Concurrency

Mutual Exclusion and Synchronization

Outline

- Read Stallings, 5.1, 5.2, 5.3, 5.4, 5.7
- Topics:
  - Critical sections and mutual exclusion
  - Pure software solutions
  - Hardware-based solutions
  - Semaphores
  - Classic Concurrency Problems
    - Producer consumer
    - Dining philosophers
    - Readers/writers problem

Problems with Concurrent Execution

- Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources
- If there is no controlled access to shared data, some processes might obtain an inconsistent view of this data
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved, which typically is not deterministic

A Simple Example

- Process P1 and P2 are running this same procedure and have access to the same variable “a”
- Processes can be interrupted anywhere
- If P1 is first interrupted after user input and P2 executes entirely
- Then the character echoed by P1 will be the one read by P2 !!

```
static char a;

void echo()
{
    cin >> a;
    cout << a;
}
```

The Critical Section Problem

- When a process executes code that manipulates shared data (or resource), we say that the process is in a critical section (CS) for that shared data
- Sometimes referred to as a *critical region*
- The execution of critical sections must be mutually exclusive: At any time, only one process is allowed to execute in its critical section (even with multiple CPUs)
- Each process must request the permission to enter the critical section (CS)

Critical Sections

- The section of code implementing this request is called the entry section
- The critical section (CS) might be followed by an exit section
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors)

Mutual Exclusion and OS

- Operating systems are usually responsible for providing processes with mutually exclusive access to system resources
  - relatively easy on a uniprocessor. Why?

    Only one process in kernel at a time
  - for a multiprocessor, kernel will need need to implement critical sections in its own code (multiple processes in kernel)
- Critical sections sometimes need to be implemented by user programs, even on a uniprocessor. Example?

  Same file, share memory
- Obviously, SMP kernels implement critical sections everywhere. Even uniprocessor kernels need critical sections. Why? Interrupt Handlers

Framework for analysis of solutions

- Each process executes at nonzero speed but no assumption on the relative speed of *n* processes
- General structure of a process:

```
repeat
  entry section
   critical section
  exit section
   remainder section
forever
```

- Many CPUs may be present but memory hardware prevents simultaneous access to the same memory location
- No assumption about order of interleaved execution
- For solutions: we need to specify entry and exit sections

Requirements for a valid solution

- Mutual Exclusion
  - At any time, at most one process can be in the critical section (CS)
- Progress
  - If no process is executing in its CS while some processes wish to enter, any process that requests entry should be granted such without delay
- Processes remain in critical section for finite time
- A waiting process cannot be delayed indefinitely (no deadlock or starvation)

Types of Solutions

- "Pure" software solutions
  - algorithms where correctness does not rely on any other assumptions beyond framework described earlier
  - Simply implemented in code, without need for atomic instructions
- Hardware solutions
  - rely on special machine instructions
    - e.g., test-and-set, xchg, cmpxchg
- Operating system solutions
  - provide system calls and data structures to the user programs; mutual exclusion implemented by kernel

Pure Software Solutions

- We consider the case of two processes
  - We assess three different algorithms
- Approach generalizes to more processes
- Notation
  - We start with 2 processes: P0 and P1
  - When presenting process Pi, Pj always denotes the other process (i != j)

Algorithm 1

- The shared variable turn is initialized (to 0 or 1) before executing any Pi
- Pi's critical section is executed iff (turn == i)
- Pi is busy waiting if Pj is in CS: mutual exclusion is satisfied
- Problem?

Alternations prohibits progress

```
Process Pi:
repeat
  while(turn!=i){};
     CS
  turn = j;
     RS
forever
```

Algorithm 2

- Add a boolean variable for each process: flag[0] and flag[1]
- Pi signals that it is ready to enter it's CS by: flag[i] = true
- Mutual Exclusion is satisfied
- Problem?

```
Process Pi:
repeat
  flag[i]= true;
  while(flag[j]){};
     CS
  flag[i]= false;
     RS
forever
```


Algorithm 3 (Peterson's Algorithm)

