Basic Synchronization Principles
Encourage Concurrency

- No widely-accepted concurrent programming languages
- No concurrent programming paradigm
  - Each problem requires careful consideration
  - There is no common model
  - See SOR example on p 189 for one example
- OS tools to support concurrency are, of necessity, low level
Critical Sections

shared float balance;

Code for p₁:

... balance = balance + amount;
... 

Code for p₂:

... balance = balance - amount;
... 

p₁

balance

p₂
Critical Sections

shared double balance;

Code for p₁

. . .
balance = balance + amount;
. . .

Code for p₂

. . .
balance = balance - amount;
. . .

Code for p₁

load   R1, balance
load   R2, amount
add    R1, R2
store  R1, balance

Code for p₂

load   R1, balance
load   R2, amount
sub    R1, R2
store  R1, balance
Critical Sections (cont)

- There is a *race* to execute critical sections
- The sections may be different code in different processes
  - Cannot detect with static analysis
- Results of multiple execution are not *determinate*
- Need an OS mechanism to resolve races
Disabling Interrupts

shared double balance;

**Code for p₁**
disableInterrupts();
balance = balance + amount;
enableInterrupts();

**Code for p₂**
disableInterrupts();
balance = balance - amount;
enableInterrupts();
Disabling Interrupts

shared double balance;

**Code for p_1**

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

**Code for p_2**

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

- Interrupts could be disabled arbitrarily long
- Really only want to prevent \( p_1 \) and \( p_2 \) from interfering with one another
- Try using a shared “lock” variable
Using a Lock Variable

shared boolean lock = FALSE;
shared double balance;

**Code for p₁**
/* Acquire the lock */
while(lock) ;
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

**Code for p₂**
/* Acquire the lock */
while(lock) ;
lock = TRUE;
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
lock = FALSE;
Using a Lock Variable

shared boolean lock = FALSE;
shared double balance;

**Code for p₁**
/* Acquire the lock */
while(lock) ;
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

**Code for p₂**
/* Acquire the lock */
while(lock) ;
lock = TRUE;
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
lock = FALSE;
Using a Lock Variable

shared boolean 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;

• Worse yet … another race condition …
• Is it possible to solve the problem?
Lock Manipulation

```c
enter(lock) {
    disableInterrupts();
    /* Loop until lock is TRUE */
    while(lock) {
        /* Let interrupts occur */
        enableInterrupts();
        disableInterrupts();
    }
    lock = TRUE;
    enableInterrupts();
}
```

```c
exit(lock) {
    disableInterrupts();
    lock = FALSE;
    enableInterrupts();
}
```
Transactions

• A *transaction* is a list of operations
  – When the system begins to execute the list, it must execute all of them without interruption, or
  – It must not execute any at all

• Example: List manipulator
  – Add or delete an element from a list
  – Adjust the list descriptor, e.g., length
Processing Two Transactions

shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;

Code for \(p_1\)

\[
\begin{align*}
&\ldots \\
&\text{/* Enter CS to delete elt */} \\
&\text{enter(lock1);} \\
&\text{<delete element>;} \\
&\text{/* Exit CS */} \\
&\text{exit(lock1);} \\
&\text{<intermediate computation>;} \\
&\text{/* Enter CS to update len */} \\
&\text{enter(lock2);} \\
&\text{<update length>;} \\
&\text{/* Exit CS */} \\
&\text{exit(lock2);} \\
\end{align*}
\]

Code for \(p_2\)

\[
\begin{align*}
&\ldots \\
&\text{/* Enter CS to update len */} \\
&\text{enter(lock2);} \\
&\text{<update length>;} \\
&\text{/* Exit CS */} \\
&\text{exit(lock2);} \\
&\text{<intermediate computation>} \\
&\text{/* Enter CS to add elt */} \\
&\text{enter(lock1);} \\
&\text{<add element>;} \\
&\text{/* Exit CS */} \\
&\text{exit(lock1);} \\
\end{align*}
\]
Deadlock

shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;

**Code for p₁**

```c
... /* Enter CS to delete elt */
    enter(lock1);
    <delete element>;
    <intermediate computation>;
/* Enter CS to update len */
    enter(lock2);
    <update length>;
/* Exit both CS */
    exit(lock1);
    exit(lock2);
...```

**Code for p₂**

