Semaphores

CS 447

Monday 3:30-5:00

Tuesday 2:00-3:30

What are the drawbacks of the algorithmic solutions?

- i.e. solutions with shared variables and atomic read and write?
  - Scalability: No of processes is to be known statically
  - Busy wait
  - Responsibility of implementation is with user
- Pointers to OS-supported solution?

Dijkstra's Semaphores

- Semaphore S is a variable
- 2 operations: P(S) and V(S)
- P – proberen/wait/down
- V – verogen/signal/up

P(S); CS V(S);	P(S); CS V(S);	P(S); CS V(S);

Original Implementation

S=K; //initial value

P()::

While (S= 0);

S=S-1;

V()::

S=S+1;

counting semaphore

Atomicity of primitives is to be guaranteed (somehow)!

How to realize a semaphore implementation that is free from busy-wait?

S=K

P	V

If (S>0) S = S - 1;
Else
insert calling process in *wait queue* associated with semaphore S,
 	
block the process
return
	If (wait queue associated with S is not empty)
	wake up one process from the queue
	
	S = S + 1;
	return

is it correct?

How to realize a semaphore implementation that is free from busy-wait?

S=K

P	V

If (S=0)
insert calling process in *wait queue* associated with semaphore S,
 	
block the process
 	
else S = S - 1;
	If (wait queue associated with S is not empty)
	wake up one process from the queue
	
	else S = S + 1;

Binary Semaphores

S=true

P	V

If (!S)
insert calling process in *wait queue*
associated with semaphore S,
 	
block the process
 	
else S = false;
	If (wait queue associated with S is not empty)
	wake up one process from the queue
	
	else S = true;

Exercise

- Implement a counting semaphore in terms of a binary semaphore:
- Pc(S) ::
  - Use Pb(S1)…Pb(Sk) and V(S1)…V(Sk)
- Similarly implement Vc(S)

- S is a shared integer – protect it through binary semaphores!

