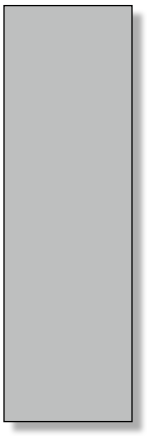
Slide 9-1

High Level Synchronization & Interprocess Communication

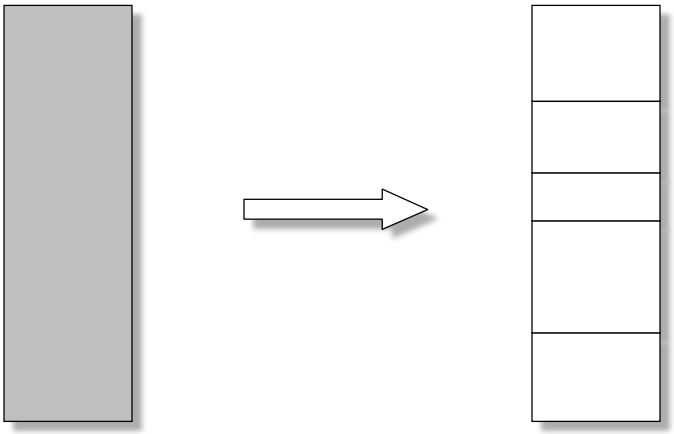
Slide 9-2

Large Amount of Work to Do



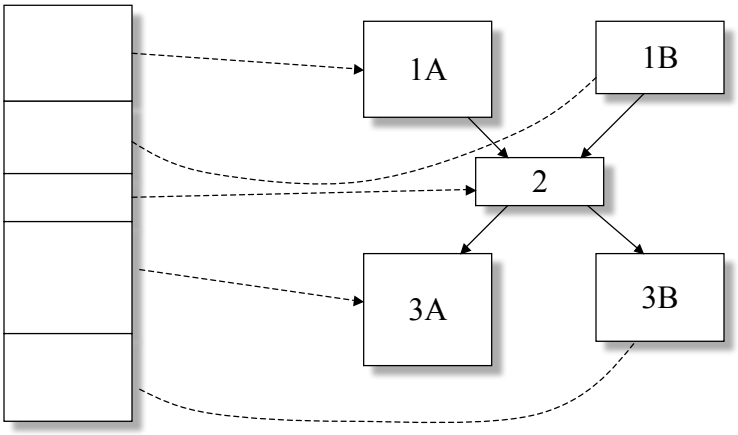
Partition the Work

Slide 9-3



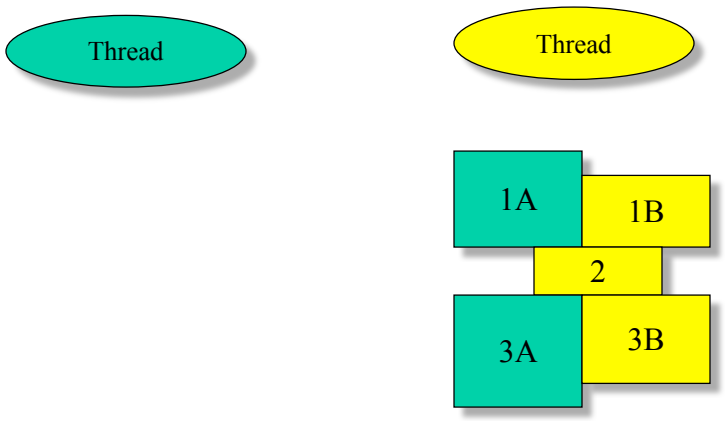
Define Dependencies

Slide 9-4



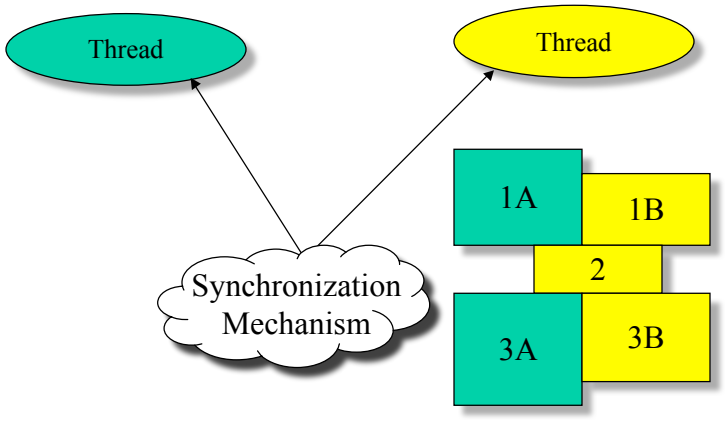
Assign Work to Threads

Slide 9-5

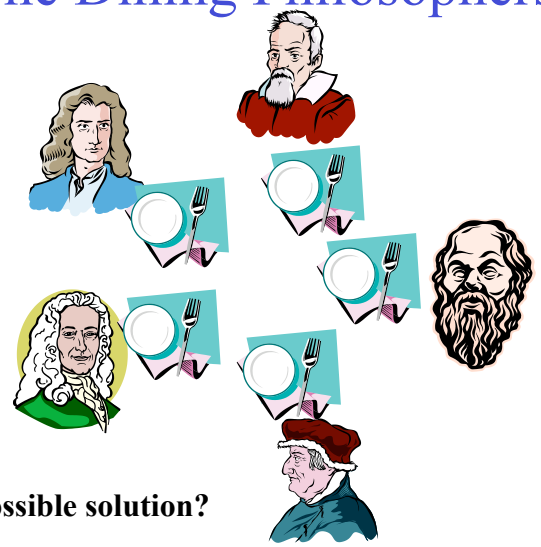


Synchronize Thread Execution

Slide 9-6



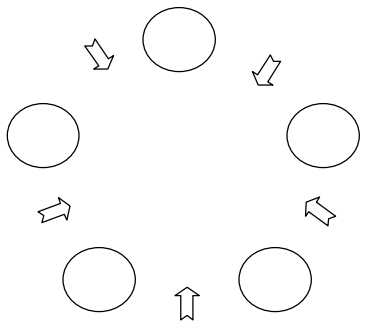
The Dining Philosophers



What is a possible solution?

Solution 1


```
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);
```

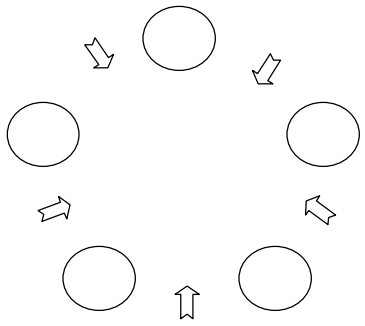


Deadlock

Solution 2


```
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);
```

Even man takes left, right - odd man takes right,left



No possibility of deadlock

Nesting Semaphore Operations

p_r P Operation Order

```
P(mutex1);
  P(mutex2);