```c
... /* Enter CS to update len */
    enter(lock2);
    <update length>;
    <intermediate computation>
/* Enter CS to add elt */
    enter(lock1);
    <add element>;
/* Exit both CS */
    exit(lock2);
    exit(lock1);
...```
Coordinating Processes

• Can synchronize with \texttt{FORK}, \texttt{JOIN} \& \texttt{QUIT}
  – Terminate processes with \texttt{QUIT} to synchronize
  – Create processes whenever critical section is complete
  – See Figure 8.7

• Alternative is to create OS primitives similar to the \texttt{enter/exit} primitives
Some Constraints

- Processes $p_0$ & $p_1$ enter critical sections
- *Mutual exclusion*: Only one process at a time in the CS
- 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
Some Language

• Let \( \text{fork}(\text{proc, N, arg}_1, \text{ arg}_2, \ldots, \text{ arg}_N) \) be a command to create a process, and to have it execute using the given \( N \) arguments

• Canonical problem:

\[
\begin{align*}
\text{Proc}_0() \{ & \quad \text{proc}_1() \{ \\
& \text{while(\text{TRUE}) } \{ \\
& \quad \text{<compute section>}; \\
& \quad \text{<critical section>}; \\
& \} \\
& \}
\}
\end{align*}
\]

<shared global declarations>
<initial processing>
fork(proc_0, 0);
fork(proc_1, 0);
Assumptions About Solutions

- 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

• Classic paper describes several software attempts to solve the problem (see problem 4, Chapter 8)
• 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 functions:

\[ V(s): [s = s + 1] \]
\[ P(s): [\text{while}(s == 0) \{ \text{wait} \}; s = s - 1] \]
Using Semaphores to Solve the Canonical Problem

Proc_0() {
    while(TRUE) {
        <compute section>;
        P(mutex);
        <critical section>;
        V(mutex);
    }
}

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

semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
Shared Account Problem

Proc_0() {  
  . . .  
  /* Enter the CS */  
  P(mutex);  
  balance += amount;  
  V(mutex);  
  . . .  
}

semaphore mutex = 1;

fork(proc_0, 0);
fork(proc_1, 0);

proc_1() {  
  . . .  
  /* Enter the CS */  
  P(mutex);  
  balance -= amount;  
  V(mutex);  
  . . .  
}
Two Shared Variables

```c
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);
    }
}

semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);

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 Driver-Controller Interface

• The semaphore principle is logically used with the busy and done flags in a controller.

• Driver signals controller with a $V(busy)$, then waits for completion with $P(done)$.

• Controller waits for work with $P(busy)$, then announces completion with $V(done)$.

• See In the Cockpit, page 204.
Bounded Buffer

Empty Pool

Full Pool

Producer

Consumer
Bounded Buffer

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);
    }
}

semaphore mutex = 1;
semaphore full = 0;    /* A general (counting) semaphore */
semaphore empty = N;  /* A general (counting) semaphore */
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);

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

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);
    }
}

semaphore mutex = 1;
semaphore full = 0;   /* A general (counting) semaphore */
semaphore empty = N;  /* A general (counting) semaphore */
buf_type buffer[N];   
fork(producer, 0);
fork(consumer, 0);
Readers-Writers Problem

Reader

Shared Resource

Writer

Shared Resource
Readers-Writers Problem
Readers-Writers Problem

Reader

Writer

Shared Resource
First Solution

```c
reader() {
    while(TRUE) {
        <other computing>;
        P(mutex);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex);
        /* Critical section */
        access(resource);
        P(mutex);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        P(writeBlock);
        /* Critical section */
        access(resource);
        V(writeBlock);
    }
}

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

```c
reader() {
    while(TRUE) {
        <other computing>;
        P(mutex);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex);
        /* Critical section */
        access(resource);
        P(mutex);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        P(writeBlock);
        /* Critical section */
        access(resource);
        V(writeBlock);
    }
}

• First reader competes with writers
• Last reader signals writers
```

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

```
reader() {
    while(TRUE) {
        <other computing>;
        P(mutex);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex);
    /* Critical section */
        access(resource);
        P(mutex);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex);
    }

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

writer() {
    while(TRUE) {
        <other computing>;
        P(writeBlock);
    /* Critical section */
        access(resource);
        V(writeBlock);
    }
}
```

