CMPT 300
Operating System I
Synchronization Tools

Dr. Hazra Imran

Summer 2022

Admin

A2 regrading request due date is July 6.

A3 submission date is July 11.

A2 reflection

I did not learn much on things relating to this course to be completely honest. This felt more like a C general assignment than something relating to this course.

I would say this assignment was rather challenging and it tested me more on my C language skills than testing me out on my Operating Systems knowledge.

I learned a lot through this assignment. I got a general idea of how shells work. I learned how to use threads and pipes appropriately. Also learned about I/O redirection, and although I had trouble with the implementation of the system call, I understood how they worked.

I learned about handling child processes with fork() and wait(), managing arrays of structs, and splitting an array of strings into tokens. I also learned how to use the tm struct to display the current time in a formatted way.

This assignment was very fun and had a lot of interesting elements to learn from. It was a very satisfactory experience as we got to build our own c shell.

Clicker

's' can be initialized to any integer value.

(A) True

(B) False

$s < 0$

blocked on s

non-negative integer value

Readers-Writers Solution

Shared data:

- Data set
- Binary Semaphore **rw_mutex** initialized to 1
  - 1 = no readers/writers;
  - 0 = a writer or some number of readers
- Integer **read_count** initialized to 0
  - Number of processes actively reading the data set
- Binary Semaphore **mutex** initialized to 1
  - Protects read_count from being accessed/modified by more than one process

Solution Case 1 : R-W

read_count= 0 Semaphore rw_mutex =1 Semaphore mutex =1

Reader

```
do {
wait(mutex);
read_count++; // one more reader
if (read_count == 1)
   wait(rw_mutex); // First reader
 signal(mutex);
...
/* reading is performed */
...
wait(mutex);

read_count--; // one less reader
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

Writer

```
do {
wait(rw_mutex);
...
/* writing is performed */
...
signal(rw_mutex);
} while (true);
```

- Reader 1 comes first.
- Mutex = ~~1~~ 0
- RC++ = we have one reader. That's why its incremented
- If (read_count==1) → True. So now,rw_mutex is 0.
- Mutex =1 (signal(mutex))
- Reader 1 in cs
- Now, writer comes. The entry code of writer will try to decrement rw_mutex(which is 0). These are binary semaphores so not possible. So, Writer will get block.

Solution Case 2 : W-R

read_count= 0 Semaphore rw_mutex =1 Semaphore mutex =1

```
Reader
do {
wait(mutex);
read_count++; // one more reader
if (read_count == 1)
   wait(rw_mutex); // First reader
 signal(mutex);
...
/* reading is performed */
...
wait(mutex);

read_count--; // one less reader
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

```
Writer

do {
wait(rw_mutex);
...
/* writing is performed */
...
signal(rw_mutex);
} while (true);
```

W1 comes first.
rw_mutex = ~~1~~ 0
Writer in CS
Ques Can reader enter in CS?
Lets check:
mutex = ~~1~~ 0
RC++ = we have one reader. That's why its incremented
If (read_count==1) → True. So now, it will try to decrement rw_mutex (which is 0).These are binary semaphores so not possible. So, Reader will get block.

Solution Case 3 : W-W

read_count= 0 Semaphore rw_mutex =1 Semaphore mutex =1

Reader

```
do {
wait(mutex);
read_count++; // one more reader
if (read_count == 1)
   wait(rw_mutex); // First reader
 signal(mutex);
...
/* reading is performed */
...
wait(mutex);

read_count--; // one less reader
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

Writer

```
do {
  wait(rw_mutex);
  ...
  /* writing is performed */
  ...
  signal(rw_mutex);
} while (true);
```

W1 comes first.
rw_mutex = ~~1~~ 0
Writer1 in CS
Ques Can Writer 2 enter in CS?
Lets check:
The entry code of writer2 will try to decrement rw_mutex(which is 0). These are binary semaphores so not possible. So, Writer2 will get block.

Solution Case 4 : R-R

read_count= 0 Semaphore rw_mutex =1 Semaphore mutex =1

Reader

```
do {
wait(mutex);
read_count++; // one more reader
if (read_count == 1)
   wait(rw_mutex); // First reader
 signal(mutex);
...
/* reading is performed */
...
wait(mutex);

read_count--; // one less reader
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

Writer

```
do {
wait(rw_mutex);
...
/* writing is performed */
...
signal(rw_mutex);
} while (true);
```

Reader1 comes first.
Mutex = ~~1~~ 0
RC++ = we have one reader. That's why its incremented
If (read_count==1) → True. So now,rw_mutex is 0.
Mutex =1 (signal(mutex))
Reader1 in cs
Now, R2 comes. Wait(mutex) , so Mutex now is 0.
Read_count=2.
If condition is False. So now Mutex =1
Reader 2 is in cs