    <access R1>;
    <access R2>;
  V(mutex2);
V(mutex1);
```

p_s P Operation Order

```
P(mutex2);
  P(mutex1);
    <access R1>;
    <access R2>;
  V(mutex1);
V(mutex2);
```

Warning: possible deadlock

This is where the abstract software solutions come in…

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
- $P_{simultaneous}(S_1, \ldots, S_n)$

Simultaneous Semaphores Def


```
P_sim(semaphore S, int N) {
L1: if ((S[0]>=1)&& … &&(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;
    }
}
```

```
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 Psimultaneous);
  } else {
}
```

Simultaneous Semaphore

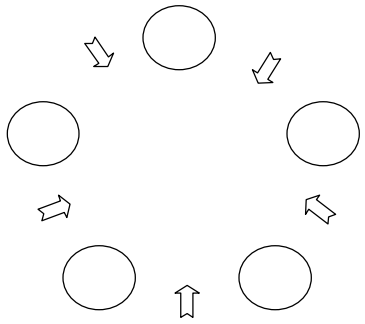

```
int R_num = 0, S_num = 0;
Queue R_wait, S_wait;
Semaphore mutex = 1;

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);
}
```

Slide 9-15

Dining Philosophers Problem

```
philosopher(int i) {
  while(TRUE) {
    // Think
    // Eat
    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);
```

