(struct semaphore *sem)
{
        unsigned long flags;
        int result = 0;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                result = __down_killable(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return result;
}
```

***down_killable* - acquire the semaphore unless killed**

```c
int down_timeout(struct semaphore *sem, long timeout)
{
        unsigned long flags;
        int result = 0;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                result = __down_timeout(sem, timeout);
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return result;
}
```

***down_timeout* - acquire the semaphore within a specified time**

futex() system call

futex - fast user-space locking

The futex() system call provides a method for waiting until a certain condition becomes true. Using futex, the majority of the synchronization operations are performed in user space. No context switching into the kernel.

If **sem_value == value (=1)**, not locked, sem_value ← (**value-1=0**), **return true**. Otherwise, **value** ← sem_value, return false (keep looping)

```
int sem_wait(sem_t *sem)
{
      unsigned value = 1;
      while (!atomic_compare_exchange_....(&sem->value, &value, value - 1,...))
      {
            if (value == 0) { // Note: value ← sem_value (check if sem_value =0)
               futex_wait(&sem->value, 0, NULL);
               value = 1; }       }
      return thrd_success; }
```

futex_wait context switch into the kernel → puts the calling process to sleep until the mutex is released.

```
void sem_post(sem_t *sem)
{
      atomic_fetch_add_explicit(&sem->value, 1, .....);
      futex_signal(&sem->value);
      return thrd_success;
}
```

Futex operations: FUTEX_WAIT

- **int futex (int * uaddr, int futex_op, int val, const struct timespec * timeout, int * uaddr2, int val3);**
- **futex_op = FUTEX_WAIT :**
  - tests if the value at the futex word pointed to by the address ***uaddr*** still contains the expected value ***val***, and if so, then **sleeps** (hang the process on the wait queue corresponding to uaddr).
  - The load of the value of the futex word is an **atomic memory access**
  - If the timeout is not NULL, the structure it points to specifies a timeout for the wait.

Futex operations: FUTEX_WAKE

- futex(***uaddr***, **FUTEX_WAKE**, ***val***, 0, 0, 0);
  - The arguments timeout, uaddr2, and val3 are ignored.
- This operation **wakes** at most ***val*** of the waiters that are waiting (e.g., inside FUTEX_WAIT) on the futex word at the address ***uaddr***.
- Most commonly, ***val*** is specified as either **1** (wake up a single waiter) or **INT_MAX** (wake up all waiters).