- Initialization:
  flag[0] = flag[1] = false
  turn = 0 or 1
- Willingness to enter CS specified by setting
  flag[i] = true
- If both processes attempt to enter their CS simultaneously, only one `turn` value will prevail
- Exit section: specifies that Pi is unwilling to enter CS

```
Process Pi:
repeat
  flag[i]= true;
  turn = j;
  do {} while
  (flag[j] and (turn==j));
     CS
  flag[i]:=false;
     RS
forever
```

Algorithm 3: Proof of Correctness

- Mutual exclusion:
- Proof by contradiction:
- Assume both P0 and P1 are in their CS
  - then flag[0] = flag[1] = true
  - but the test for entry cannot have been true for both processes at the same time (because turn favors one);
  - therefore one process must have entered its CS first (without loss of generality, say P0)
  - but this means that P1 could not have found turn = 1 and therefore could not have entered its CS (i.e. contradiction)

Algorithm 3: Progress

Consider P0 blocked in the entry loop:

-- Case I: (Stuck)

– P1 is not interested in entering its CS

- then flag[1] = false
- hence the while loop is false for P0 and it can go

– Case II: (Deadlock)

– P1 is also blocked at the while loop

– - impossible, because turn = 0 or 1

– - hence the while loop is false for some process and it can go

Proof: Bounded Waiting

- Case III: (Starvation)
- P1 is executing its CS repeatedly
  - upon exiting its CS, P1 sets flag[1] = false
  - hence the while loop is false for P0 and it can go (sufficient?)

- However, P1 may attempt to re-enter its CS before P0 has a chance to run.
  - but to re-enter, P1 sets flag[1] to true and sets turn to 0
  - hence the while loop is true for P1 and it waits
  - the while loop is now false for P0 and it can go

- Note: Can extend to N processes (Bakery Algorithm)

What about process failures?

- If all three criteria (mutex, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS)
  - since failure in RS is just like having an infinitely long RS
- However, no valid solution can provide robustness against a process failing in its critical section (CS) – other recovery methods are needed

Drawbacks of software solutions

- Processes that are requesting to enter in their critical section are busy waiting (consuming processor time needlessly)
- Can a process busy-wait for its entire timeslice?
- If critical sections are long, it would be more efficient to block (i.e., suspend) those processes that are waiting...
- BTW, when is busy waiting considered acceptable?

  Kernel locks

Hardware Solutions: Interrupt Disabling

- Not allowed for user processes!
- What about the kernel? On a uniprocessor:
- On a multiprocessor:

  Atomic instructions

```
Process Pi:
repeat
  disable interrupts
   critical section
  enable interrupts
   remainder section
forever
```

Special Machine Instructions

- Processor designers have implemented machine instructions that perform two actions atomically (indivisible) on the same memory location (e.g., reading and writing)
- The execution of such an instruction is mutually exclusive (even with multiple CPUs)
- Such instructions can be used to provide mutual exclusion
- To satisfy all three requirements of the CS problem (incl. bounded waiting) requires additional mechanisms.

test-and-set semantics

- A C++ description of test-and-set:

```
bool testset(int& i)
{
  if (i==0) {
    i=1;
    return true;
  } else {
    return false;
  }
}
```

- An algorithm that uses testset for Mutual Exclusion:
- Shared variable b is initialized to 0
- Only the first Pi who sets b enters the CS

```
Process Pi:
repeat
  repeat{}
  until testset(b);
     CS
  b:=0;
     RS
forever
```

test-and-set properties

- Mutual exclusion is preserved: if Pi enter CS, the other Pj are busy waiting
- Problem: still uses busy waiting
- When Pi exit CS, the selection of the Pj who will enter CS is arbitrary: no bounded waiting. Hence starvation is possible
- Other instructions on various processors, such as xchg(a,b) swaps the content of a and b.

Using xchg for mutual exclusion

- Shared variable b is initialized to 0
- Each Pi has a local variable k
- The only Pi that can enter CS is the one who finds b=0
- This Pi excludes all the other Pj by setting b to 1

```
Process Pi:
repeat
  k:=1
  repeat xchg(k,b)
  until k=0;
     CS
  b:=0;
     RS
forever
```

But xchg suffers from the same drawback as test-and-set

OS-Based Solutions

- We saw that we can implement mutual exclusion:
  - Purely in code, with no help from hardware or the OS
  - By disabling interrupts (but only for uniprocessor kernel)
  - With atomic instructions (either user or kernel)
  - Problem with these approaches?
- Alternative approach for supporting user processes: Implement mutual exclusion (and process synchronization) in the OS.
- Examples:
  - mutexes, semaphores, condition variables, monitors

Advantage of OS support?