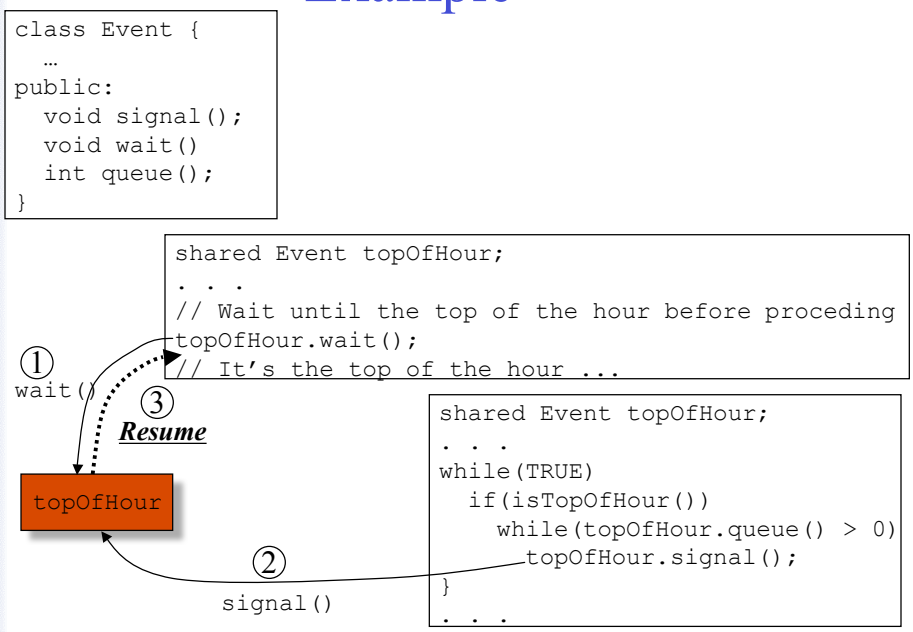


Slide 9-16

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

Example



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

Slide 9-21

Signal Handling

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

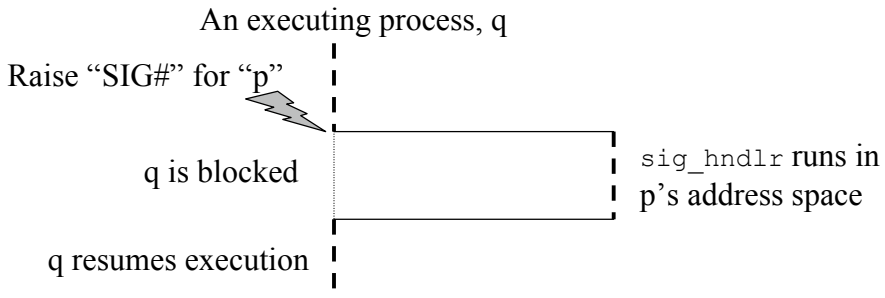
```
void myHndlr(...) {
/* ARBITRARY CODE */
}
```

Signal Handling

Slide 9-22

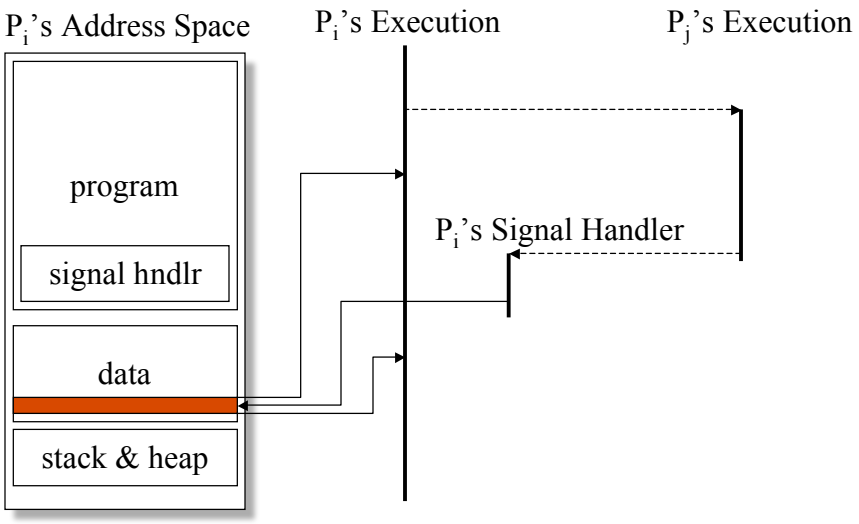
```
/* code for process p */
. . .
signal(SIG#, sig_hndlr);
. . .
/* ARBITRARY CODE */
```

```
void sig_hndlr(...) {
/* ARBITRARY CODE */
}
```



Using UNIX Signals

Slide 9-23



Toy Signal Handler

Slide 9-24

```
#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 (2)


```
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
}
```

Monitors

- Specialized form of ADT
  - Encapsulates implementation
  - Public interface (types & functions)
- Only one process can be executing in the ADT at a time

```
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(){                writer(){
  while(TRUE) {            while(TRUE) {
    . . .                    . . .
    startRead();             startWriter();
    finishRead();            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

- **Deadlock can happen**

```
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;
  };
};
```

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:

```
. . .
if(resourceNotAvailable()) resourceCondition.wait();
/* now available … continue … */
. . .
```

- Mesa semantics:

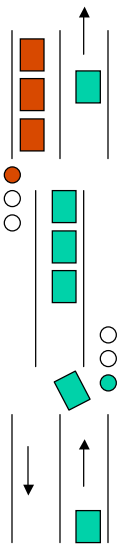
```
. . .
while(resourceNotAvailable()) resourceCondition.wait();
/* now available … continue … */
. . .
```

2nd Try at Readers & Writers


```
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



Example: Synchronizing Traffic


```
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 ...

```
#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() { // Initilization
    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);
  };
}
```

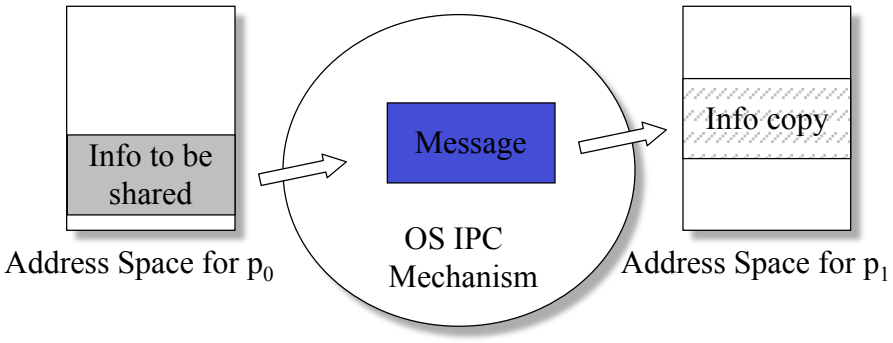
Experience with Monitors

- 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.

Interprocess Communication (IPC)

- Different processes - no shared memory space
- IPC - a way for different processes to communicate even if they exist on different machines
- OS copies info from sending process' memory space to receiving process' memory space

IPC Mechanisms

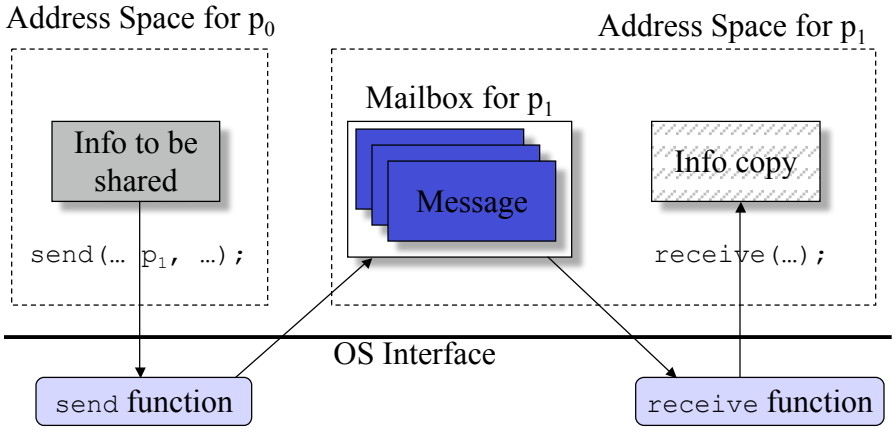


- Must bypass memory protection mechanism for local copies
- Must be able to use a network for remote copies

Refined IPC Mechanism

Slide 9-43

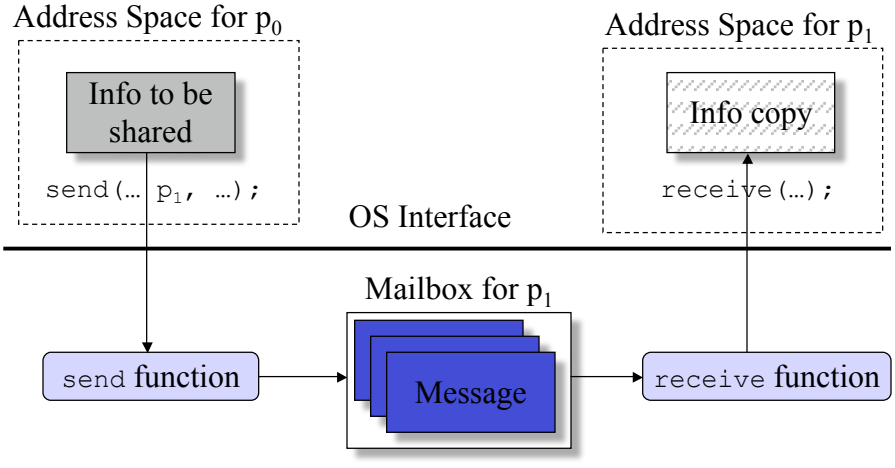
- Spontaneous changes to p_1's address space
- Avoid through the use of mailboxes



