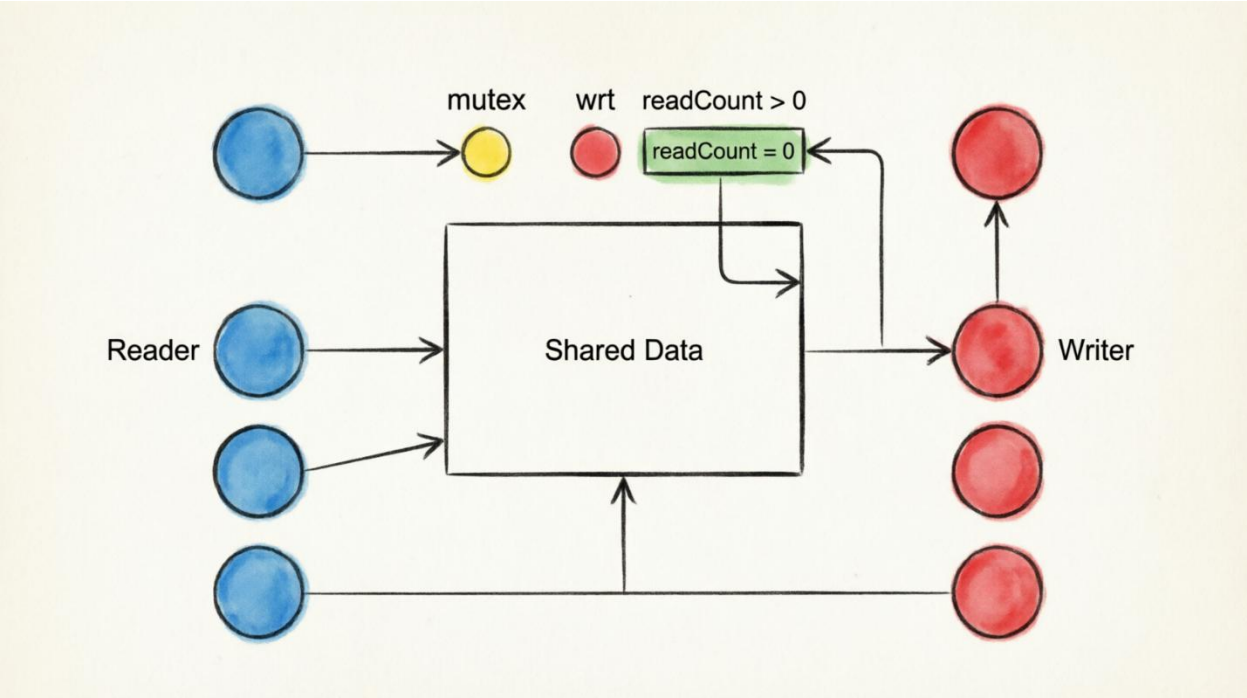
Reader-Writer Problem

1 Reader-Priority (Readers Preferred)

Behavior: New readers can enter as long as at least one reader is active. Writers may starve if readers arrive continuously.



1.1 Semaphores

```
semaphore mutex = 1;   // protects readCount
semaphore wrt   = 1;   // controls write access
int readCount = 0;

Reader() {
   wait(mutex);
   readCount++;
   if (readCount == 1) wait(wrt);  // first reader locks writers out
   signal(mutex);

   // --- READ SHARED DATA ---

   wait(mutex);
   readCount--;
   if (readCount == 0) signal(wrt); // last reader unlocks writers
   signal(mutex);
```

```
}

Writer() {
   wait(wrt);
   // --- WRITE SHARED DATA ---
   signal(wrt);
}
```