Semaphores

- Synchronization tool (usually provided by the OS) that does not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed via atomic and mutually exclusive operations:
  - wait(S) -- sometimes P(S)
  - signal(S) -- sometimes V(S)
- To avoid busy waiting: when a process has to wait, it will be put in a queue of blocked processes waiting for the same event

Semaphores

- Hence, a semaphore can be implemented as a structure with two fields:
  – count: integer
  – queue: list of processes

- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue
- The signal operation removes (according to a fair policy like FIFO) one process from the queue and puts it in the list of ready processes

Semaphore's operations (atomic)

```
wait(S){
  S.count--;
  if (S.count < 0) {
    block this process in S.queue
  }

signal(S){
  S.count++;
  if (S.count <= 0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

S.count must be initialized to a nonnegative value (depending on application)

Semaphores: Observations

- When S.count >=0: the number of processes that can execute wait(S) without being blocked = S.count
- When S.count<0: the number of processes waiting on S is = |S.count|
- Atomicity and mutual exclusion: no two process can be in wait(S) and signal(S) (on the same semaphore, S) at the same time (even with multiple CPUs)
- Hence the blocks of code defining wait(S) and signal(S) are, in fact, critical sections

Using semaphores for critical sections!

- For n processes
- Initialize S.count to 1
- Then only 1 process is allowed into CS (mutual exclusion)
- To allow k processes into CS, we initialize S.count to k

```
Process Pi:
repeat
  wait(S);
   CS
  signal(S);
   RS
forever
```

Using semaphores for synchronization

- We have two processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Then define a semaphore “synch”
- Initialize synch to 0
- Proper synchronization is achieved by having in P1:
  - S1;
  - signal(synch);
- And having in P2:
  - wait(synch);
  - S2;

Semaphores in Linux

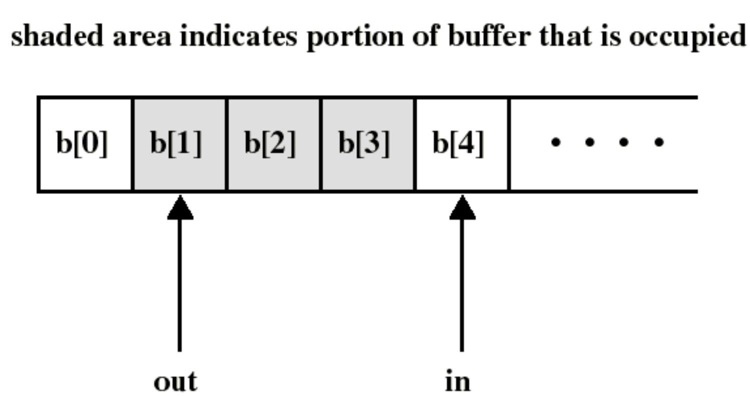
- Included in POSIX standard
- Can be used by processes and threads
- Implementation:
  - integer whose value is never allowed to fall below zero
  - sem_post(3): increment the semaphore value by one
  - sem_wait(3): decrement the semaphore value by one. If the value of a semaphore is currently zero, then a sem_wait operation will block until the value becomes greater than zero.

The producer/consumer problem

- A producer process produces information that is consumed by a consumer process
  - Ex1: a print program produces characters that are consumed by a printer
  - Ex2: an assembler produces object modules that are consumed by a loader
- We need a buffer to hold items that are produced and eventually consumed
- A common paradigm for cooperating processes

P/C: unbounded buffer

- We assume first an unbounded buffer consisting of a linear array of elements
- `in` points to the next item to be produced
- `out` points to the next item to be consumed



P/C: unbounded buffer

- We need a semaphore S to perform mutual exclusion on the buffer: only 1 process at a time can access the buffer
- We need another semaphore N to synchronize producer and consumer on the number N of items in the buffer (N = in - out)
  - Note: an item can be consumed only after it has been created

P/C: unbounded buffer

- The producer is free to add an item into the buffer at any time (since buffer is unbounded)
  - it performs wait(S) before appending and signal(S) afterwards to prevent customer access
  - It also performs signal(N) after each append to increment N
- The consumer must first wait(N) to see if there is an item to consume and then use wait(S)/signal(S) to access the buffer

Solution of P/C: unbounded buffer

