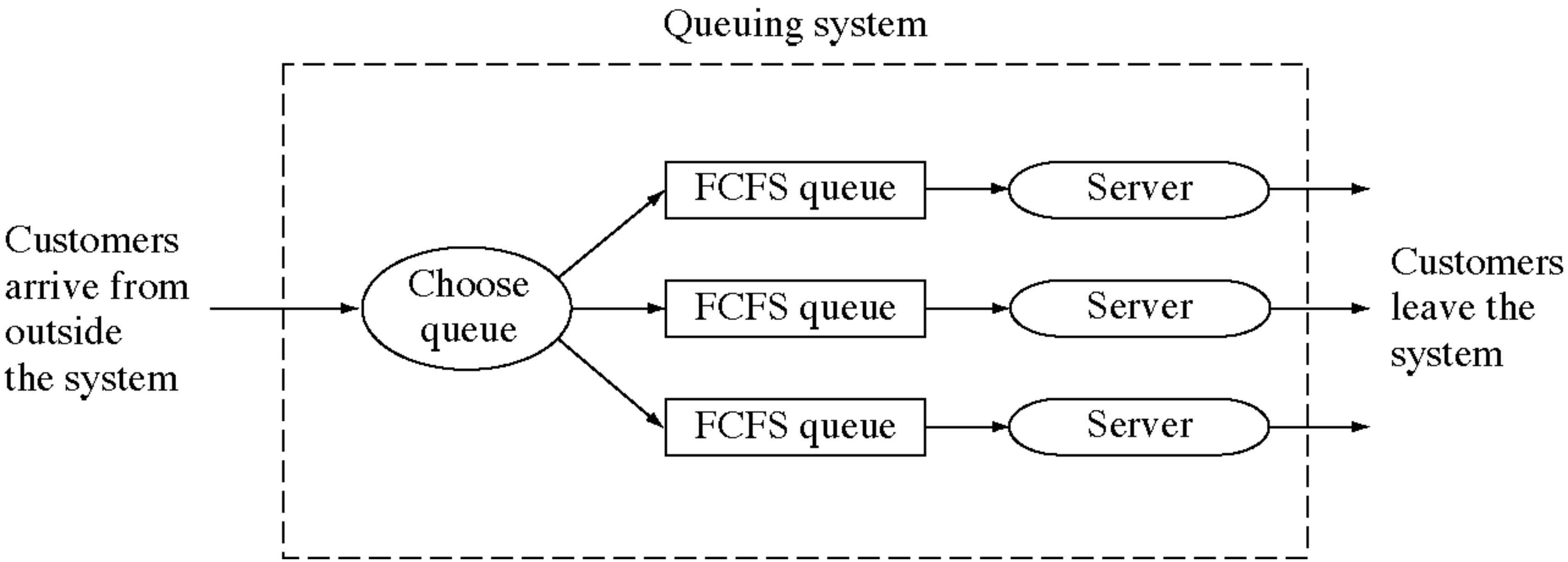
Processes

Chapter 8

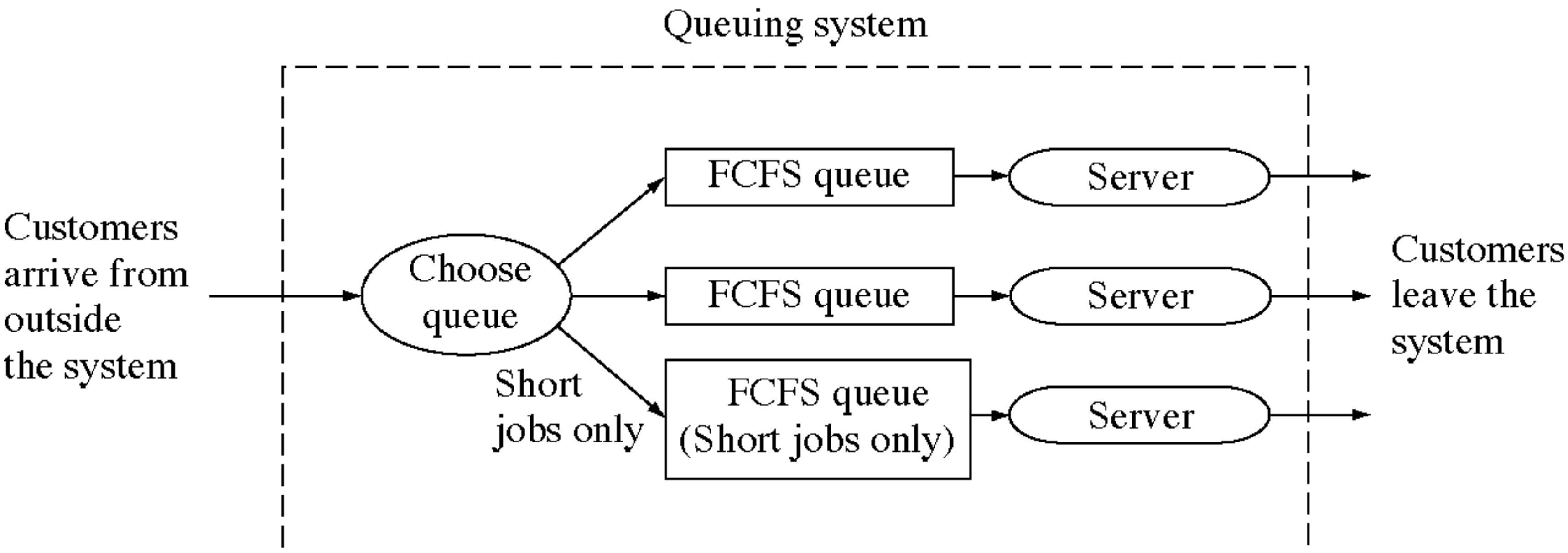
Key concepts in chapter 8

- Non-preemptive scheduling: first-come-first-served (or first-in-first-out), shortest-job-first, priority
- Preemptive scheduling: round-robin, multiple round-robin queues
- Policy versus mechanism
- Deadlock and starvation
- Remote procedure calls
- Synchronization
- Semaphores
- Monitors