Slide 9-44

Refined IPC Mechanism

- OS manages the mailbox space
- More secure message system



Slide 9-45

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

Slide 9-46

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

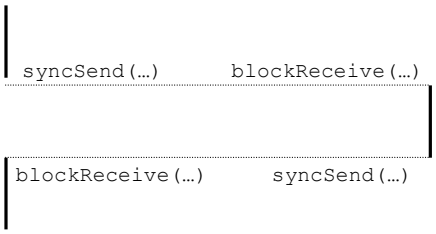
Synchronized IPC

Code for p_1

```
/* signal p2 */
syncSend(message1, p2);
<waiting …>;
/* wait for signal from p2 */
blockReceive(msgBuff, &from);
```

Code for p_2

```
/* wait for signal from p1 */
blockReceive(msgBuff, &from);
<process message>;
/* signal p1 */
syncSend(message2, p1);
```



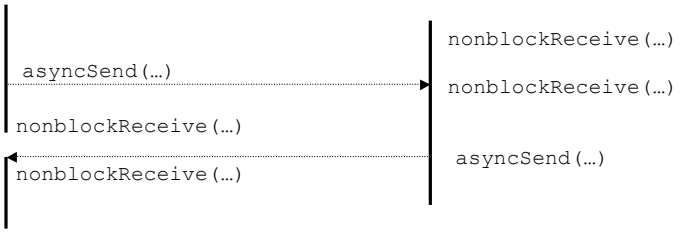
Asynchronous IPC

Code for p_1

```
/* signal p2 */
asyncSend(message1, p2);
<other processing>;
/* wait for signal from p2 */
while(!nbReceive(&msg, &from));
```

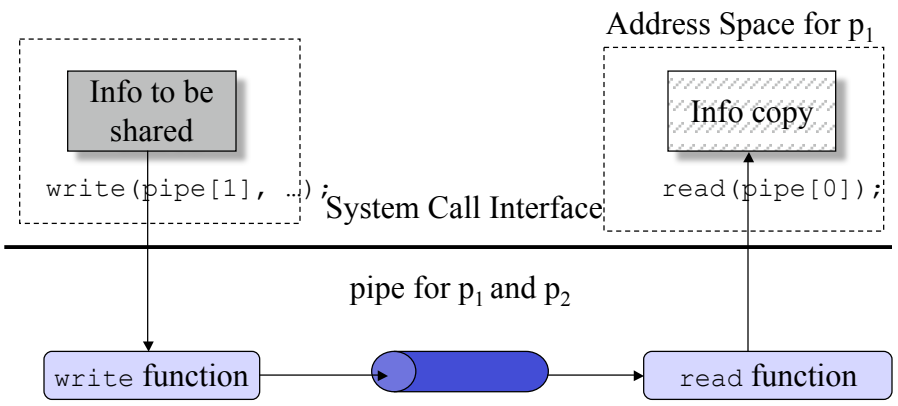
Code for p_2

```
/* test for signal from p1 */
if(nbReceive(&msg, &from)) {
  <process message>;
  asyncSend(message2, p1);
}else<
  <other processing>;
}
```



UNIX Pipes

Slide 9-51



UNIX Pipes (cont)

Slide 9-52

- 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

Slide 9-53

UNIX Pipes (cont)

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

```
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);
  . . .
}
```

Slide 9-54

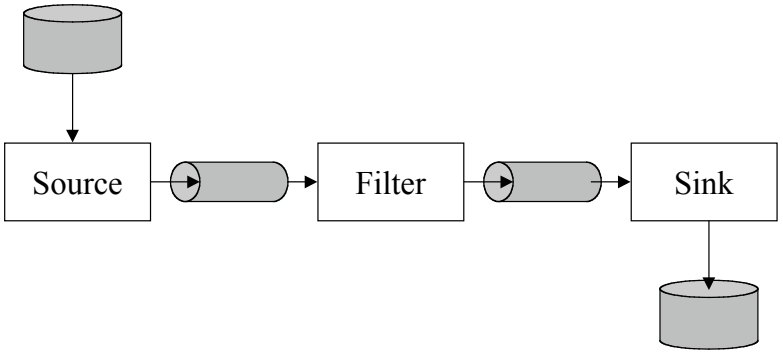
UNIX Pipes (cont)

- 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);
ioctl(pipeID[0], FIONBIO, &on);
. . .
read(pipeID[0], buffer, len);
if(errno != EWOULDBLOCK) {
  /* no data */
} else { /* have data */
```

Source, Filter and Sink Processes

Slide 9-55



Information Flow Through UNIX Pipes

Slide 9-56