```
Initialization:
  S.count:=1;
  N.count:=0;
  in:=out:=0;
```

```
append(v):
b[in]:=v;
in++;

take():
w:=b[out];
out++;
return w;
```

```
Producer:
repeat
  produce v;
  wait(S);
  append(v);
  signal(S);
  signal(N);
forever
```

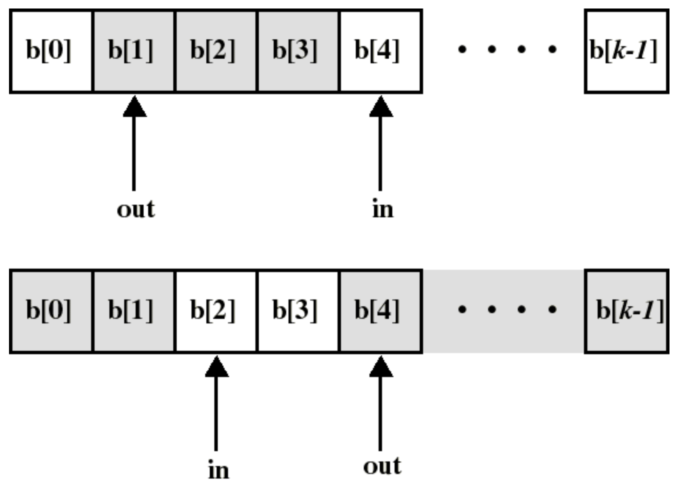
```
Consumer:
repeat
  wait(N);
  wait(S);
  w:=take();
  signal(S);
  consume(w);
forever
```

■ critical sections

P/C: unbounded buffer

- Remarks:
  - Putting signal(N) inside the CS of the producer (instead of outside) has no effect since the consumer must always wait for both semaphores before proceeding
  - The consumer must perform wait(N) before wait(S), otherwise deadlock occurs if consumer enter CS while the buffer is empty
- BTW, using semaphores and debugging solutions is not always trivial…

P/C: finite circular buffer of size k



- can consume only when number N of (consumable) items is at least 1
- can produce only when number E of empty spaces is at least 1

P/C: finite circular buffer of size k

- As before:
  - we need a semaphore S to have mutual exclusion on buffer access
  - we need a semaphore N to synchronize producer and consumer on the number of consumable items
- In addition:
  - we need a semaphore E to synchronize producer and consumer on the number of empty spaces, so producer knows if it can append

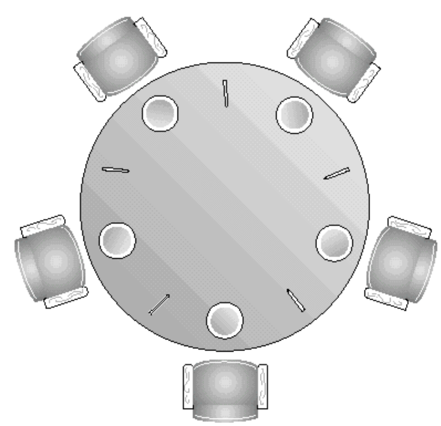
Solution

```
      Initialization: S.count:=1; in:=0;
                      N.count:=0; out:=0;
                      E.count:=k;

append(v):
b[in]:=v;                  Producer:       Consumer:
in:=(in+1)mod k;           repeat          repeat
                             produce v;      wait(N);
                             wait(E);        wait(S);
take():                      wait(S);        w:=take();
w:=b[out];                   append(v);      signal(S);
out:=(out+1)mod k;           signal(S);      signal(E);
return w;                    signal(N);      consume(w);
                           forever         forever
```

- critical sections

The Dining Philosophers Problem

- Classic synchronization problem…
- Five philosophers who only eat and think
- Each need to use two forks for eating
- We have only 5 forks
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



The Dining Philosophers Problem

- Each philosopher is a process
- One semaphore per fork:
  - fork: array[0..4] of semaphores
  - Initialization: fork[i].count:=1 for i:=0..4
- Solution is shown a the right.
- Problem?

