Concurrency

concurrency	the simultaneous occurrence of events or circumstances; agreement or union in action

Value of concurrency – speed and economics

But few widely-accepted concurrent programming languages (Java is an exception)

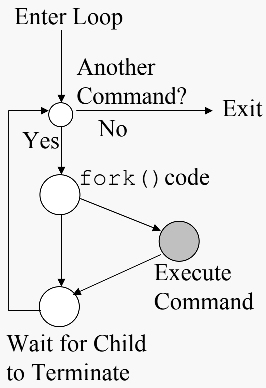
Few concurrent programming paradigms

- each problem requires careful consideration
- there is no common model

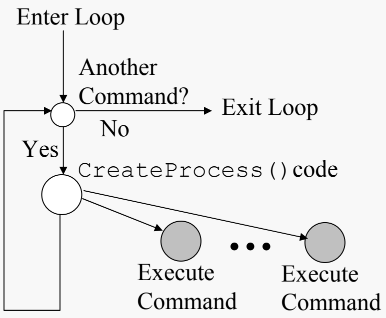
OS tools to support concurrency tend to be:

- low level (not that there's anything wrong with that)
- non-portable (pthreads and Java may be exceptions)

Command Execution

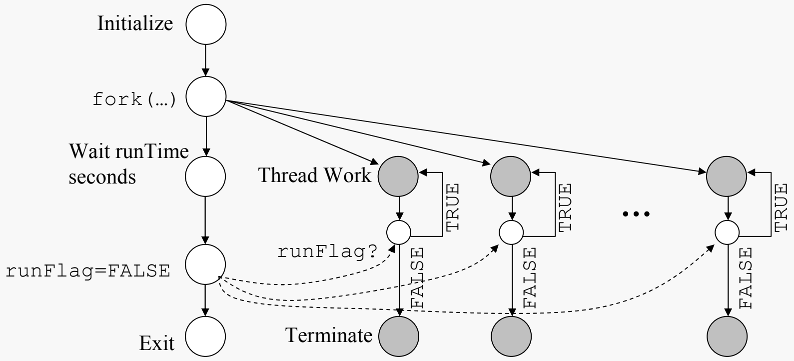


(a) UNIX Shell



(b) Windows Command Launch

Synchronizing on a Shared Variable



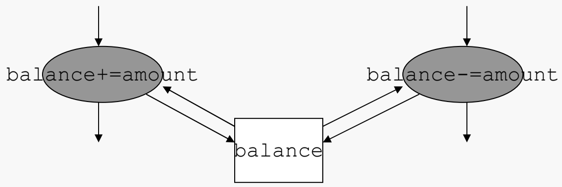
Critical Sections

critical section a segment of code that cannot be (safely) executed while some other process is in a corresponding segment of code

```
shared double balance;
```

Code for p_1 Code for p_2

```
 . . .                                  . . .
balance = balance + amount;   balance = balance - amount;
 . . .                                  . . .
```



Critical Sections

Execution of p_1 **Execution of p_2**

```
        …
        load  R1, balance
        load  R2, amount
  ---------------------------------------->
  Timer interrupt
                                             …
                                             load  R1, balance
                                             load  R2, amount
                                             sub   R1, R2
                                             store R1, balance
                                             …
  Timer interrupt
  ---------------------------------------->
        add   R1, R2
        store R1, balance
        …
```

Critical Sections

mutual exclusion only one process can be in the critical section at a time

There is a *race* to execute critical sections

The sections may be defined by different code in different processes

- ∴ cannot easily detect with static analysis

Without mutual exclusion, results of multiple execution are not *determinate*

Need an OS mechanism so programmer can resolve races

Disabling Interrupts

```
shared double balance;
```

Code for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

Interrupts could be disabled for arbitrarily long periods

Really only want to prevent p_1 and p_2 from interfering with one another; this blocks all p_i

Try using a shared "lock" variable

Using a Lock Variable

```
shared bool   lock = FALSE;
shared double balance;
```