Semaphore based solutions to benchmark synchronization problems

- Producers and Consumers
- Dining Philosophers
- Readers and Writers

They have richer synchronization constraints than mere critical sections

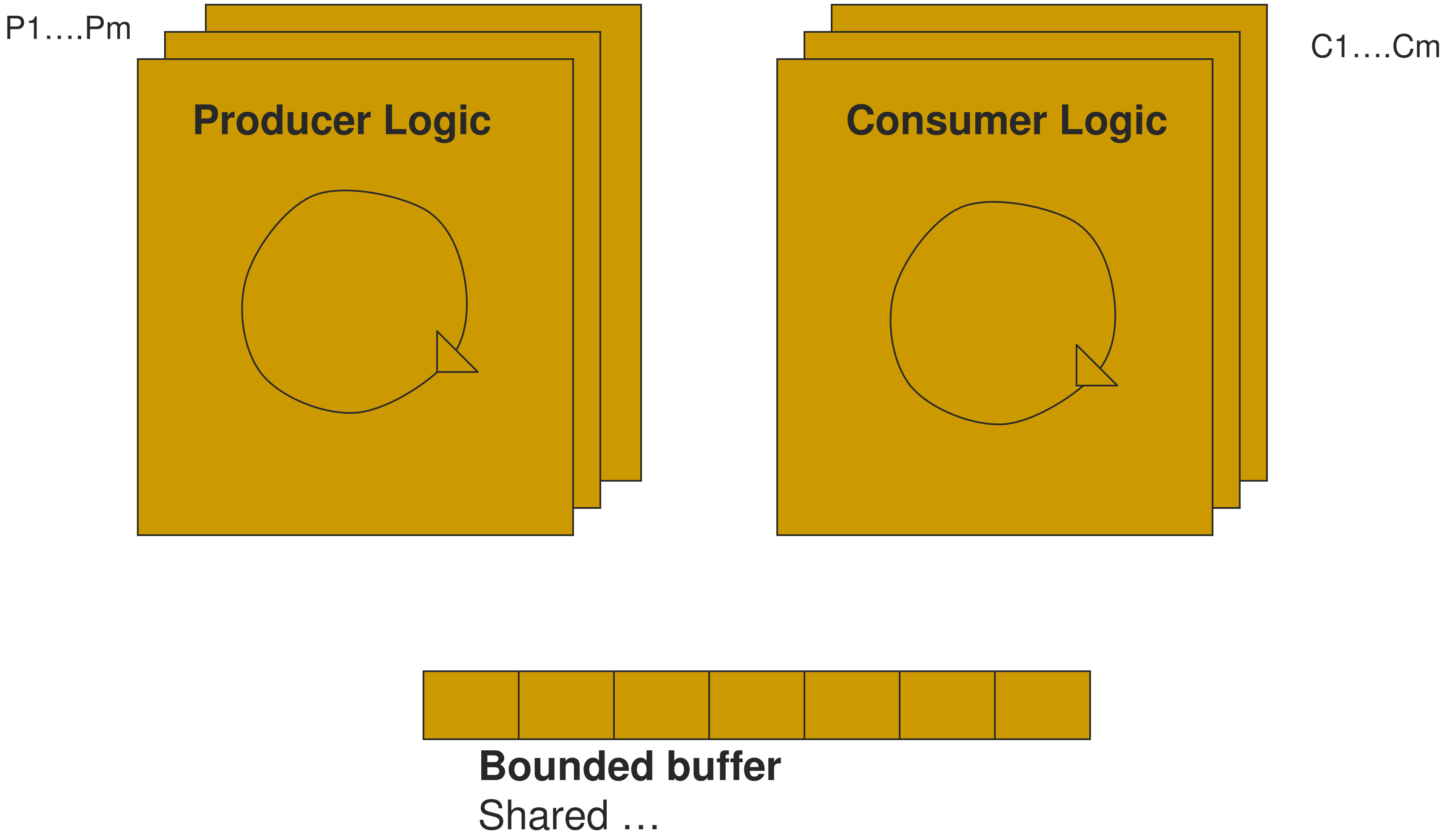
Producers and Consumers

- Common **bounded** buffer
- Producers keep producing items in this bounded buffer
- Consumers keep pulling them out of the buffer
- Buffer state must be consistent in presence of concurrency

Producers and Consumers: Additional constraints

- If buffer is full:
  - Let the producer wait
- If buffer is empty:
  - Let the consumer wait
- When the *triggering event* occurs, wait must be terminated

Try a semaphore based solution to producers and consumers



Producer : Attempt I

S1=size of buffer

S2=0; S3=0 or size

P(S1)

    If (buffer is not full) insert item; V(S3)

    else P(S2)

Producers and Consumers : Attempt II → solution

Shared Buffer

Sp=size of buffer

Sc=0; Smutex=1;

Producer

P(Sp)

P(Smutex) do the insertion V(Smutex)

V(Sc)