FCFS at the supermarket



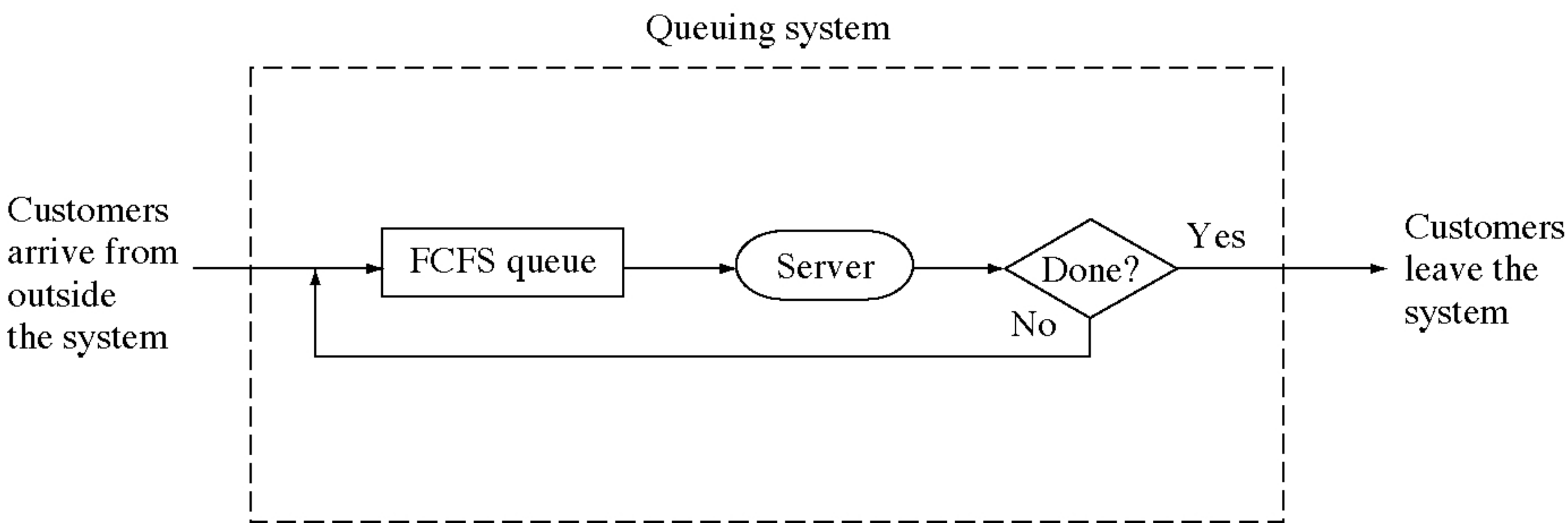
SJF at the supermarket



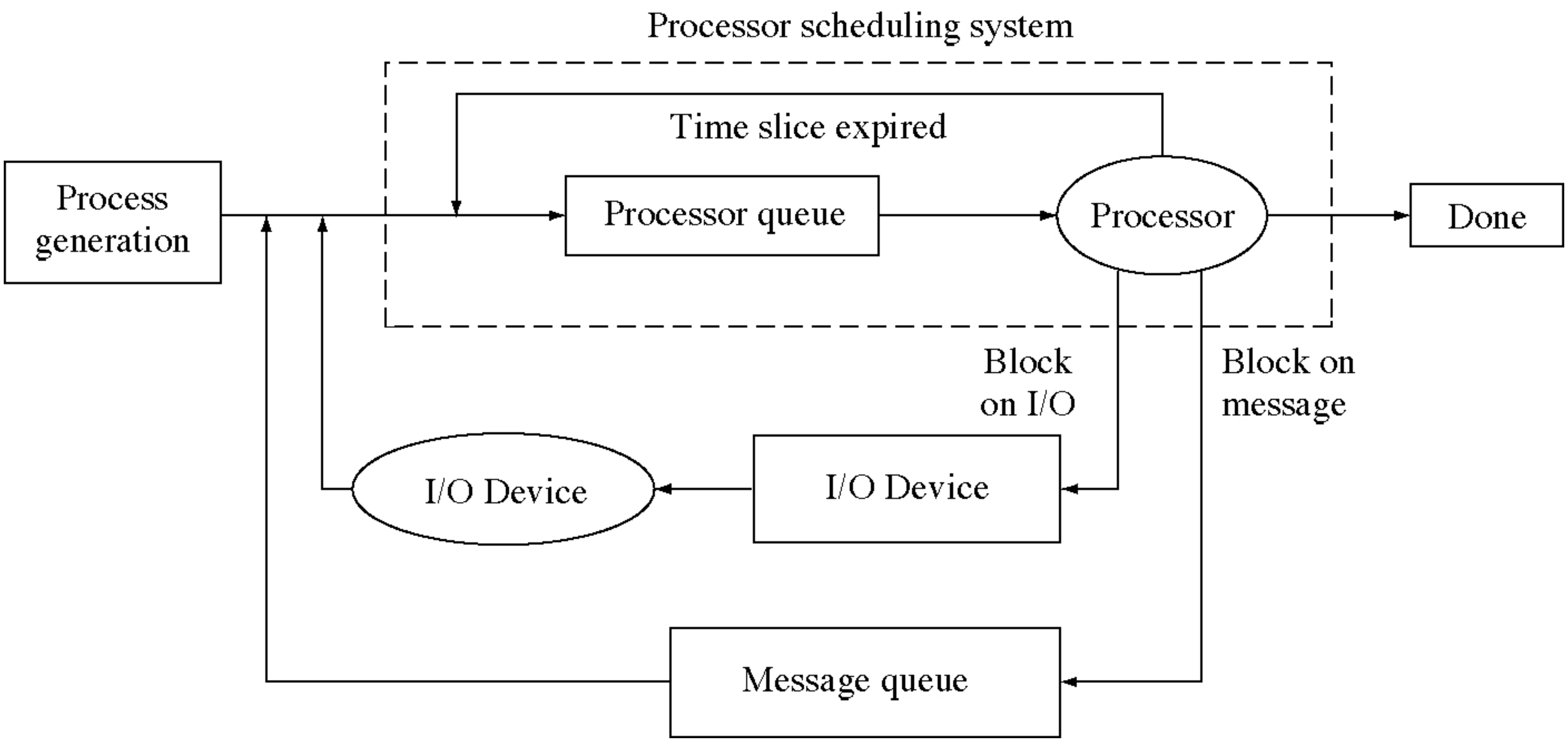
Everyday scheduling methods

- First-come, first served
- Shorter jobs first
- Higher priority jobs first
- Job with the closest deadline first
- Round-robin