Solution Case 4 : R-R

read_count= 0 Semaphore rw_mutex =1 Semaphore mutex =1

Reader

```
do {
wait(mutex);
read_count++; // one more reader
if (read_count == 1)
   wait(rw_mutex); // First reader
 signal(mutex);
...
/* reading is performed */
...
wait(mutex);

read_count--; // one less reader
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

Writer

```
do {
wait(rw_mutex);
...
/* writing is performed */
...
signal(rw_mutex);
} while (true);
```

So now Mutex =1
Read_count =2
Reader 2 want to exit the cs. It will execute the exit code.
mutex = ~~1~~ 0
Read_count =~~2~~ 1
If condition is false
Signal(mutex) will execute and mutex will be again

Reader- Writer

Specifically, two rules must be enforced:

1. A reader is permitted to join other readers currently in the CS only when no writer is waiting. When the last reader exits the CS, the writer is allowed to enter.

Rule 1 guarantees that writers cannot starve

2. All readers that have arrived while a writer is in the CS must be allowed to enter before the next writer.

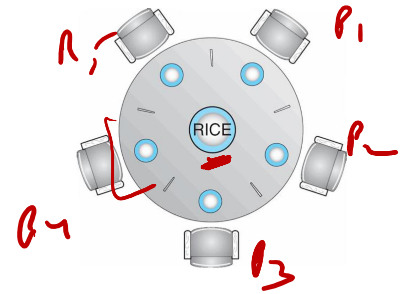
Rule 2 guarantees that readers cannot starve.

max concurrency of readers.

Dining Philosophers Problem

- Five philosophers spend their lives thinking and eating while sitting at a round table around a bowl of rice
- A chopstick is placed between each philosopher
- Philosophers **cannot** interact with their neighbours
- Each philosopher will think and occasionally eat