```
Process Pi:
repeat
 think;
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
forever
```


The Dining Philosophers Problem

- One solution: admit only four philosophers at a time.
- Then one philosopher can always eat when the other three are holding one fork
- Hence, we can use another semaphore T that would limit at four the number of philosophers "sitting at the table"
- Initialize: T.count:=4

```
Process Pi:
repeat
 think;
 wait(T);
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
 signal(T);
forever
```

Deadlock, livelock, starvation

- Deadlock: permanent blocking of a set of processes competing for resources and/or communicating
- Three conditions for deadlock:
  - **mutual exclusion** in access to resources
  - a process may **hold** resources **and wait** for others
  - no preemption – cannot forcibly remove resource from process holding it.
- Livelock – set of processes continuously change states (regarding resource acquisition) but do not make progress.
- Runnable processes never obtains resource

Examples

Readers/Writers Problem

- Classic problem in computer science
- A data area is shared among many processes
    - some processes only read the data area, (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
    1. any number of readers may simultaneously read the file
    2. only one writer at a time may write to the file
    3. if a writer is writing to the file, no reader may read it

Readers have priority

- Semaphore mutex
    - Simply protects readcount var
    - Could just be a mutex, if such primitives are available
- Semaphore wsem
    - For mutually exclusive access to critical section
- Problem with this solution?

```
semaphore wsem=1, mutex=1; readcount=0;

writer()
{
   wait(wsem);
   // Do the writing (Critical Section)
   signal(wsem);
}

reader()
{
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wsem);
   signal(mutex);
   // Do the reading (Critical Section)
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wsem);
   signal(mutex);
}
```

Writers have priority

- Somewhat more complicated
- Need to keep track of writers who might be waiting
- Need to enable them to “jump ahead” of waiting readers
- Some additional variables:

```
// global variables.  this solution uses only semaphores, not mutexes
int readcount = 0;  // how many readers are currently in
int writecount = 0;  // how many writers are currently in (or want in)
semaphore readcount_mutex = 1;  // mutually exclusive updating of readcount
semaphore writecount_mutex = 1; // mutually exclusive updating of writecount
semaphore reader_gate = 1;  // why needed?
semaphore write_sem = 1;, // mutual exclusion on critical section
semaphore read_sem = 1;   // readers have to wait if any writer desires access
```

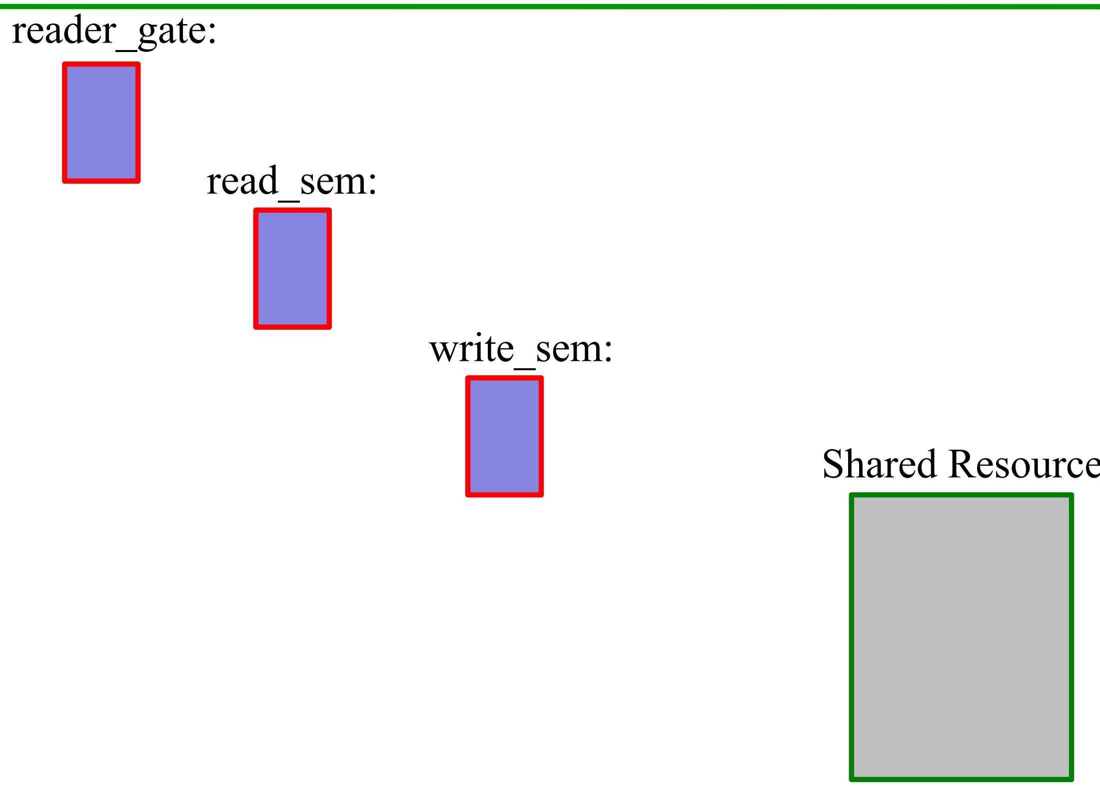
Writers have priority (writer code)