Round-robin scheduling



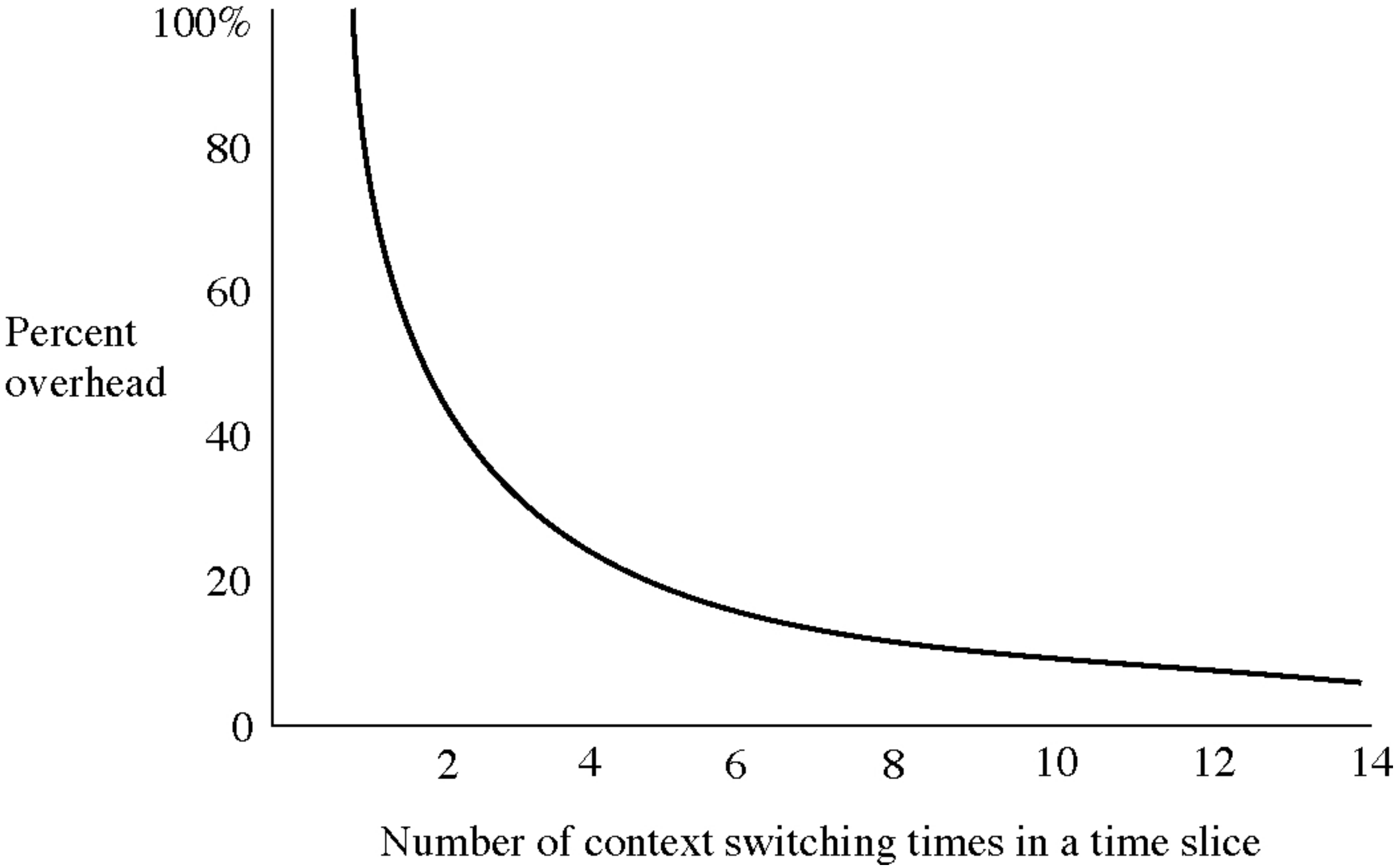
Process flow in an OS



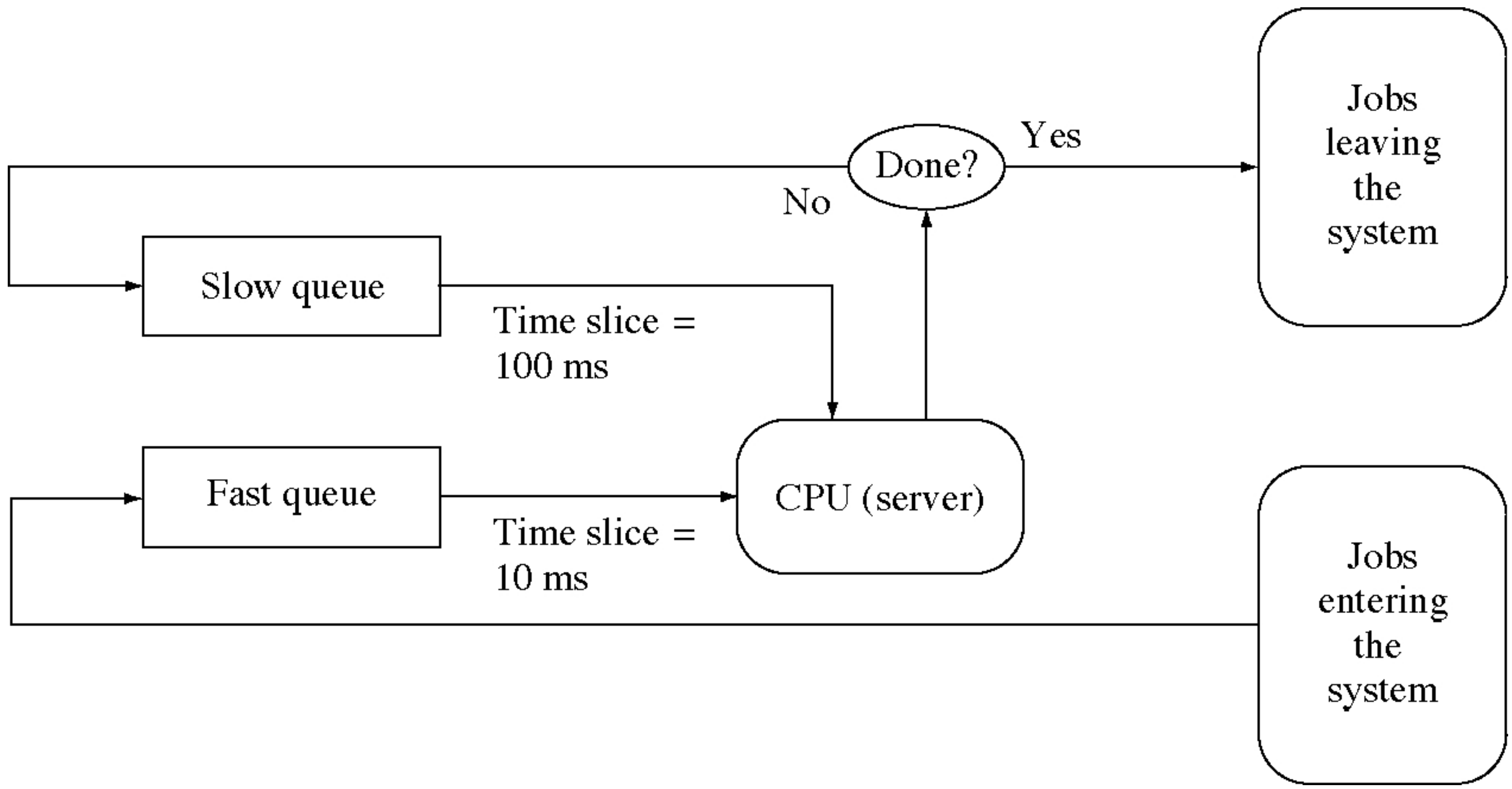
Round-robin scheduling

- Preemptive: processes lose the processor before they are finished using it
- Time-slice or quantum: time of each turn
  - 5 to 100 milliseconds
  - could be adaptive and change with the load
- Performs poorly if the load is heavy

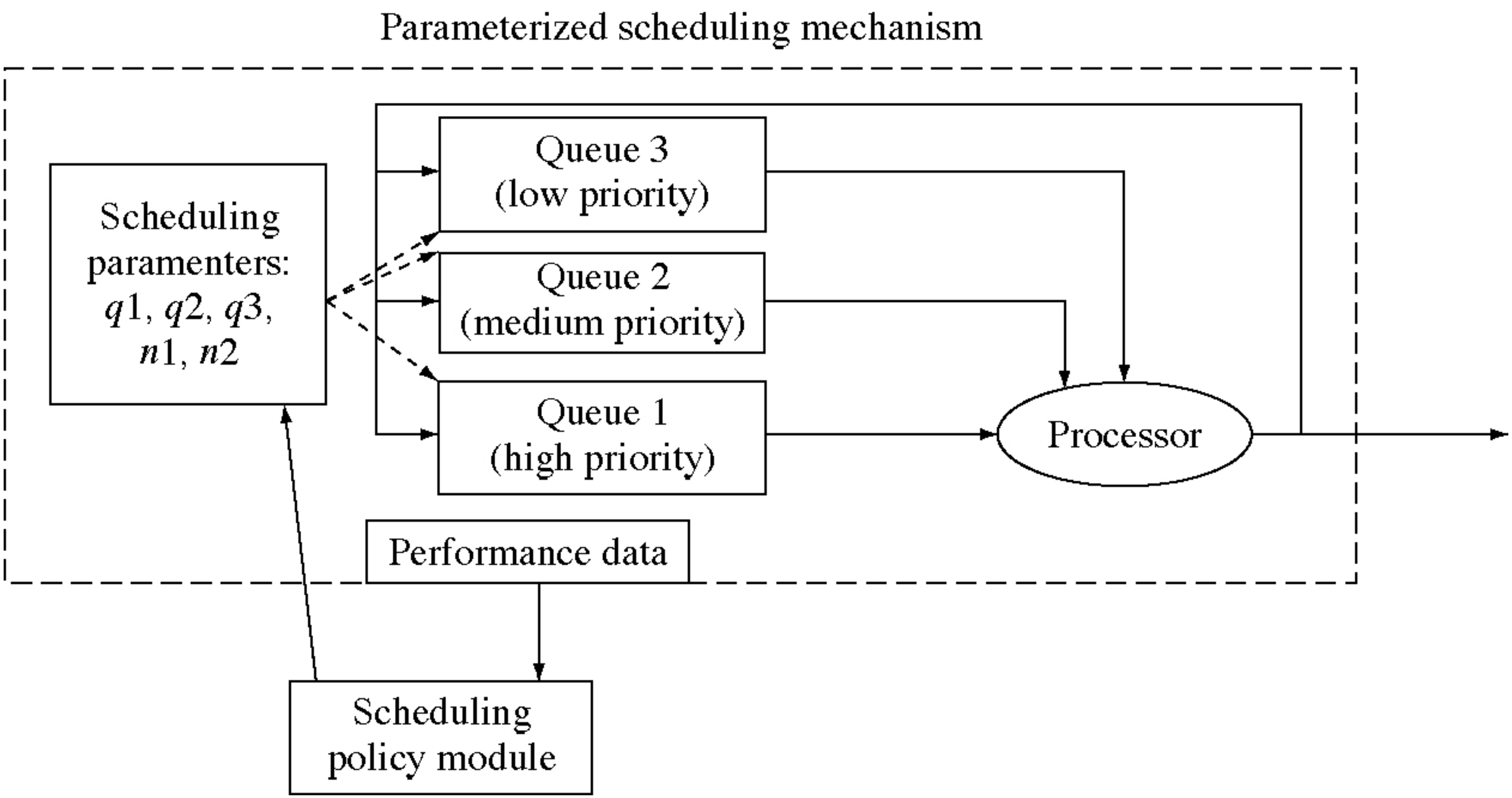
Round-robin time slice



Two-queue scheduling



Three-queue scheduling



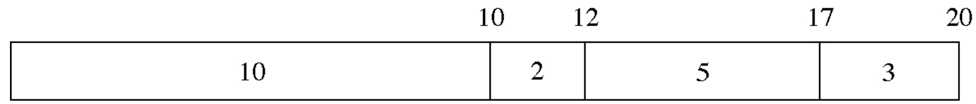
Policy and mechanism in scheduling

- We can parameterize the three-queue scheduler
  - and make it into a range of scheduling algorithms
- Scheduling policy shifts through the day
  - so schedulers should have settable parameters
  - the scheduler is the mechanism

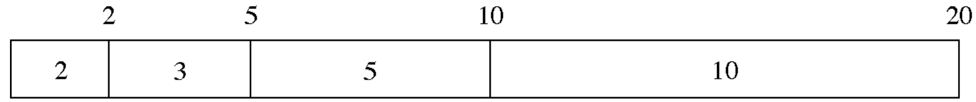
A scheduling example

- Job 1, 10 seconds, priority 3
- Job 2, 2 seconds, priority 2
- Job 3, 5 seconds, priority 1
- Job 4, 3 seconds, priority 4