    - When ready to eat, a philosopher will try to pick up two chopsticks (one at a time) so he can eat some rice
    - A philosopher needs 2 chopsticks to eat
    - When done eating, a philosopher will put down each chopstick, one at a time
- How can the philosophers sit and eat together without anyone starving?

  - Think of each chopstick as a semaphore



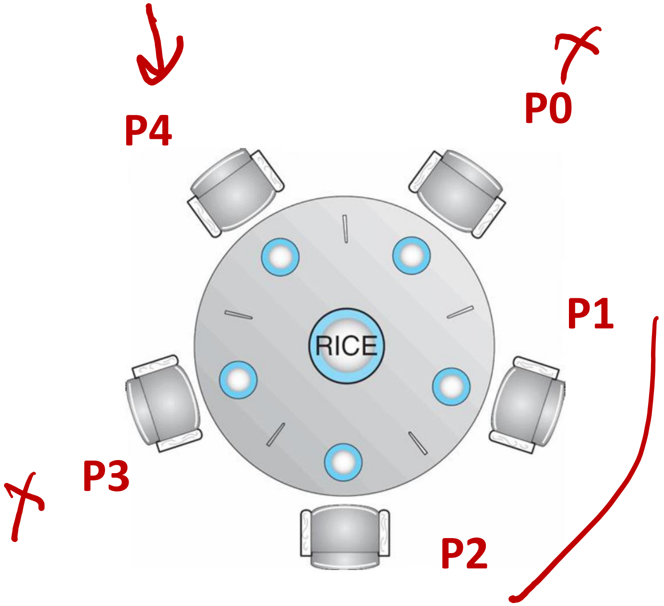
Clicker

When P4 is eating then ______ can eat concurrently.

(A) Only P1

(B) P0 or P1

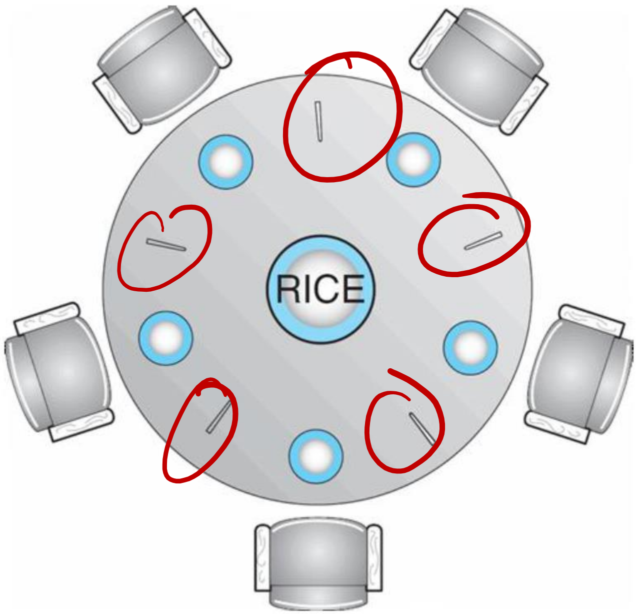
(C) P1 or P2



Dining Philosophers Problem

Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

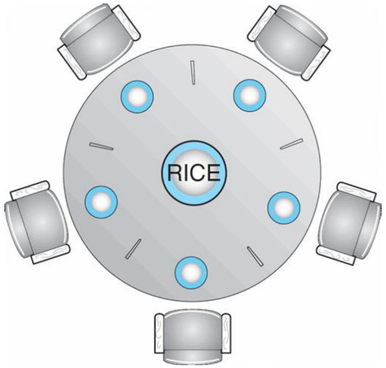


Dining Philosophers Problem

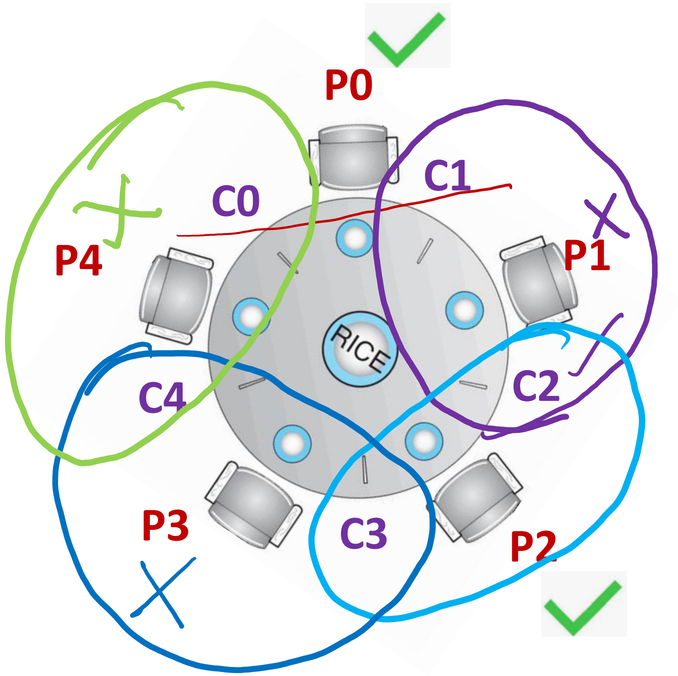
Possible Solution?

Instruct each philosopher to behave as follows:

- Think until the left chopstick is available; when it is pick it up
- Think until the right chopstick is available; when it is pick it up
- Eat some rice
- Put the left chopstick down
- Put the right chopstick down
- Go back to thinking



Dining Philosophers Problem



P0 comes in and grab C1 and C0 and start eating.

P1 comes in and grabs C2 and tries to grab C1 (which is with P~~1~~0). So, get blocked.

P2 comes in, C3 and C2 are both available so P2 grabs both and starts eating.

P3 comes in, grabs C4 and tries to grab C3 (which is with P2), so get blocked.

P4 comes in, C0 is with P1 so get blocked.

dealock

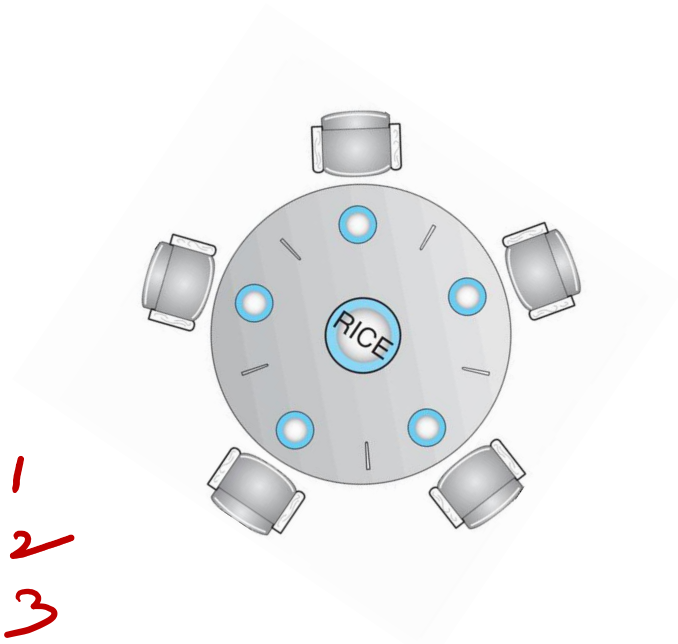
Dining Philosophers Problem

