Large Amount of Work to Do

Partition the Work

Define Dependencies

Assign Work to Threads

Synchronize Thread Execution

The Dining Philosophers
First Try Solution

```c
philosopher(int i) {
    // Think
    // Eat
    P(fork[i]);
    P(fork[(i+1) mod 5]);
    eat();
    V(fork[(i+1) mod 5]);
    V(fork[i]);
}
```

```c
semaphore fork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

One “Answer?”

```c
philosopher(int i) {
    // Think
    // Eat
    j = i % 2;
    P(fork[(i+j) mod 5]);
    P(fork[(i+1-j) mod 5]);
    eat();
    V(fork[(i+1-j) mod 5]);
    V(fork[(i+j) mod 5]);
}
```

```c
semaphore fork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

Nesting Semaphore Operations

<table>
<thead>
<tr>
<th>P Operation Order</th>
<th>P Operation Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(mutex1);</td>
<td>P(mutex2);</td>
</tr>
<tr>
<td>P(mutex2);</td>
<td>P(mutex1);</td>
</tr>
<tr>
<td>&lt;access R1&gt;;</td>
<td>&lt;access R1&gt;;</td>
</tr>
<tr>
<td>&lt;access R2&gt;;</td>
<td>&lt;access R2&gt;;</td>
</tr>
<tr>
<td>V(mutex2);</td>
<td>V(mutex1);</td>
</tr>
<tr>
<td>V(mutex1);</td>
<td>V(mutex2);</td>
</tr>
</tbody>
</table>

(a) (b)

Abstracting Semaphores

- Relatively simple problems, such as the dining philosophers problem, can be very difficult to solve
- Look for abstractions to simplify solutions
  - AND synchronization
  - Events
  - Monitors
  - … there are others ...

AND Synchronization

- Given two resources, R₁ and R₂
- Some processes access R₁, some R₂, some both in the same critical section
- Need to avoid deadlock due to ordering of P operations
- \( P_{\text{simultaneous}}(S_1, \ldots, S_n) \)

Simultaneous Semaphores Def

```c
P_sim(semaphore S, int N) {
    L1: if ((S[0]>=1)&\& \ldots \&\&(S[N-1]>=1)) {
        for(i=0; i<N; i++) S[i]--;
    } else {
        Enqueue the calling thread in the queue for the first S[i] where S[i]<1;
        The calling thread is blocked while it is in the queue;
        // when the thread is removed from the queue
        Goto L1;
    }
}
```

```c
V_sim(semaphore S, int N) {
    for(i=0; i<N; i++) {
        S[i]++;
    }
    // Dequeue all threads in the queue for S[i];
    All such threads are now ready to run
    (but may be blocked again in P_sim);
}
```
Simultaneous Semaphore

```c
int R_num = 0, S_num = 0;
Queue R_wait, S_wait;
Semaphore mutex = 1;
P_sim(PID callingThread, semaphore R, semaphore S) {
    L1: P(mutex);
    if(R.val>0)&&(S.val>0)) {
        P(R); P(S);
        V(mutex);
    } else {
        if(R.val==0) {
            R_num++;
            enqueue(callingThread, R_wait);
            V(mutex);
            goto L1;
        } else {
            S_num++;
            enqueue(callingThread, S_wait);
            V(mutex);
            goto L1;
        }
    }
}
V_sim(semaphore R, semaphore S) {
    P(mutex);
    V(R); V(S);
    if(R_num>0) {
        R_num--;
        dequeue(R_wait);  // Release a thread
    }
    if(S_num>0) {
        S_num--;
        dequeue(S_wait);  // Release a thread
    }
    V(mutex);
}
```

Dining Philosophers Problem

```c
philosopher(int i) {
    while(TRUE) {
        // Think
        P_sim(fork[i], fork [(i+1) mod 5]);
        eat();
        V_sim(fork[i], fork [(i+1) mod 5]);
    }
}
semaphore fork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

Events

- Exact definition is specific to each OS
- A process can wait on an event until another process signals the event
- Have event descriptor (“event control block”)
- Active approach
  - Multiple processes can wait on an event
  - Exactly one process is unblocked when a signal occurs
  - A signal with no waiting process is ignored
- May have a queue function that returns number of processes waiting on the event

UNIX Signals

- A UNIX signal corresponds to an event
  - It is raised by one process (or hardware) to call another process’s attention to an event
  - It can be caught (or ignored) by the subject process
- Justification for including signals was for the OS to inform a user process of an event
  - User pressed delete key
  - Program tried to divide by zero
  - Attempt to write to a nonexistent pipe
  - etc.

Dining Philosophers Problem