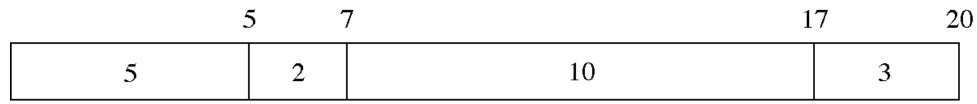
A scheduling example



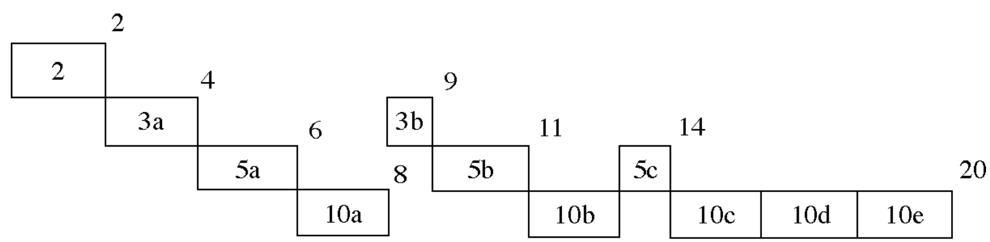
First-come, first-served: 10 + 12 + 17 + 20 = 59/4 = 14.75



Shortest-job-first: 2 + 5 + 10 + 20 = 37/4 = 9.25



Priority: 5 + 7 + 17 + 20 = 49/4 = 12.25



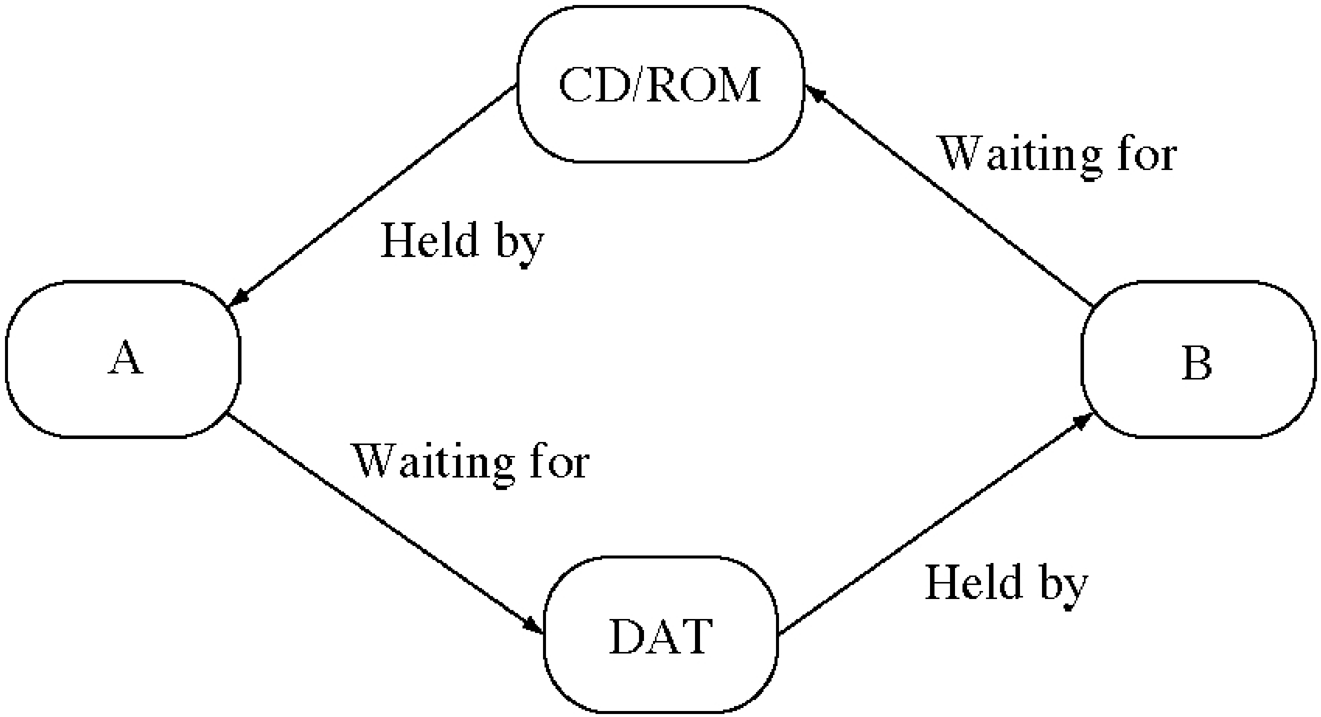
Round robin (time slice = 2): 2 + 9 + 14 + 20 = 45/4 = 11.25

Round robin (continuous): 8 + 11 + 15 + 20 = 54/4 = 13.50

Scheduling in real OSs

- All use a multiple-queue system
- UNIX SVR4: 160 levels, three classes
  - time-sharing, system, real-time
- Solaris: 170 levels, four classes
  - time-sharing, system, real-time, interupt
- OS/2 2.0: 128 level, four classes
  - background, normal, server, real-time
- Windows NT 3.51: 32 levels, two classes
  - regular and real-time
- Mach uses “hand-off scheduling”

Two-process deadlock



Deadlock

- When two or more processes are waiting for each other to finish using resources
- This causes the system to come to a halt
- Deadlock is a problem if you don’t take steps to prevent or detect it.
- Message deadlock:
  - ProcessA { receive(B, msg); send(B, msg): }
  - ProcessB { receive(A, msg); send(A, msg): }
  - Could result from a message being lost

Conditions for deadlock

- Resources cannot be preempted.
- Resources cannot be shared.
- A process is holding one resource and requesting another.
- Circular waits are possible.

