High-Level Synchronization
Dining Philosophers

while (TRUE) {
    think();
    eat();
}

Quiz: Write a synchronization schema for the problem
Dining Philosophers Problem

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

philosopher(int i) {
    while(TRUE) {
        // 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]);
    }
}

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);
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, $R_1$ and $R_2$
• Some processes access $R_1$, some $R_2$, some both in the same critical section
• Need to avoid deadlock due to ordering of P operations
• $P_{\text{simultaneous}}(S_1, \ldots, S_n)$
AND Synchronization (cont)

semaphore mutex = 1;
semaphore block = 0;

P.sim(int S, int R) {
    P(mutex);
    S--;
    R--;
    if((S < 0) || (R < 0)) {
        V(mutex);
        P(block);
    } else 
        V(mutex);
}

V.sim(int S, int R) {
    P(mutex);
    S++;
    R++;
    if(((S >= 0) && ((S == 0) || (R == 0))) &&
       ((R >= 0) && ((S == 0) || (R == 0))))
       V(block);
    V(mutex);
}
Dining Philosophers Problem

philosopher(int i) {
    while(TRUE) {
        // Think
        // Eat
        P_{\text{simultaneous}}(fork[i], fork [(i+1) mod 5]);
        eat();
        V_{\text{simultaneous}}(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

- May mean different things in each OS
- A process can \textit{wait} on an event until another process \textit{signals} the event
- Have \textit{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 \textit{queue} function that returns number of processes waiting on the event
Example

class Event {
  ...
public:
  void signal();
  void wait();
  int queue();
}

shared Event topOfHour;
...

// Wait until the top of the hour before proceeding
topOfHour.wait();
// It’s the top of the hour ...

while(TRUE)
  if(isTopOfHour())
    while(topOfHour.queue() > 0)
      topOfHour.signal();
}
...

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.
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

/* code for process p
   . . .
signal(SIG#, sig_hndlr);
   . . .
/* ARBITRARY CODE */

void sig_hndlr(...) {
    /* ARBITRARY CODE */
}
Signal Handling

/* code for process p
 . . .
signal(SIG#, sig_hndlr);
 . . .
/* ARBITRARY CODE */

An executing process, q
Raise “SIG#” for “p”
q is blocked
q resumes execution

void sig_hndlr(...)
{
/* ARBITRARY CODE */
}
#include <signal.h>

static void sig_handler(int);

int main () {
    int i, parent_pid, child_pid, status;
    if(signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("Parent: Unable to create handler for SIGUSR1\n");
    if(signal(SIGUSR2, sig_handler) == SIG_ERR)
        printf("Parent: Unable to create handler for SIGUSR2\n");
    parent_pid = getpid();
    if((child_pid = fork()) == 0) {
        kill(parent_pid, SIGUSR1);
        for (;;) pause();
    } else {
        kill(child_pid, SIGUSR2);
        printf("Parent: Terminating child ... \n");
        kill(child_pid), SIGTERM);
        wait(&status);
        printf("done\n");
    }
}

Toy Signal Handler (Fig 9.4)
static void sig_handler(int signo) {
    switch(signo) {
    case SIGUSR1: /* Incoming SIGUSR1 */
        printf("Parent: Received SIGUSER1\n");
        break;
    case SIGUSR2: /* Incoming SIGUSR2 */
        printf("Child: Received SIGUSER2\n");
        break;
    default: break;
    }
    return
}
NT Events

Signal
- Implicitly
- Manually
- Callbacks

WaitForSingleObject(foo, time);

Signaled/not signaled flag

Kernel object
Waitable timer

Thread
NT Events

Thread

WaitForSingleObject(foo, time);

Thread

WaitForSingleObject(foo, time);

Thread

WaitForSingleObject(foo, time);

Thread

WaitForSingleObject(foo, time);
NT Events

Thread

WaitForMultipleObjects(foo, time);
Monitors

• Specialized form of ADT
  – Encapsulates implementation
  – Public interface (types & functions)

• Only one process can be executing in the ADT at a time

```cpp
monitor anADT {
    semaphore mutex = 1; // Implicit
    ...
    public:
    proc_i(...) {
        P(mutex); // Implicit
        <processing for proc_i>;
        V(mutex); // Implicit
    }
    ...
}
```
Example: Shared Balance

monitor sharedBalance {
    double balance;
public:
    credit(double amount) {balance += amount;};
    debit(double amount) {balance -= amount;};
    ...;
};
Example: Readers & Writers

monitor readerWriter_1 {
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;

public:
    startRead() {
    }
    finishRead() {
    }
    startWrite() {
    }
    finishWrite() {
    }
};

reader()
while(TRUE) {
   ...
    startRead();
    finishRead();
    ...
}
fork(reader, 0);
...
fork(reader, 0);
fork(writer, 0);
...
fork(writer, 0);

writer()
while(TRUE) {
    ...
    startWriter();
    finishWriter();
    ...
}

fork(reader, 0);
...
fork(reader, 0):
fork(writer, 0);
...
fork(writer, 0);
Example: Readers & Writers

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;
    }
};
Example: Readers & Writers

monitor readerWriter_1 {
  int numberOfReaders = 0;
  int numberOfWriters = 0;
  boolean busy = FALSE;

  public:
    startRead() {
      while(numberOfWriters != 0) {
        numberOfReaders++;
      }
    }
    finishRead() {
      numberOfReaders--;
    }
}