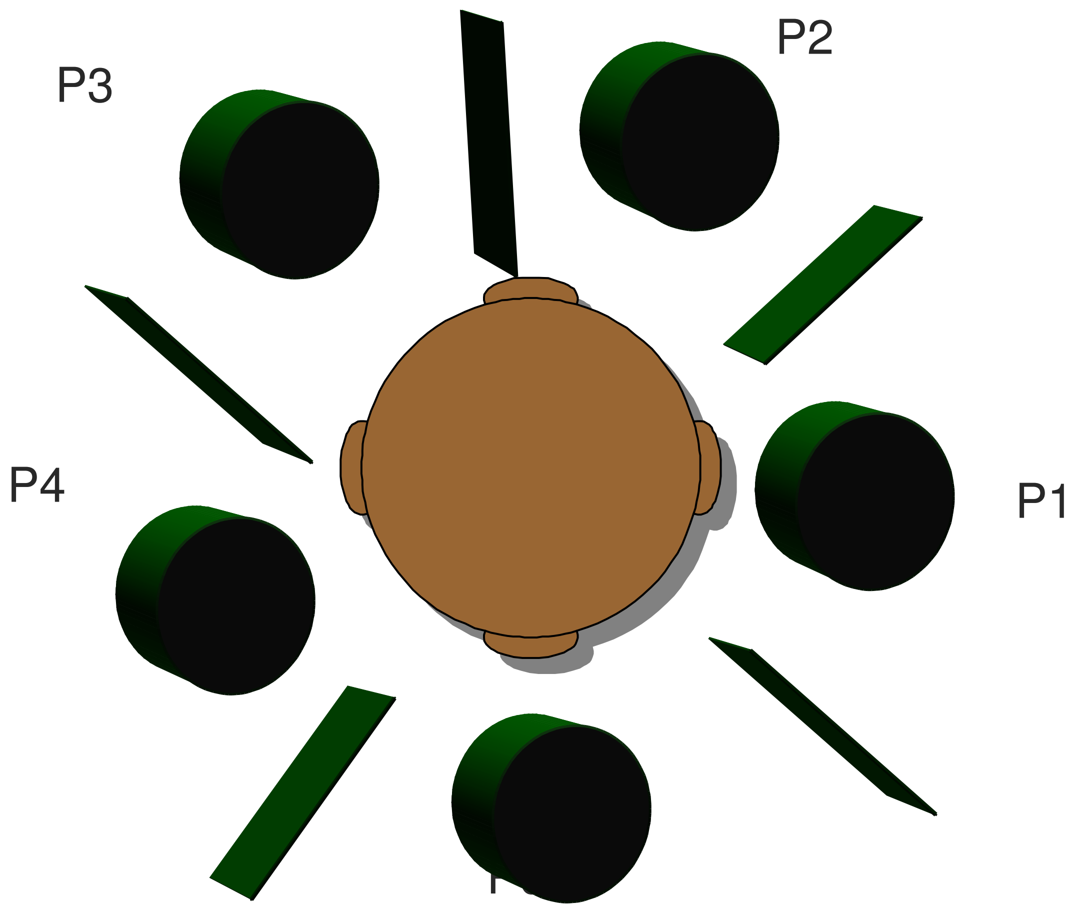
Consumer

P(Sc)

P(Smutex) fetch V(Smutex)

V(Sp)

Dining Philosophers



Attempt a solution

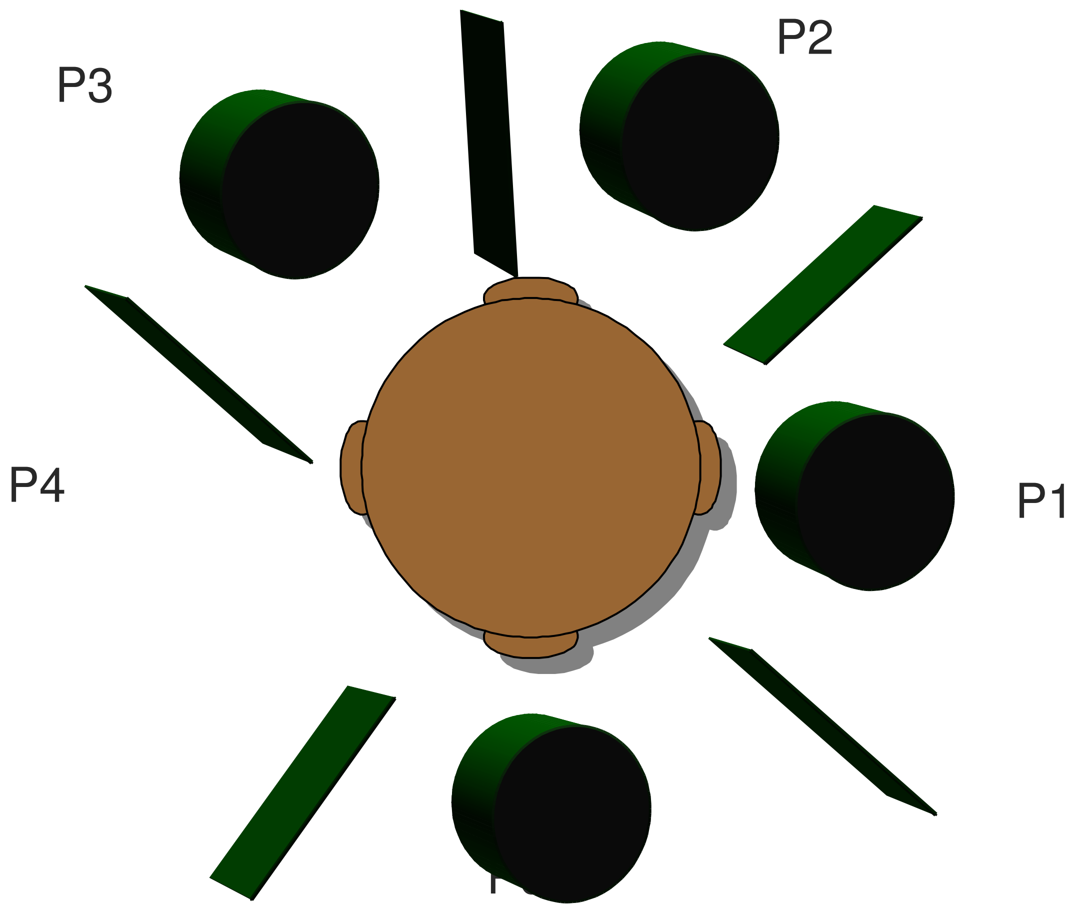
```
Shared forks[N], Semaphore S[N]
---------------------------------------------
Pi::
 while (true) {
     P(S[i])
     P(S[i+1 % N])
     eat
     V (S[i])
     V(S[i+1 %N])
     think
}
```