```c
philosopher(int i) {
    while(TRUE) {
        // Think
        P_sim(fork[i], fork [(i+1) mod 5]);
        eat();
        V_sim(fork[i], fork [(i+1) mod 5]);
    }
}
semaphore fork[5] = (1,1,1,1,1);
fork(philosopher, 1, 0);
fork(philosopher, 1, 1);
fork(philosopher, 1, 2);
fork(philosopher, 1, 3);
fork(philosopher, 1, 4);
```

More on Signals

- Each version of UNIX has a fixed set of signals (Linux has 31+ of them)
- `signal.h` defines the signals in the OS
- App programs can use SIGUSR1 & SIGUSR2 for arbitrary signalling
- Raise a signal with `kill(pid, signal)`
- Process can let default handler catch the signal, catch the signal with own code, or cause it to be ignored
More on Signals (cont)

- OS signal system call
  - To ignore: `signal(SIG#, SIG_IGN)`
  - To reinstate default: `signal(SIG#, SIG_DFL)`
  - To catch: `signal(SIG#, myHandler)`
- Provides a facility for writing your own event handlers in the style of interrupt handlers

Signal Handling

```c
/* code for process p */
void sig_hndlr(...) {
    /* ARBITRARY CODE */
}
```

An executing process, q

Raise “SIG#” for “p”

q is blocked

sig_hndlr runs in p’s address space

q resumes execution

Using UNIX Signals

![Diagram showing process address space and signal handling]

Example: Shared Balance

```c
monitor sharedBalance {
    double balance;
    public:
        credit(double amount) {balance += amount;};
        debit(double amount) {balance -= amount;};
}
```

Example: Readers & Writers

```c
monitor readerWriter_1 {
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    public:
        startRead() {
            while(TRUE) {
                ... startRead(); ... startWriter(); ... finishRead(); ... finishWriter(); ...
            }
        }
        fork(reader, 0); ... fork(writer, 0);
}
```
Example: Readers & Writers

```c
monitor readerWriter_1 {
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    public:
        startRead() {
            while(numberOfWriters != 0) ;
            numberOfReaders++;
        }
        finishRead() {
            numberOfReaders--;
        }
        startWrite() {
            numberOfWriters++;
            while( busy || (numberOfReaders > 0) ) ;
            busy = TRUE;
        }
        finishWrite() {
            numberOfWriters--;
            busy = FALSE;
        }
};
```

• Deadlock can happen

Sometimes Need to Suspend

• Process obtains monitor, but detects a condition for which it needs to wait
• Want special mechanism to suspend until condition is met, then resume
  – Process that makes condition true must exit monitor
  – Suspended process then resumes
• **Condition Variable**

Condition Variables

• Essentially an event (as defined previously)
• Occurs only inside a monitor
• Operations to manipulate condition variable
  – wait: Suspend invoking process until another executes a signal
  – signal: Resume one process if any are suspended, otherwise do nothing
  – queue: Return TRUE if there is at least one process suspended on the condition variable

Active vs Passive signal

• Hoare semantics: same as active semaphore
  – p₀ executes signal while p₁ is waiting ⇒ p₀ yields the monitor to p₁
  – The signal is only TRUE the instant it happens
• Brinch Hansen (“Mesa”) semantics: same as passive semaphore
  – p₀ executes signal while p₁ is waiting ⇒ p₀ continues to execute, then when p₀ exits the monitor p₁ can receive the signal
  – Used in the Xerox Mesa implementation

Hoare vs Mesa Semantics

• Hoare semantics:
  ```java
  if(!resourceNotAvailable()) resourceCondition.wait();
  /* now available ... continue ... */
  ```
• Mesa semantics:
  ```java
  while(!resourceNotAvailable()) resourceCondition.wait();
  /* now available ... continue ... */
  ```
Another Try at Readers & Writers

```c
monitor readerWriter_2 {
  int numberOfReaders = 0;
  boolean busy = FALSE;
  condition okToRead, okToWrite;

public:
  startRead() {
    if(busy || (okToWrite.queue()))
      okToRead.wait();
    numberOfReaders++;
    okToRead.signal();
  }
  finishRead() {
    numberOfReaders--;
    if(numberOfReaders == 0)
      okToWrite.signal();
  }
  startWrite() {
    if((numberOfReaders != 0) || busy)
      okToWrite.wait();
    busy = TRUE;
  }
  finishWrite() {
    busy = FALSE;
    if(okToRead.queue())
      okToRead.signal();
    else
      okToWrite.signal();
  }
};
```

Dining Philosophers … again ...