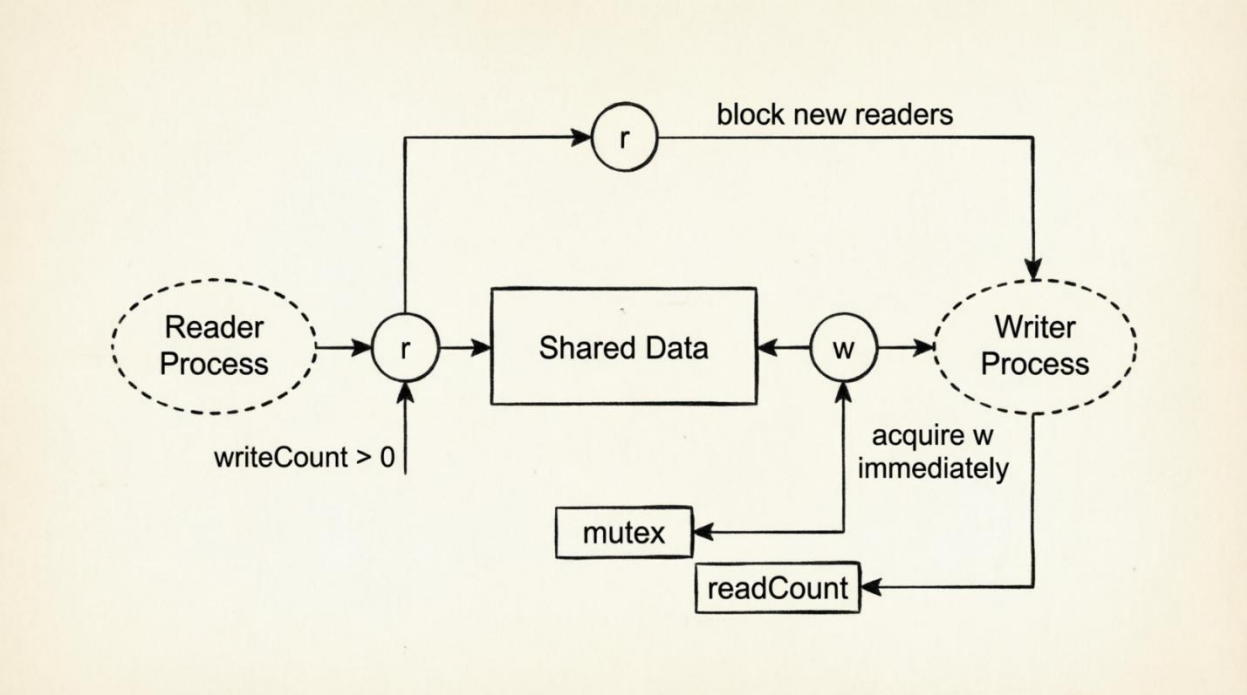
1.2 Monitors

```
monitor RW_ReaderPriority {
   int readers = 0, writers = 0;
   condition okToRead, okToWrite;

   void startRead() {
      while (writers > 0) wait(okToRead);
      readers++;
      signal(okToRead); // Allow concurrent readers
   }
   void endRead() {
      readers--;
      if (readers == 0) signal(okToWrite);
   }
   void startWrite() {
      while (readers > 0 || writers > 0) wait(okToWrite);
      writers++;
   }
   void endWrite() {
      writers--;
      signal(okToRead); // Always prefer waking readers
      signal(okToWrite); // Wake writers only if no readers waiting
   }
}
```

2 Writer-Priority (Writers Preferred)

Behavior: When a writer arrives, new readers are blocked until the writer finishes. Prevents writer starvation at the cost of reader latency.



2.1 Semaphores

```
semaphore r     = 1;   // blocks new readers when writers are waiting
semaphore w     = 1;   // exclusive write lock
semaphore mutex = 1;   // protects counters
int readCount  = 0;
int writeCount = 0;

Reader() {
   wait(r);            // may block if writeCount > 0
   wait(mutex);
   readCount++;
   if (readCount == 1) wait(w);
   signal(mutex);
   signal(r);

   // --- READ SHARED DATA ---

   wait(mutex);
   readCount--;
   if (readCount == 0) signal(w);
   signal(mutex);
}

Writer() {
   wait(mutex);
```

```
    writeCount++;
    if (writeCount == 1) wait(r); // block incoming readers
    signal(mutex);

    wait(w);
    // --- WRITE SHARED DATA ---
    signal(w);

    wait(mutex);
    writeCount--;
    if (writeCount == 0) signal(r); // unblock readers
    signal(mutex);
}
```