Code for p_1

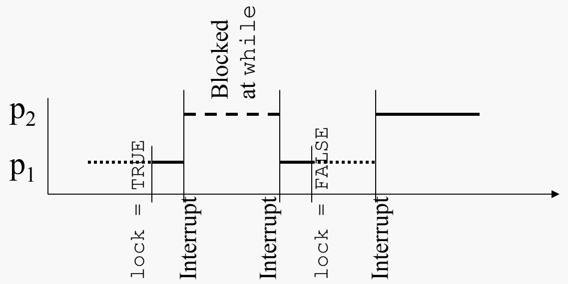
```
/* Acquire the lock */
  while (lock);
  lock = TRUE;
/* Execute critical sect */
  balance = balance + amount;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
  while (lock);
  lock = TRUE;
/* Execute critical sect */
  balance = balance - amount;
/* Release lock */
  lock = FALSE;
```

Will this work?

Busy Wait Condition



At best, the solution requires busy-waiting on the part of the "blocked" process.

Busy-waiting wastes CPU cycles and is inelegant.

However…

Unsafe "Solution"

Consider what could happen if an context switch occurred just after P1 exits its busy-wait loop:

Code for p_1

```
/* Acquire the lock */
  while (lock);
  lock = TRUE;
/* Execute critical sect */
  balance = balance + amount;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
  while (lock);
  lock = TRUE;
/* Execute critical sect */
  balance = balance - amount;
/* Release lock */
  lock = FALSE;
```

Looks like we've replaced one race condition with another.

Is it possible to solve the problem?

Canonical Problem

```
<shared global declarations>
<initial processing>
fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_0() {
  while (true) {
    <compute section>;
    <critical section>;
  }
}
```

```
proc_1() {
  while (true) {
    <compute section>;
    <critical section>;
  }
}
```

We must find a way to enforce mutual exclusion on the respective critical sections.

Solution Constraints and Assumptions

Only processes competing for a CS are involved in resolving who enters the CS

Once a process attempts to enter its CS, it cannot be postponed indefinitely

After requesting entry, only a bounded number of other processes may enter before the requesting process

Memory read/writes are indivisible (simultaneous attempts result in some arbitrary order of access)

There is no priority among the processes

Relative speeds of the processes/processors is unknown

Processes are cyclic and sequential

Dijkstra Semaphore

Invented in the 1960s

Conceptual OS mechanism, with no specific implementation defined

Basis of all contemporary OS synchronization mechanisms

Classic paper describes several software attempts to solve the problem

Found a software solution, but then proposed a simpler hardware-based solution

A *semaphore*, s, is a nonnegative integer variable that can only be changed or tested by these two indivisible (atomic) functions:

```
V(s): [s = s + 1]
P(s): [while (s == 0) {wait}; s = s - 1]
```

Solving the Canonical Problem

```
semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_0() {
  while (true) {
    <compute section>;
    P( mutex );
    <critical section>;
    V( mutex );
  }
}
```

```
proc_1() {
  while (true) {
    <compute section>;
    P( mutex );
    <critical section>;
    V( mutex );
  }
}
```

Remember that `P()` and `V()` are, by definition, indivisible operations.

Semaphore Solution to Shared Balance

If semaphores are available, there is a simple solution to the shared balance problem:

Code for p_1

```
/* Acquire the semaphore */
   P( mutex );
➡
/* Execute critical sect */
   balance = balance + amount;
/* Release semaphore */
   V( mutex );
```

What if there's a context switch at the indicated point now?

No problem at all.

And there cannot be a context switch within the body of `P()` or `V()`.