deadlock possible

Deadlock-free solution?

Let’s try one

Dining Philosophers without a deadlock

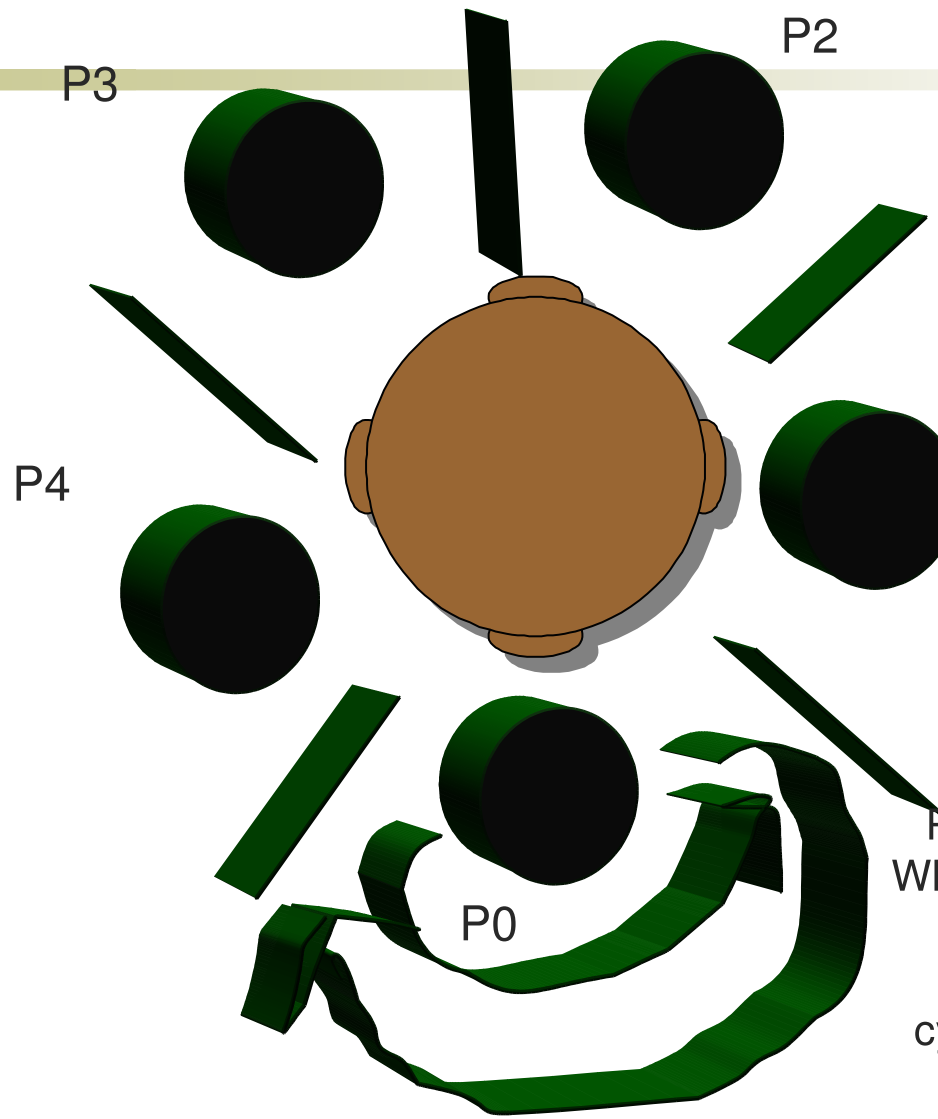


Implementation

Shared forks[N], Semaphore S[N] = {1,..1}, Table=N-1

```
Pi::
 while (true) {
      P(Table)
      P(S[i])
      P(S[i+1 % N])
      eat