• Deadlock can happen

  startWrite() {
    numberOfWriters++;
  }
}

  startWrite() {
    numberOfWriters++;
    while(
      busy ||
      (numberOfReaders > 0)
    ) {
      busy = TRUE;
    }
    busy = TRUE;
  }
}

  finishWrite() {
    numberOfWriters--;
    busy = FALSE;
  }
};
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
  - $p_0$ executes signal while $p_1$ is waiting?
  - $p_0$ yields the monitor to $p_1$
  - The signal is only TRUE the instant it happens

- Brinch Hansen ("Mesa") semantics: same as passive semaphore
  - $p_0$ executes signal while $p_1$ is waiting?
  - $p_0$ continues to execute, then when $p_0$ exits the monitor $p_1$ 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 ... */
```
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()
    }
};
Example: Synchronizing Traffic

- One-way tunnel
- Can only use tunnel if no oncoming traffic
- OK to use tunnel if traffic is already flowing the right way
monitor tunnel {
    int northbound = 0, southbound = 0;
    trafficSignal nbSignal = RED, sbSignal = GREEN;
    condition busy;
    public:
    nbArrival() {
        if(southbound > 0) busy.wait();
        northbound++;
        nbSignal = GREEN; sbSignal = RED;
    }
    sbArrival() {
        if(northbound > 0) busy.wait();
        southbound++;
        nbSignal = RED; sbSignal = GREEN;
    }
    depart(Direction exit) {
        if(exit = NORTH {
            northbound--;
            if(northbound == 0) while(busy.queue()) busy.signal();
        else if(exit == SOUTH) {
            southbound--;
            if(southbound == 0) while(busy.queue()) busy.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;
    }
};
```
Dining Philosophers ... again ...

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() {...};
    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.

• See paper by Lampson & Redell
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
IPC Mechanisms

• Must bypass memory protection mechanism for local copies
• Must be able to use a network for remote copies
Refined IPC Mechanism

- Spontaneous changes to $p_1$’s address space
- Avoid through the use of mailboxes

Address Space for $p_0$

Info to be shared

send(... $p_1$, ...);

Address Space for $p_1$

Mailbox for $p_1$

Message

Info copy

receive(...);

OS Interface

send function

receive function
Refined IPC Mechanism

- OS manages the mailbox space
- More secure message system

Address Space for $p_0$

Info to be shared

```
send(... p_1, ...);
```

Address Space for $p_1$

Info copy

```
receive(...);
```

OS Interface

```
send function
```

```
Message
```

```
receive function
```
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

Code for p₁

/* signal p₂ */
syncSend(message₁, p₂);
<waiting ...>;

/* wait for signal from p₁ */
blockReceive(msgBuff, &from);

/* wait for signal from p₂ */
blockReceive(msgBuff, &from);

/* signal p₁ */
syncSend(message₂, p₁);

Code for p₂

/* signal p₂ */
syncSend(message₁, p₂);

/* wait for signal from p₁ */
blockReceive(msgBuff, &from);

/* process message */

/* signal p₁ */
syncSend(message₂, p₁);

/* wait for signal from p₂ */
blockReceive(msgBuff, &from);
Asynchronous IPC

/* signal p_2 */
asyncSend(message_1, p_2);
<other processing>;
/* wait for signal from p_2 */
while(!nbReceive(&msg, &from));

/* test for signal from p_1 */
if(nbReceive(&msg, &from)) {
    <process message>;
    asyncSend(message_2, p_1);
} else {
    <other processing>;
}

asyncSend(…)
nonblockReceive(…)
nonblockReceive(…)
nonblockReceive(…)
nonblockReceive(…)
asyncSend(…)

UNIX Pipes

Info to be shared

write(pipe[1], ...);

System Call Interface

pipe for p₁ and p₂

write function

Info copy

read(pipe[0]);

read function
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

```c
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);
  .
}
```
UNIX Pipes (con)

• 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

```c
#include <sys/ioctl.h>
.
int pipeID[2];
.
pipe(pipeID);
ioctl(pipeID[0], FIONBIO, &on);
.
read(pipeID[0], buffer, len);
if(errno != EWOULDBLOCK) {
    /* no data */
} else { /* have data */
```
Explicit Event Ordering

• Alternative technique of growing importance in network systems

• Rely on knowing the relative order of occurrence of every event
  – (occurrence of y in p_j) < (occurrence of x in p_i)
  – Then can synchronize by explicitly specifying each relation (when it is important)

  advance(eventCount) : Announces the occurrence of an event related to eventCount, causing it to be incremented by 1

  await(eventCount, v) : Causes process to block as long as eventCount < v.
Bounded Buffer

```c
producer() {
    int i = 1;
    while(TRUE) {
        await(out, i-N);
        produce(buffer[(i-1)mod N]);
        advance(in);
        i++;
    }
}

consumer() {
    int i = 1;
    while(TRUE) {
        await(in, i);
        consume(buffer[(i-1)mod N]);
        advance(out);
        i++;
    }
}

eventcount in = 0; out = 0;
fork(producer, 0);
fork(consumer, 0);
```
More on EventCounts

• Notice that **advance** and **await** need not be uninterruptible

• There is no requirement for shared memory

• For full use of this mechanism, actually need to extend it a bit with a sequencer

• Underlying theory is also used to implement “virtual global clocks” in a network

• Emerging as a preferred synchronization mechanism on networks