Code for p_2

```
/* Acquire the semaphore */
   P( mutex );
/* Execute critical sect */
   balance = balance - amount;
/* Release semaphore */
   V( mutex );
```

Sharing Two Variables

```
int x, y;
fork(proc_A, 0);
fork(proc_B, 0);
```

```
proc_A() {
   while (true) {
      <compute section A1>;
      update(x);
      <compute section A2>;
      retrieve(y);
   }
}
```

```
proc_B() {
   while (true) {
      retrieve(x);
      <compute section B1>;
      update(y);
      <compute section B2>;
   }
}
```

In effect, the processes are using each of the two shared variables as a one-way communication channel.

But values may be lost, and the same value may be retrieved multiple times.

Semaphore Solution

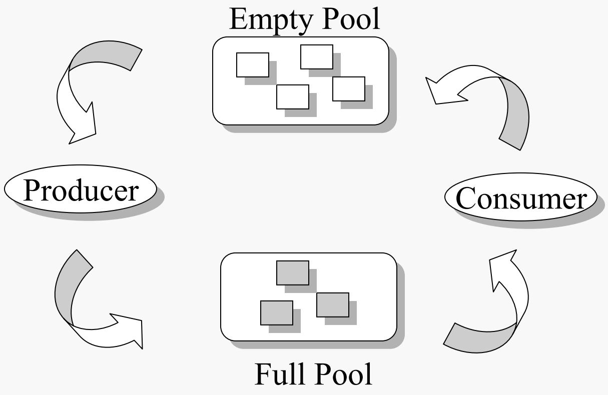
```
int x, y;
semaphore s1 = 0, s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

```
proc_A() {
  while (true) {
    <compute section A1>;
    update(x);
    // signal proc_B
    V(s1);
    <compute section A2>;
    // wait for proc_B
    P(s2);
    retrieve(y);
  }
}
```

```
proc_B() {
  while (true) {
    // wait for proc_A
    P(s1);
    retrieve(x);
    <compute section B1>;
    update(y);
    // signal proc_A
    V(s2);
    <compute section B2>;
  }
}
```

The semaphores are being used here in a more complex manner…

Bounded Buffer Problem



Empty Pool

Producer

Consumer

Full Pool

Bounded Buffer Problem (2)

```
semaphore mutex = 1;
semaphore full  = 0;     // A general (counting) semaphore
semaphore empty = N;     // A general (counting) semaphore
buf_type buffer[N];
```

```
producer() {
  buf_type *next, *here;
  while (true) {
    produce_item(next);
    // Claim an empty
    P(empty);
➡   P(mutex);
      here = obtain(empty);
    V(mutex);
    copy_buffer(next, here);
    P(mutex);
      release(here, fullPool);
    V(mutex);
    // Signal a full buffer
    V(full);
  }
}
```

```
consumer() {
  buf_type *next, *here;
  while (true) {
    // Claim full buffer
    P(mutex);
➡   P(full);
      here = obtain(full);
    V(mutex);
    copy_buffer(here, next);
    P(mutex);
      release(here, emptyPool);
    V(mutex);
    // Signal an empty buffer
    V(empty);
    consume_item(next);
  }
}
```

Bounded Buffer Problem (3)

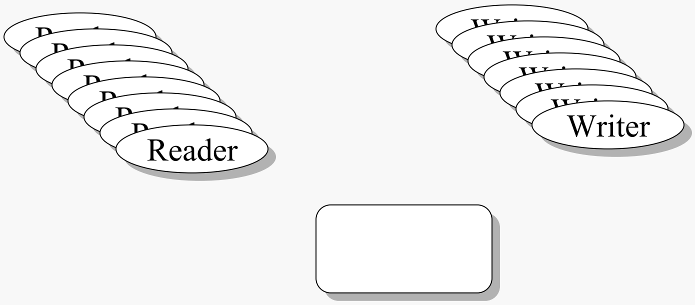
```
semaphore mutex = 1;
semaphore full  = 0;     // A general (counting) semaphore
semaphore empty = N;     // A general (counting) semaphore
buf_type buffer[N];
```