```
writer()
{
  // indicate writer is waiting
  wait(writecount_mutex);
     writecount++;
     if (writecount == 1)
        wait(read_sem);
  signal(writecount_mutex);
  wait(write_sem);

  // writing is performed here

  signal(write_sem);

  wait(writecount_mutex);
     writecount := writecount - 1;
     if (writecount == 0)  // no other writers are waiting, so let in any waiting readers
        signal(read_sem);
  signal(writecount_mutex);
}
```

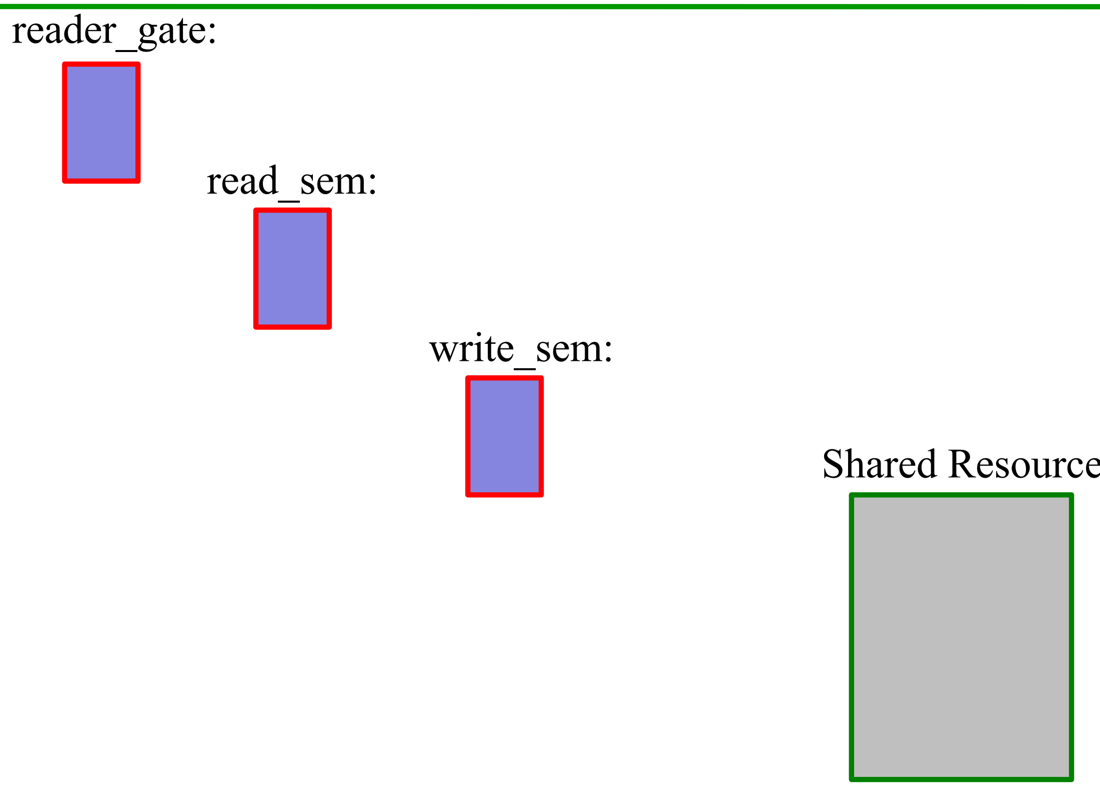
Writers have priority (reader code)