Dealing with deadlock

- Three solutions
  - *Prevention*: place restrictions on resource request to make deadlock impossible
  - *Avoidance*: plan ahead to avoid deadlock.
  - *Recovery*: detect when deadlock occurs and recover from it

Deadlock prevention

- Allow preemption
  – this is not easy
- Allow sharing
  – sometimes you can create multiple, virtual copies of the resource
- Avoid hold-and-wait
  – require all resources to be acquired in one request
- Avoid circular wait
  – require resource to be acquired in a fixed order

Deadlock avoidance

- There are algorithms for this
  - e.g. the Banker’s algorithm
- But the have problems
  - they are slow
  - they do not account for resource failure or loss

Deadlock recovery

- The usual solution
- There are algorithms to detect deadlock
  - but they are slow
  - timeouts are another solution

Two-phase locking

- Each process has two stages
  - Stage 1
    - acquire all resources you will need
    - if a resource is locked, then release all resources you are holding and start over
  - Stage 2
    - Use the resources
- Deadlock is prevented (no hold-and-wait)

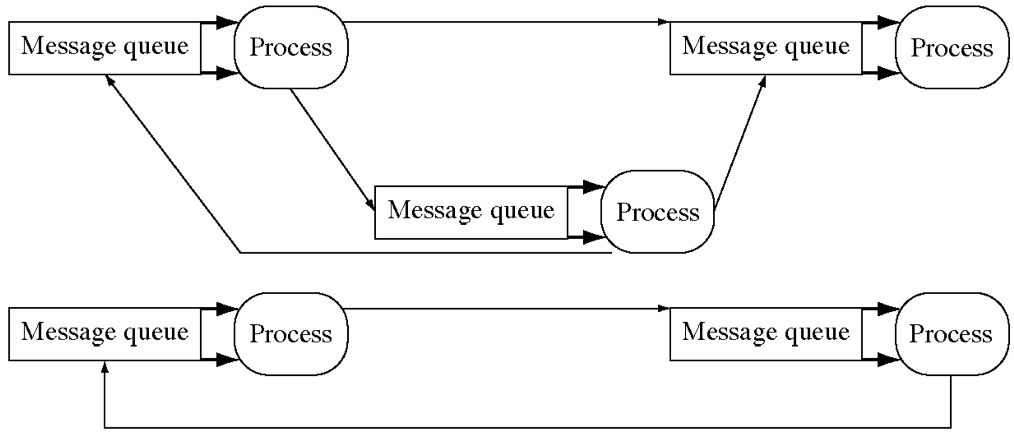
Starvation

- When a process is unlucky and cannot get a resource it needs
  - for example, in a random queue
  - calling in to a number that is usually busy
- Most common when you are requesting two resources together

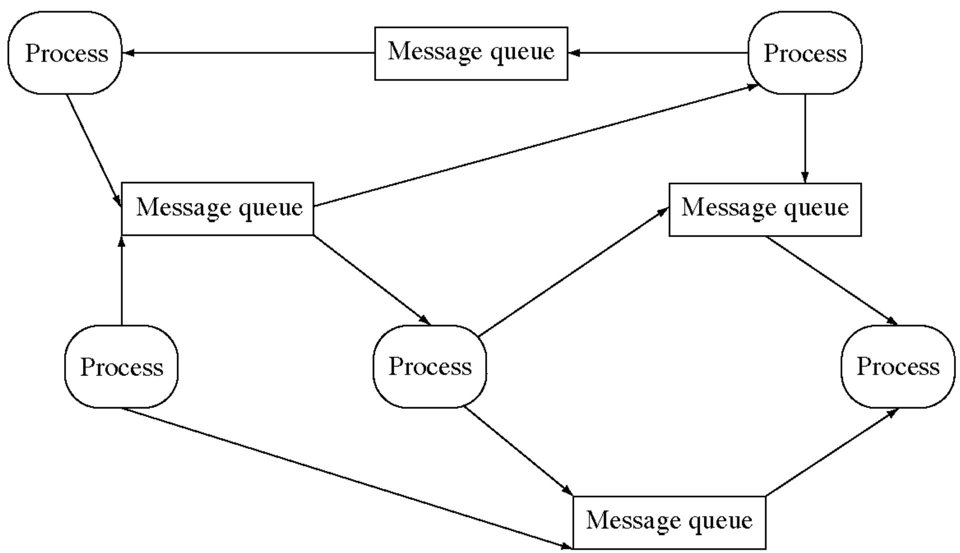
Variations in message-passing

- Using PIDs as message addresses
  - less flexible than indirection through queues
- Non-blocking receives
  - if there is no message, receive returns an error
  - more flexible but require busy waiting for a message
- Blocking sends
  - send, receive, reply model
  - less flexible but avoids OS message buffering

Addressing messages to processes



(a) Messages directly to processes

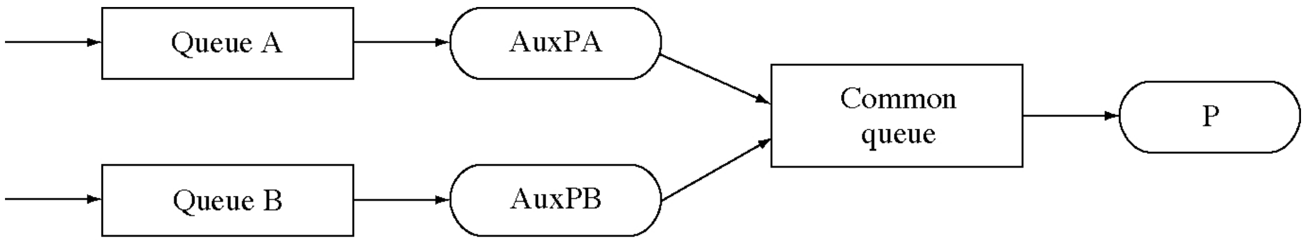


(b) Message-to-message queues

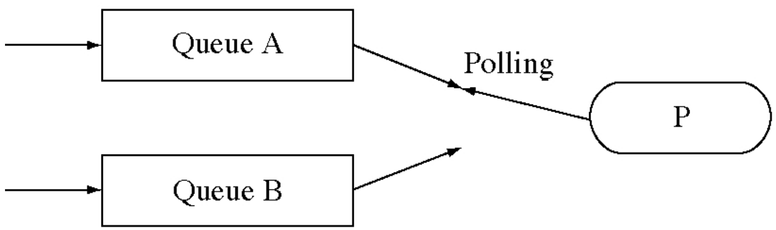
Design technique: Indirection

- Indirection: going through an intermediate object to get to an object
- Examples
  - sending a message to a queue then another processes receives it from the queue
  - access a private variable in an object using set and get functions
- Advantage: you get control just before the object is accessed

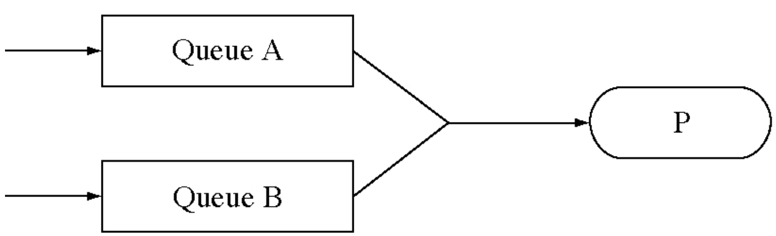
Monitoring two message queues simultaneously