```
producer() {
  buf_type *next, *here;
  while (true) {
    produce_item(next);
    // Claim an empty
    P(empty);
    P(mutex);
      here = obtain(empty);
    V(mutex);
    copy_buffer(next, here);
    P(mutex);
      release(here, fullPool);
    V(mutex);
    // Signal a full buffer
    V(full);
  }
}
```

```
consumer() {
  buf_type *next, *here;
  while (true) {
    // Claim full buffer
  ↻ P(full);    ↺
    P(mutex);
      here = obtain(full);
    V(mutex);
    copy_buffer(here, next);
    P(mutex);
      release(here, emptyPool);
    V(mutex);
    // Signal an empty buffer
    V(empty);
    consume_item(next);
  }
}
```

Readers-Writers Problem

Readers-Writers Problem

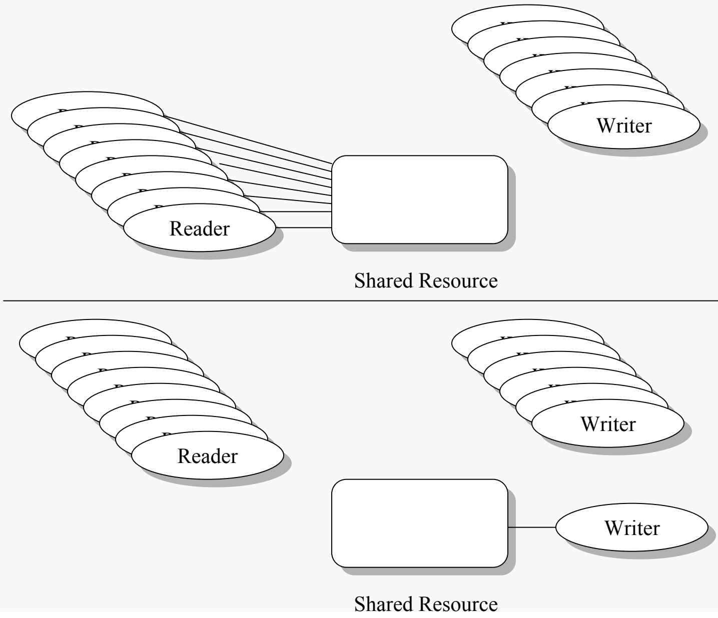


Shared Resource

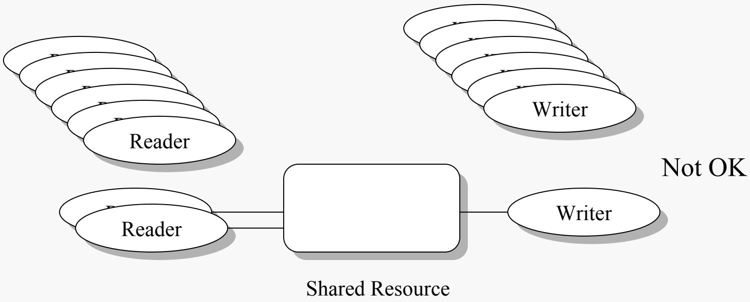
It's logically acceptable for an arbitrary number of readers to access the shared resource at the same time…

…but if a writer is accessing the shared resource, it's unsafe to allow any other reader or writer to access it at the same time.

Readers-Writers Problem



Readers-Writers Problem



This is simply a more complex version of the shared balance problem.

As before, unfortunate context switches between readers and the writer could lead to readers receiving incorrect data.

Similar issues arise with multiple writers.

First Solution

```
reader() {
  while (true) {
    <other computing>;
    P(mutex);              //  1
      readCount++;
      if (readCount == 1)  //  2
        P(writeBlock);     //  3
    V(mutex);              //  4
    // Critical section
    access(resource);      //  5
    P(mutex);              //  6
      readCount--;         //  7
      if (readCount == 0)  //  8
        V(writeBlock);     //  9
    V(mutex);              // 10
  }
}
```

```
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
fork(reader, 0);
fork(writer, 0);
```