```c
#define N ___
enum status(EATING, HUNGRY, THINKING);
monitor diningPhilosophers {
  status state[N];
  condition self[N];
  test(int i) {
    if((state[(i-1) mod N] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i+1) mod N] != EATING)) {
      state[i] = EATING;
      self[i].signal();
    }
  }

  public:
    diningPhilosophers() { // Initialization
      for(int i = 0; i < N; i++) state[i] = THINKING;
    }
    pickUpForks(int i) {
      state[i] = HUNGRY;
      test(i);
      if(state[i] != EATING) self[i].wait();
    }
    putDownForks(int i) {
      state[i] = THINKING;
      test((i-1) mod N);
      test((i+1) mod N);
    }
};
```

Experience with Monitors

- Danger of deadlock with nested calls
- Monitors were implemented in Mesa
  - Used Brinch Hansen semantics
  - Nested monitor calls are, in fact, a problem
  - Difficult to get the right behavior with these semantics
  - Needed timeouts, aborts, etc.

Refined IPC Mechanism

- Spontaneous changes to p₁’s address space
- Avoid through the use of mailboxes

- Must bypass memory protection mechanism for local copies
- Must be able to use a network for remote copies

```
OS IPC

<table>
<thead>
<tr>
<th>Info to be shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Space for p₀</td>
</tr>
<tr>
<td>Message</td>
</tr>
<tr>
<td>Info copy</td>
</tr>
<tr>
<td>Address Space for p₁</td>
</tr>
</tbody>
</table>

OS Interface

<table>
<thead>
<tr>
<th>send function</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(… p₁, …);</td>
</tr>
<tr>
<td>Mailbox for p₁</td>
</tr>
<tr>
<td>Message</td>
</tr>
<tr>
<td>Info copy</td>
</tr>
<tr>
<td>receive function</td>
</tr>
<tr>
<td>receive(…);</td>
</tr>
</tbody>
</table>
```
**Refined IPC Mechanism**
- OS manages the mailbox space
- More secure message system

**Interprocess Communication (IPC)**
- Signals, semaphores, etc. do not pass information from one process to another
- Monitors support information sharing, but only through shared memory in the monitor
- There may be no shared memory
  - OS does not support it
  - Processes are on different machines on a network
- Can use messages to pass info while synchronizing

**Message Protocols**
- Sender transmits a set of bits to receiver
  - How does the sender know when the receiver is ready (or when the receiver obtained the info)?
  - How does the receiver know how to interpret the info?
  - Need a protocol for communication
    - Standard "envelope" for containing the info
    - Standard header
  - A message system specifies the protocols

**Transmit Operations**
- Asynchronous send:
  - Delivers message to receiver’s mailbox
  - Continues execution
  - No feedback on when (or if) info was delivered
- Synchronous send:
  - Goal is to block sender until message is received by a process
    - Variant sometimes used in networks: Until the message is in the mailbox

**Receive Operation**
- Blocking receive:
  - Return the first message in the mailbox
  - If there is no message in mailbox, block the receiver until one arrives
- Nonblocking receive:
  - Return the first message in the mailbox
  - If there is no message in mailbox, return with an indication to that effect

**Synchronized IPC**

```c
/* signal p */
syncSend(message1, p2);
<waiting ...>
/* wait for signal from p */
blockReceive(msgBuff, &from);
<process message>

/* signal p */
blockReceive(msgBuff, &from);
<syncSend(message2, p1)>

syncSend(...) blockReceive(...)
blockReceive(...) syncSend(....)
```
Asynchronous IPC

Code for p₁:

/* signal p₂ */
asyncSend(message₁, p₂);
<other processing>
/* test for signal from p₂ */
if(nbReceive(&msg, &from)) {
  <process message>
  asyncSend(message₁, p₂);
} else {
  <other processing>
}
asyncSend(…)
nonblockReceive(…)
nonblockReceive(…)
nonblockReceive(…)

Code for p₂:

Asynchronous IPC

UNIX Pipes

UNIX Pipes (cont)

- The pipe interface is intended to look like a file interface
  - Analog of open is to create the pipe
  - File read/write system calls are used to send/receive information on the pipe
- What is going on here?
  - Kernel creates a buffer when pipe is created
  - Processes can read/write into/out of their address spaces from/to the buffer
  - Processes just need a handle to the buffer

UNIX Pipes (cont)

- File handles are copied on fork
- … so are pipe handles

UNIX Pipes (cont)

• The normal write is an asynchronous op (that notifies of write errors)
• The normal read is a blocking read
• The read operation can be nonblocking

#include <sys/ioctl.h>

int pipeID[2];
pipe(pipeID);

if(fork() == 0) { /* the child */
  read(pipeID[0], childBuf, len);
  <process the message>;
} else { /* the parent */
  write(pipeID[1], msgToChild, len);
}
Information Flow Through UNIX Pipes

- Info to be shared
- write(pipe[1])
- Address Space for p_i
- Info copy
- read(pipe[0])
- System Call Interface

Pipe for p_i and p_j

write function → read function