(a) Use extra processes to create a common queue



(b) Poll both queues

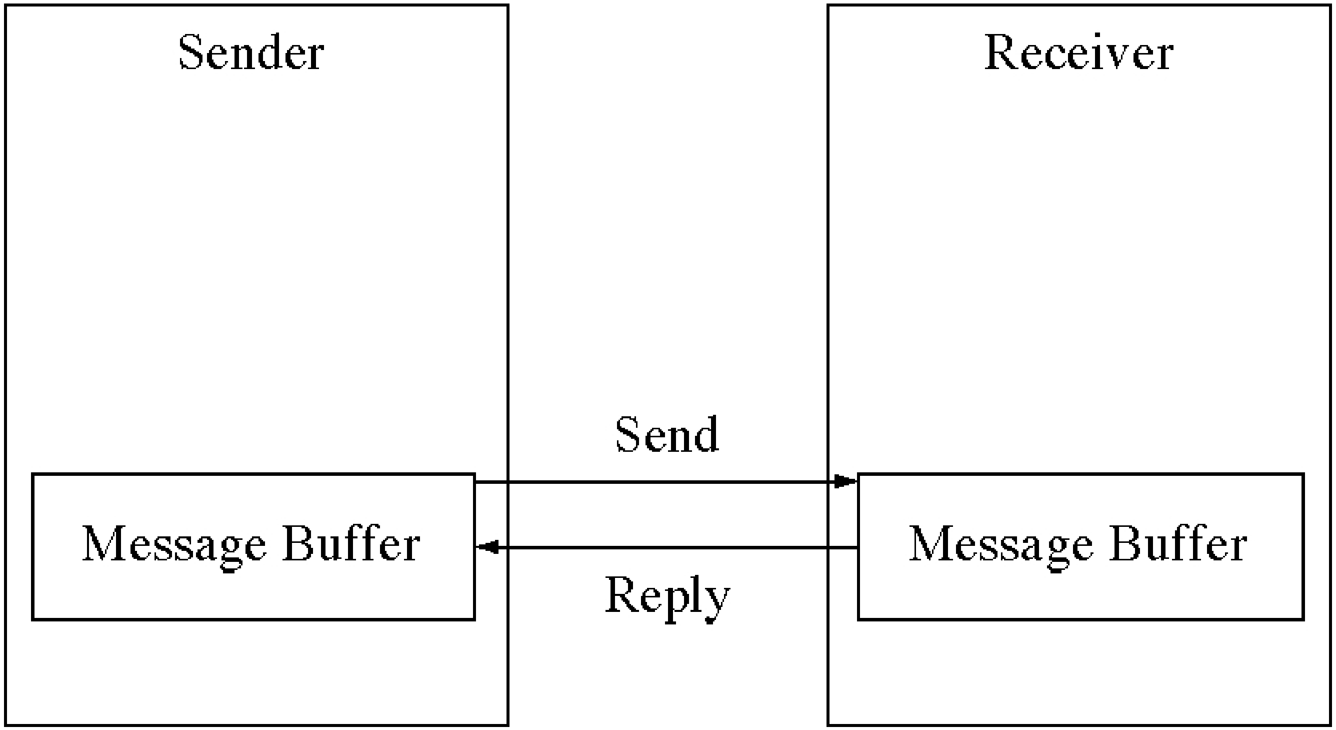


(c) Read from two queues simultaneously.

Design technique: Adding a new facility

- Three ways to do it
    - Example: receiving from two messages queues
  - 1. use an existing facility
    - create threads to listen on each queue
  - 2. create a new low-level primitive and use it
    - add a non-blocking receive and poll with it
  - 3. create a new high-level primitive to do the job
    - add a system call to receive from two queues