```
writer() {
  while (true) {
    <other computing>;
    P(writeBlock);        // 1
    // Critical section
      access(resource);  // 2
    V(writeBlock);        // 3
  }
}
```

First reader competes with writers
Last reader signals writers

First Solution

```
reader() {
  while (true) {
    <other computing>;
    P(mutex);              //  1
      readCount++;
      if (readCount == 1)  //  2
        P(writeBlock);     //  3
    V(mutex);              //  4
    // Critical section
    access(resource);      //  5
    P(mutex);              //  6
      readCount--;         //  7
      if (readCount == 0)  //  8
        V(writeBlock);     //  9
    V(mutex);              // 10
  }
}
```

```
writer() {
  while (true) {
    <other computing>;
    P(writeBlock);        // 1
    // Critical section
      access(resource);  // 2
    V(writeBlock);        // 3
  }
}
```

Any writer must wait for all readers

Readers can starve writers

Updates can be delayed forever

May not be what we want

Writer Precedence v1

```
reader() {
  while (true) {
    <other computing>;

④➔  P(readBlock);               //  1
        P(mutex1);              //  2
          readCount++;          //  3
          if (readCount == 1)   //  4
  ②➔      P(writeBlock);        //  5
        V(mutex1);              //  6
①➔  V(readBlock);               //  7

      access(resource);         //  8
    P(mutex1);                  //  9
      readCount--;              // 10
      if (readCount == 0)       // 11
        V(writeBlock);          // 12
    V(mutex1);                  // 13
  }
}
```

```
writer() {
  while (true) {
    <other computing>;
    P(mutex2);               //  1
      writeCount++;          //  2
      if (writeCount == 1)   //  3
③➔      P(readBlock);        //  4
    V(mutex2);               //  5
    P(writeBlock);           //  6
      access(resource);      //  7
    V(writeBlock);           //  8
    P(mutex2);               //  9
      writeCount--;          // 10
      if (writeCount == 0)   // 11
        V(readBlock);        // 12
    V(mutex2);               // 13
  }
}
```

```
int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1;
```

Writer Precedence v2

```
reader() {
  while (true) {
    <other computing>;
    P(writePending);            //  1
      P(readBlock);             //  2
        P(mutex1);              //  3
          readCount++;          //  4
          if (readCount == 1)   //  5
            P(writeBlock);      //  6
        V(mutex1);              //  7
      V(readBlock);             //  8
    V(writePending);            //  9
      access(resource);         // 10
    P(mutex1);                  // 11
      readCount--;              // 12
      if (readCount == 0)       // 13
        V(writeBlock);          // 14
    V(mutex1);                  // 15
  }
}
```

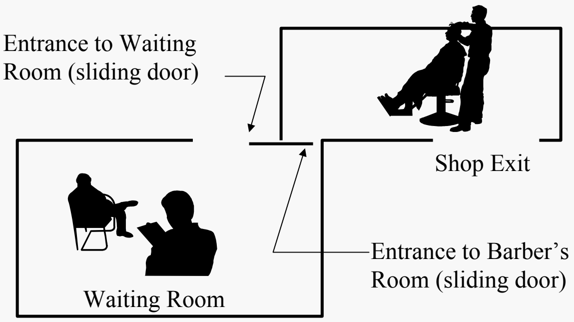
```
writer() {
  while (true) {
    <other computing>;
    P(mutex2);              //  1
      writeCount++;         //  2
      if (writeCount == 1)  //  3
        P(readBlock);       //  4
    V(mutex2);              //  5
    P(writeBlock);          //  6
      access(resource);     //  7
    V(writeBlock);          //  8
    P(mutex2)               //  9
      writeCount--;         // 10
      if (writeCount == 0)  // 11
        V(readBlock);       // 12
    V(mutex2);              // 13
  }
}
```

```
int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
```

The Sleepy Barber

Barber can cut one person's hair at a time

Other customers wait in a waiting room



Sleepy Barber (aka Bounded Buffer)