      V (S[i])
      V(S[i+1 %N])
      V(Table)
      think
}
```

Dining Philosophers without a deadlock



P0 picks up right fork before left,
While others pick up left before right

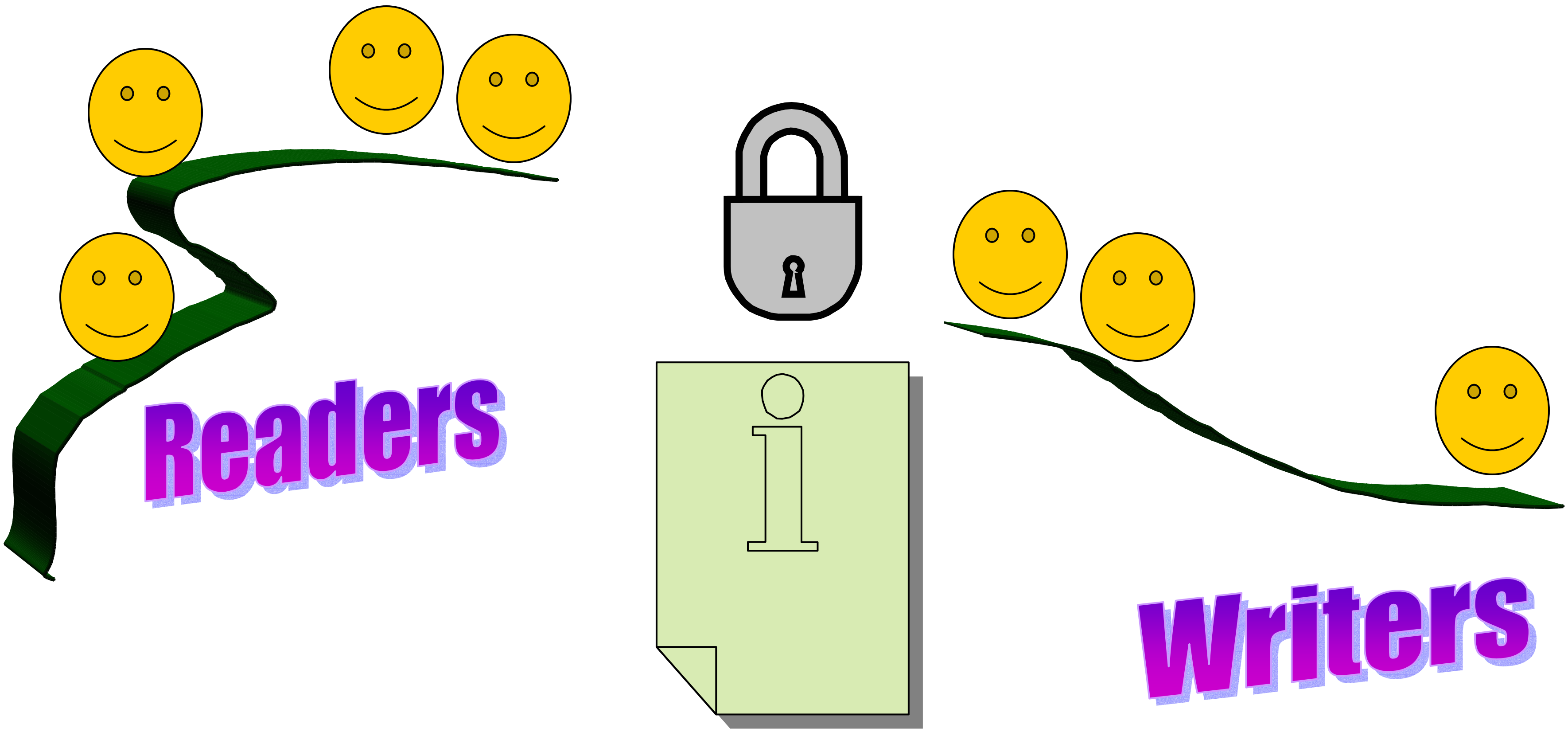
Prove that a hold and wait cycle cannot occur, and hence no deadlock