```
reader()
{
    wait(reader_gate);  // first gate for readers
       wait(read_sem);   // is it ok for reader to wait
          wait(readcount_mutex); // update readcount
             readcount := readcount + 1;
             if (readcount == 1)
                 wait(write_sem);  //
          signal(readcount_mutex);
       signal(read_sem);
    signal(reader_gate);

  //reading is performed here

  // reading is done; decrement readcount; if all readers done, release write_sem
    wait(readcount_mutex);
       readcount = readcount - 1;
         if (readcount == 0)
             signal(write_sem); // ok to let someone in (either reader or writer), writers priority
     signal(readcount_mutex);
}
```

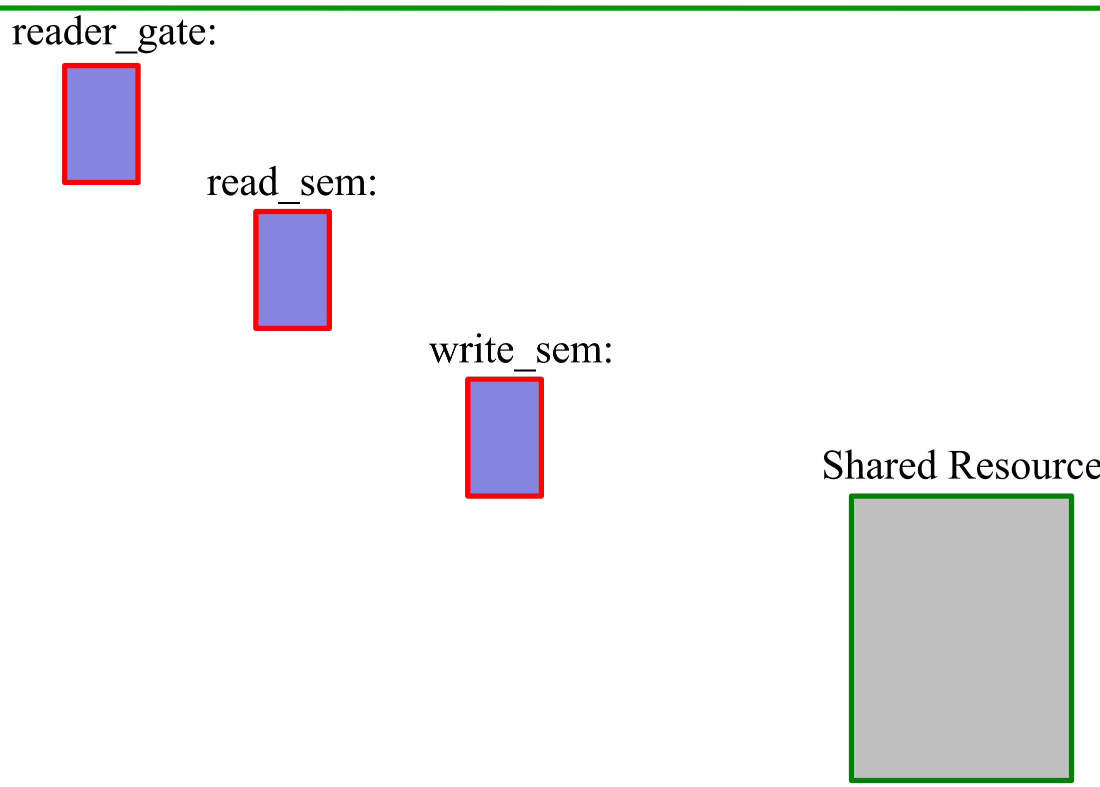

Example



Example



Example



Summary

- In this chapter we have addressed the issue of concurrent execution among processes/threads
- Key issues
  - Mutual exclusion (progress, no starvation, no deadlock)
  - Synchronization
- Methods
  - Mutexes, semaphores (others: monitors, condition vars)
- Classic problems that arise in OS and general CS:
  - Resource dependencies (dining philosophers)
  - Producer/consumer problem
  - Readers/writers problem