```
customer() {
  while (true) {
    customer = nextCustomer();      //  1
    if (emptyChairs == 0)           //  2
      continue;                     //  3
    P(chair);                       //  4
      P(mutex);                     //  5
        emptyChairs--;              //  6
        takeChair(customer);        //  7
      V(mutex);                     //  8
    V(waitingCustomer);             //  9
  }
}
```

```
barber() {
  while (true) {
    P(waitingCustomer);   //  1
      P(mutex);           //  2
        emptyChairs++;    //  3
        takeCustomer();   //  4
      V(mutex);           //  5
    V(chair);             //  6
  }
}
```

```
semaphore mutex = 1, chair = N, waitingCustomer = 0;
int emptyChairs = N;
```

Cigarette Smoker's Problem

Three smokers (processes)

Each wish to use tobacco, papers, & matches

- only need the three resources periodically
- must have all at once

3 processes sharing 3 resources

- solvable, but difficult

Implementing Semaphores

Minimize effect on the I/O system

Processes are only blocked on their own critical sections (not critical sections that they should not care about)

If disabling interrupts, be sure to bound the time they are disabled

`enter()` & `exit()`

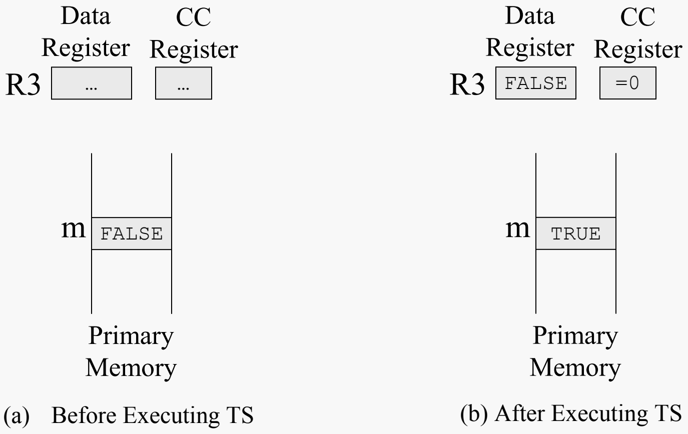
```
class semaphore {
private:
  int value;
public:
  semaphore(int v = 1) { value = v;}

  P(){
    disableInterrupts();
    while(value == 0) {
      enableInterrupts();
      disableInterrupts();
    }
    value--;
    enableInterrupts();
  }

  V(){
    disableInterrupts();
    value++;
    enableInterrupts();
  }
};
```

Test and Set Instruction

```
TS(m): [Reg_i = memory[m]; memory[m] = TRUE;]
// returned value is specified in control code reg
```



(a) Before Executing TS

(b) After Executing TS

Using the TS Instruction

```
bool s = false;           // access control is "open"
  . . .
  while (TS(s));          // first caller gets in, but
                          //   sets access control "closed"
    <critical section>
  s = false;              // set access control to "open"
  . . .
```

```
semaphore s = 1;
  . . .
  P(s);
    <critical section>
  V(s);
  . . .
```

Implementing the General Semaphore

```
struct semaphore {
  int  value = <initial value>;
  bool mutex = false;
  bool hold = true;
};

shared struct semaphore s;

P(struct semaphore s) {
  while (TS(s.mutex)) ;
  s.value--;
  if (s.value < 0) (
    s.mutex = false;
➡   while (TS(s.hold));
  }
  else
    s.mutex = false;
}
```

```
V(struct semaphore s) {
  while (TS(s.mutex));
  s.value++;
  if (s.value <= 0) (
➡   while (!s.hold);
    s.hold = false;
  }
  s.mutex = false;
}
```

Active vs Passive Semaphores

A process can dominate the semaphore

- performs V operation, but continues to execute
- performs another P operation before releasing the CPU
- called a passive implementation of V

Active implementation calls scheduler as part of the V operation.

- changes semantics of semaphore!
- cause people to rethink solutions