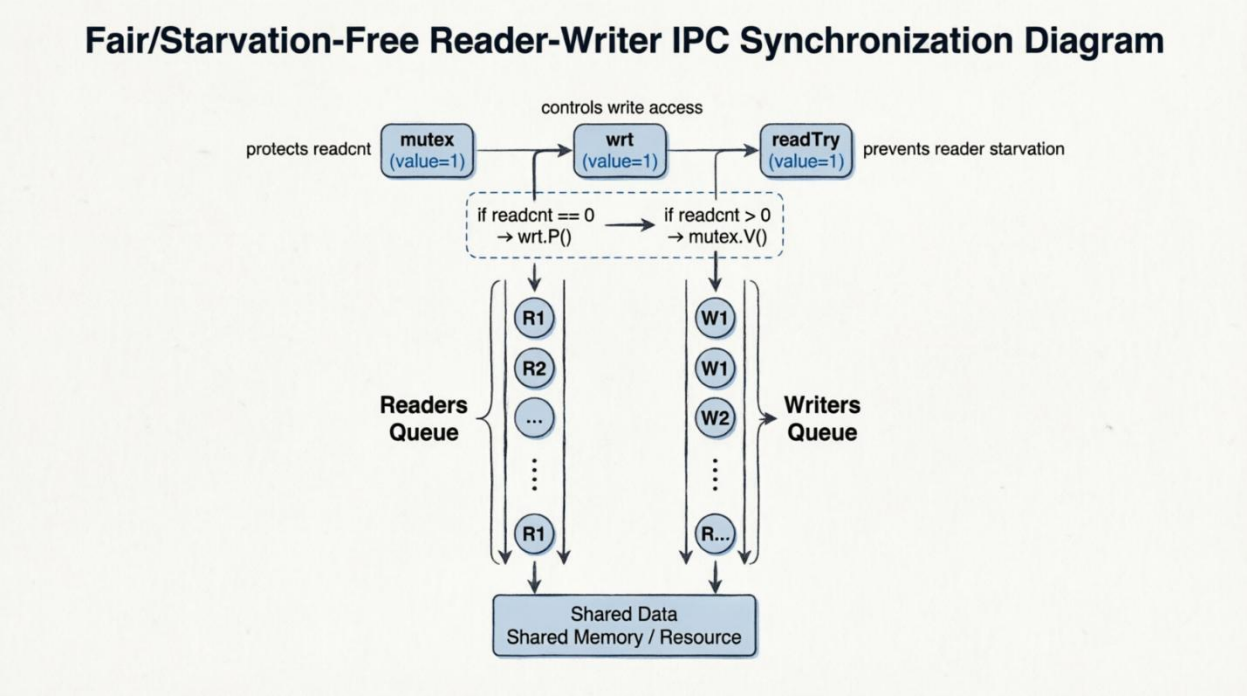
2.2 Monitors

```
monitor RW_WriterPriority {
    int readers = 0, writers = 0, waitW = 0;
    condition okToRead, okToWrite;

    void startRead() {
        while (writers > 0 || waitW > 0) wait(okToRead);
        readers++;
        signal(okToRead);
    }
    void endRead() {
        readers--;
        if (readers == 0) signal(okToWrite);
    }
    void startWrite() {
        waitW++;
        while (readers > 0 || writers > 0) wait(okToWrite);
        waitW--;
        writers++;
    }
    void endWrite() {
        writers--;
        if (waitW > 0) signal(okToWrite);
        else signal(okToRead);
    }
}
```

3 Fair / Starvation-Free (Alternating or FIFO

Behavior: Readers and writers are served in arrival order. Neither class can indefinitely monopolize the resource.



3.1 Semaphores

```
semaphore mutex   = 1;   // protects readCount
semaphore wrt     = 1;   // write lock
semaphore readTry = 1;   // ensures writers get a turn
int readCount = 0;

Reader() {
  wait(readTry);       // writers also wait on this → FIFO fairness
  wait(mutex);
  readCount++;
  if (readCount == 1) wait(wrt);
  signal(mutex);
  signal(readTry);

  // --- READ SHARED DATA ---

  wait(mutex);
  readCount--;
  if (readCount == 0) signal(wrt);
  signal(mutex);
```

```
}

Writer() {
   wait(readTry);       // queues behind arriving readers
   wait(wrt);
   // --- WRITE SHARED DATA ---
   signal(wrt);
   signal(readTry);     // passes turn to next waiting process
}
```

3.1 Monitors

```
monitor RW_Fair {
   int readers = 0, writers = 0;
   int waitR = 0, waitW = 0;
   int turn = 0; // 0 = readers' turn, 1 = writers' turn
   condition okToRead, okToWrite;

   void startRead() {
      waitR++;
      while (writers > 0 || (waitW > 0 && turn == 1)) wait(okToRead);
      waitR--; readers++; turn = 0;
      signal(okToRead);
   }
   void endRead() {
      readers--;
      if (readers == 0 && waitW > 0) { turn = 1; signal(okToWrite); }
   }
   void startWrite() {
      waitW++;
      while (readers > 0 || writers > 0 || (waitR > 0 && turn == 0)) wait(okToWrite);
      waitW--; writers++; turn = 1;
   }
   void endWrite() {
      writers--;
      if (waitR > 0) { turn = 0; signal(okToRead); }
      else signal(okToWrite);
   }
}
```