```
do {
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5] );
...
// eat
...
signal ( chopstick[i] );
signal ( chopstick[ (i + 1) % 5] );
...
// think
...
} while (TRUE);
```

left

Right

Each philosopher will grab left chopstick first



1 mod 5 1
2 mod 5 2
3 mod 5 3
4 mod 5 4
5 mod 5 0

Dining Philosophers Problem

Ques. What is the problem with this algorithm?

If each philosopher picks up his left chopstick at the same time, then they all sit waiting for the right chopstick forever (i.e. deadlock)

How do we solve this?

Dining Philosophers Problem

How do we solve the deadlock problem?

- **Option 1:** at most 2 philosophers can eat at the same time (using 4 chopsticks)
- **Option 2:** if we can prevent all five of the philosophers from picking up the first chopstick simultaneously, then we can guarantee that at least one can pick up the second chopstick

Dining Philosophers Problem

- Introduce another common semaphore. Call it flag.
- Initialize to 4
- Before picking up the first chopstick, the philosophers must wait on the flag.
- Once done with their chopsticks, they must signal the flag.

Dining Philosophers Problem

flg = 4

```
do {
     wait (flag) ;
     wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                // eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );
     signal (flag);
                // think
} while (TRUE);
```

Deadlock

- **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation : Indefinite blocking

- A process may never be removed from the semaphore queue in which it is suspended

- The semaphore/mutex might still be released, but another waiting process can get it first

Next topic : Deadlock

- Definition
- Techniques for preventing

Learning Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks

- To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

A set of processes each holding a resource and waiting to acquire a resource held by another process

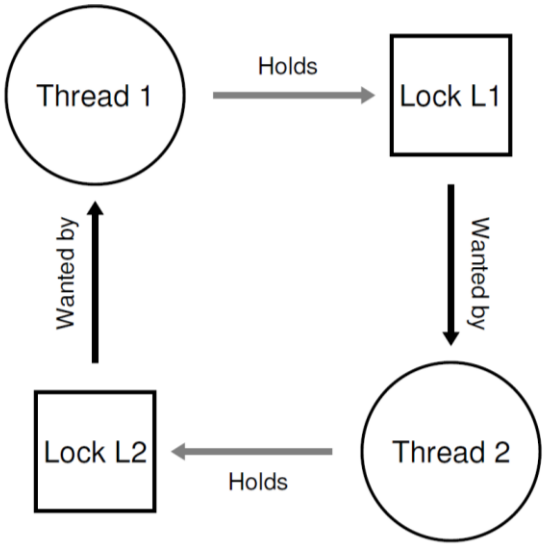
Example: two threads running **simultaneously** on two cores

Time

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

```
Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

- Thread 1 waits for Thread 2 to release L2
- Thread 2 waits for Thread 1 to release L1
- Two threads wait for each other



Necessary Conditions for Deadlock

Deadlock **may** arise if four conditions hold **simultaneously**

1. **Mutual exclusion:** threads require exclusive control of resources
2. **Hold and wait:** threads are already holding resources but also are waiting for additional resources being held by other threads
3. **No preemption:** resource released only voluntarily by the thread holding it
4. **Circular wait:** there exists a set {P0, P1, …, Pn} of processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for P2, …, Pn–1 is waiting for Pn, and Pn is waiting for P0

Circular wait implies hold & wait situation

* "Threads" and "processes" used interchangeably here

Deadlock detection

- Deadlock detection is when we try to find deadlocks after they have occurred and then try to take corrective action.
  - E.g., killing one of the threads that are in deadlock.
- Deadlock detection isn't typically done by the OS, but instead, it is done by the user or in a library.
- Deadlocks can be detected on-the-fly, by running **cycle detection algorithms** on the graph that defines the current use of resources.

System model

CPU memory space, I/O devices files

- resource types: $R_1, R_2, \ldots, R_n$
- each resource R has W_i instances

① Reusable
② Consumable
created by process
signals interrupts

- each process utilizes a resource as follows:

open() new() malloc()

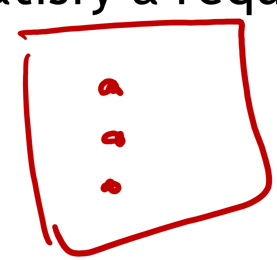
Request → use → release

close() free() delete()

Process will block if the resource is not free

Any instance of a resource type can be used to satisfy a request of that resource.

R_1



Deadlock

Given the graph, we can run any cycle detection algorithm.

Single instance

If a cycle is found,
we have a deadlock.

Multiple instances

there might not be
deadlock even if a cycle
is detected

N C