Implementation

Shared forks[N], Semaphore S[N]

```
Pi::
 while (true) {
     if (i==0) P(S[i+1%N]) else P(S[i])
     if (i=0) P(S[i]) else P(S[i+1 % N])
     eat
     V (S[i])
     V(S[i+1 %N])
     think
}
```

Readers and Writers Synchronization



Attempt I

Semaphore W=1 R=1

Reader: P(W) read V(W)

Writer: P(W) P(R) write V(W) V(R)

Semaphore R is not being us used, and
Each reader and each writer simply takes
An independent CS on the shared file.
We want more than this.

Semaphore S=1

Reader: P(S) read V(S)

Writer: P(S) write V(S)

Attempt II

Semaphore W=1 mutex-r=1
Shared int r=0;

Reader:

```
P(Mutex-r)
r=r+1
if (r==1) P(W)
V(Mutex-r)
read
P(Mutex-r)
r=r-1
if (r=0) V(W)
V(Mutex-r)
```

Writer:

```
P(Mutex-r)
P(W)
V(Mutex-r)
write
V(W)
```

correct,
deadlock possible!

A solution

- Readers

P (Mutex)
 r=r+1;
 if (r=1) P(Writer);
V(Mutex)

Read

P(Mutex)
 r=r-1;
 if (r=0) V(Writer)
V(Mutex)

- Writers

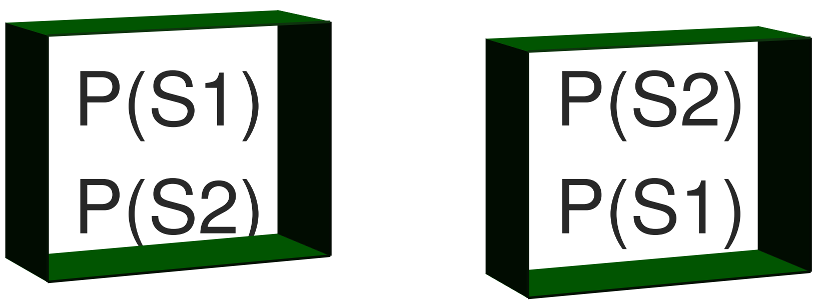
P (Writer)

Write

V (Writer)

Care to be taken with Semaphores (drawbacks)

- User programs must still use P and V correctly
- A forgotten P, or a misplaced V
- Possibility of deadlocks-



Better Higher level synchronization primitives?

- Critical Regions
- Conditional Critical Regions
- Monitors
  - These were supported in concurrent programming languages
  - Today's semaphore system calls allow monitor type synchronization as well