- First reader competes with writers
- Last reader signals writers
- Any writer must wait for all readers
- Readers can starve writers
- "Updates" can be delayed forever
- May not be what we want
Writer Takes Precedence

```c
reader() {
    while(TRUE) {
        <other computing>;

        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);

        access(resource);
        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
        V(readBlock);

        access(resource);
        P(mutex1);
        readCount--;  
        if(readCount == 0) 
            V(writeBlock);
        V(mutex1);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);
```

```c
writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        writeCount++;
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);
        P(writeBlock);
        access(resource);
        access(resource);
        P(mutex2);
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}
```
Writer Takes Precedence

reader() {
    while(TRUE) {
        <other computing>;

        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);

        access(resource);
        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);

writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        writeCount++;
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);
        P(writeBlock);
        access(resource);
        V(writeBlock);
        P(mutex2)
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}
Writer Takes Precedence

```c
reader() {
    while(TRUE) {
        <other computing>;

        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);

        access(resource);
        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);

writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        writeCount++;
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);
        P(writeBlock);
        access(resource);
        V(writeBlock);
        P(mutex2)
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}
```

Writer Takes Precedence

reader() {
    while(TRUE) {
        <other computing>;

        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);

        access(resource);
        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);

writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        writeCount++;
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);
        P(writeBlock);
        access(resource);
        V(writeBlock);
        P(mutex2)
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}
Writer Takes Precedence

reader() {
    while(TRUE) {
        <other computing>;

        4. P(readBlock);
        4. P(mutex1);
        4. readCount++;
        4. if(readCount == 1)
            4. P(writeBlock);
        4. V(mutex1);
        4. V(readBlock);

        2. access(resource);
        2. P(mutex1);
        2. readCount--;
        2. if(readCount == 0)
            2. V(writeBlock);
        2. V(mutex1);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);

writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        3. writeCount++;
        3. if(writeCount == 1)
            3. P(readBlock);
        3. V(mutex2);
        3. P(writeBlock);
        3. access(resource);
        3. V(writeBlock);
        3. P(mutex2)
        3. writeCount--;
        3. if(writeCount == 0)
            3. V(readBlock);
        3. V(mutex2);
    }
}
readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);
Sleepy Barber Problem

- Barber can cut one person’s hair at a time
- Other customers wait in a waiting room
Sleepy Barber Problem
(Bounded Buffer Problem)

customer() {
    while(TRUE) {
        customer = nextCustomer();
        if(emptyChairs == 0)
            continue;
        P(chair);
        P(mutex);
        emptyChairs--;
        takeChair(customer);
        V(mutex);
        V(waitingCustomer);
    }
}

barber() {
    while(TRUE) {
        P(waitingCustomer);
        P(mutex);
        emptyChairs++;
        takeCustomer();
        V(mutex);
        V(chair);
    }
}

semaphore mutex = 1, chair = N, waitingCustomer = 0;
int emptyChairs = N;
fork(customer, 0);
fork(barber, 0);
Dining Philosophers

while(TRUE) {
    think();
    eat();
}
Cigarette Smokers’ 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
Implementing Semaphores: Disabling Interrupts

class semaphore {
    int value;
public:
    semaphore(int v = 1) { value = v;};
P(){
        disableInterrupts();
        while(value == 0) {
            enableInterrupts();
            disableInterrupts();
        }
        value--;
        enableInterrupts();
    }
    V(){
        disableInterrupts();
        value++;  
        value++;
        enableInterrupts();
    }
};
Implementing Semaphores: Test and Set Instruction

- TS(m): \([\text{Reg}_i = \text{memory}[m]; \text{memory}[m] = \text{TRUE};]\)

```java
boolean s = FALSE;
...
while(TS(s)) ;
<critical section>
s = FALSE;
...
```

```java
semaphore s = 1;
...
P(s) ;
<critical section>
V(s);
...
```
**General Semaphore**

struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean 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;
    } else
        s.mutex = FALSE;
}
General Semaphore

```c
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean 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;
}
```

• Block at arrow
• Busy wait
General Semaphore

struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean 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;
    } else
        s.mutex = FALSE;
}

• Block at arrow
• Busy wait
• Quiz: Why is this statement necessary?
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