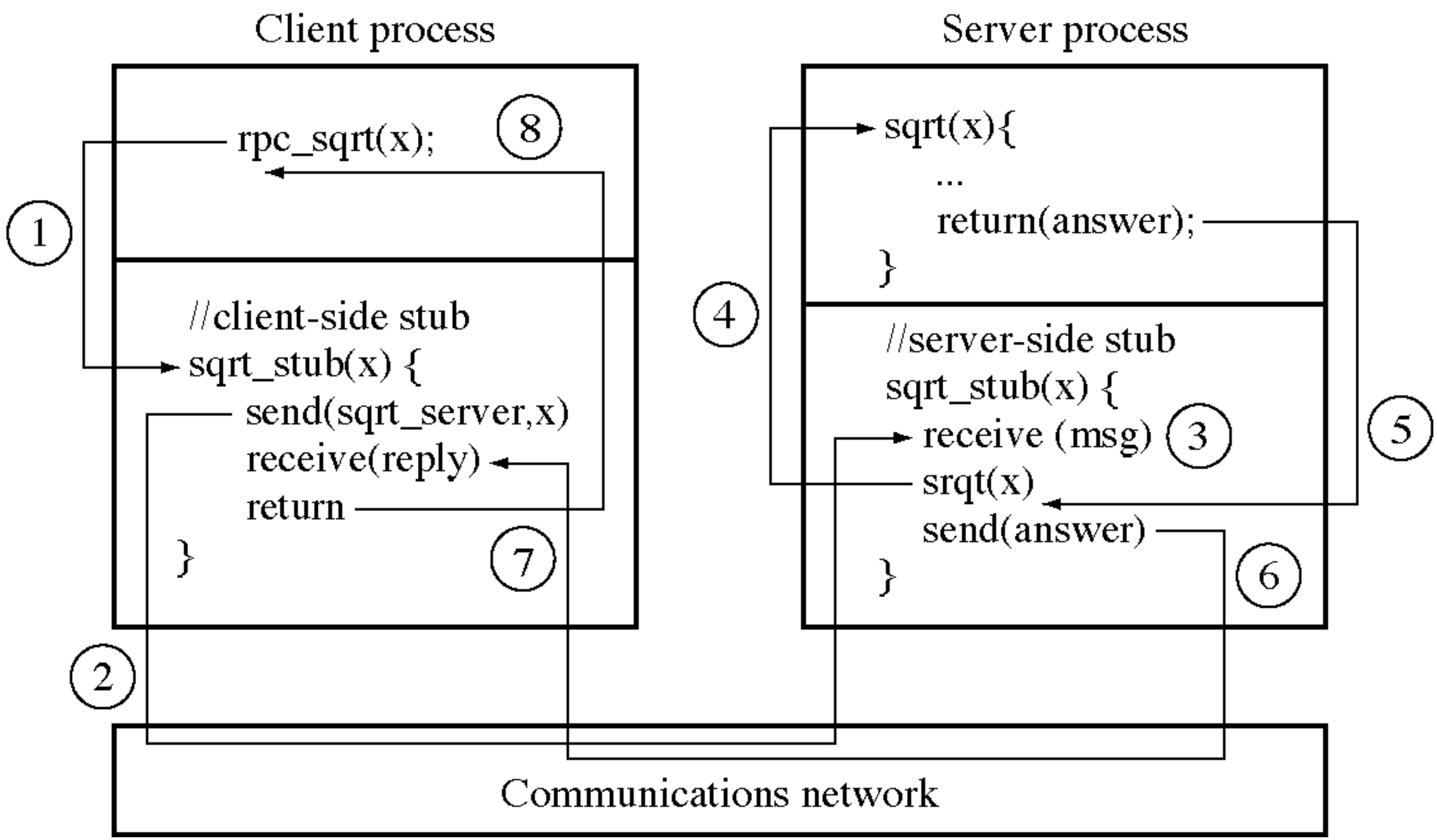
Send-receive-reply model



Remote procedure calls (RPCs)

- Remote procedure call (or RPC)
  - another IPC technique
  - RPC uses the familiar procedure call model
- Calling a procedure in another address space looks the same as calling a local procedure
- Implemented with stubs

Remote procedure call flow of control



Implementing RPCs: Client-side stub

- ```
  retValue = RemoteService( arg1, arg2, arg3 );
  // client-side "stub" procedure
  int RemoteService( int arg1, int arg2, int arg3 ) {
    MessageBuffer msg;
    msg[0] = RemoteServiceCode;
    msg[1] = arg1;
    msg[2] = arg2;
    msg[3] = arg3;
    SendMessage( RemoteServer, msg );
    ReceiveMessage( RemoteServer, msg );
    return msg[1];
  }
  ```
  ```

RPC implementation issues

- An RPC is not really procedure call
  - arguments must be by value (no pointers)
  - server must be found
  - server can fail
  - messages can be lost
- RPC libraries are provided to allow reuse of most of the RPC mechanism

Synchronization

- *Synchronization* is processes (or threads) waiting for each other, basically signaling
- In the OS we used various methods: disabling interrupts, spin locks, kernel-mode processes
- In user processes we have seen only messages, which combine data transfer with synchronization, but they can be separated

Semaphores

- Basic synchronization problems:
  - mutual exclusion
  - signaling
- Semaphores are a shared-memory synchronization primitive that solves these two problems

Semaphore operations

- int AttachSemaphore(int global_id)
  - a global, numerical ID (global_id)
  - it returns a local identifier for the semaphore
- int Wait(int local_id)
  - local_id from the AttachSemaphore
- int Signal(int local_id)
- int DetachSemaphore(int local_id)
  - local_id from the AttachSemphore

Binary semaphore implementation

- ```
  void Wait( int sema_id ) {
    Semaphore * sema = GetSemaphoreFromId( sema_id );
    Lock( sema->lock );
    if( !sema->busy ) {
      sema->busy = True;
    } else {
      sema->queue->Insert( current_process );
      Block( current_process );
    }
    Unlock( sema->lock );
  }
  void Signal( int sema_id ) {
    Semaphore * sema = GetSemaphoreFromId( sema_id );
    Lock( sema->lock );
    if( sema->queue->Empty() ) {
      sema->busy = False;
    } else {
      process = sema->queue->Remove();
      Unblock( process );
    }
    Unlock( sema->lock );
  }
  ```
  ```

Two-process mutual exclusion

```
// This is the code for process A or B
void main( int argc, char * argv[ ] ) {
  int mutex_sema = AttachSemaphore( FMutexID );
  if( ThisIsProcessA() )
    Signal( mutex_sema );
  // Initialize the semaphore to not busy
  while( 1 ) {
    DoOtherThings();
    // The critical section
    Wait( mutex_sema );
    UseFileF();
    Signal( mutex_sema );
  }
   DetachSemaphore( mutex_sema );
}
```

Two-process rendezvous

- ```
  void main(int argc,char *argv[]){ // Game player A
    int a_sema = AttachSemaphore( GPAID );
    int b_sema = AttachSemaphore( GPBID );
    // Tell B that A is ready to go.
    Signal( b_sema );
    // Wait until B is ready to go.
    Wait( a_sema );
    DetachSemaphore( a_sema );
    DetachSemaphore( b_sema );
  }
  void main(int argc,char *argv[]){ // Game player B
    int b_sema = AttachSemaphore( GPBID );
    int a_sema = AttachSemaphore( GPAID );
    // Tell A that B is ready to go.
    Signal( a_sema );
    // Wait until A is ready to go.
    Wait( b_sema );
    DetachSemaphore( a_sema );
    DetachSemaphore( b_sema );
  }
  ```
  ```

Producer-consumer

- ```
  Buffer buffer;
  void main() { // The Producer
    int empty_buffer = AttachSemaphore(EmptyBufferID);
    int full_buffer = AttachSemaphore(FullBufferID);
    while( 1 ) {
      Wait( empty_buffer );
      FillBuffer( buffer );
      Signal( full_buffer );
    }
    DetachSemaphore(empty_buffer);
    DetachSemaphore(full_buffer);
  }
  void main() { // The Consumer
    int empty_buffer = AttachSemaphore(EmptyBufferID);
    int full_buffer = AttachSemaphore(FullBufferID);
    Signal( empty_buffer );
    while( 1 ) {
      Wait( full_buffer );
      ConsumeBuffer( buffer );
      Signal( empty_buffer );
    }
    DetachSemaphore( empty_buffer );
    DetachSemaphore( full_buffer );
  }
  ```
  ```

Counting semaphore implementation

- ```
  void Wait( int sema_id ) {
    Semaphore * sema = GetSemaphoreFromId( sema_id );
    Lock( sema->lock );
    if( sema->count > 0 ) --sema->count;
    else {
      sema->queue->Insert( current_process );
      Block( current_process );
    }
    Unlock( sema->lock );
  }
  void Signal( int sema_id ) {
    Semaphore * sema = GetSemaphoreFromId( sema_id );
    Lock( sema->lock );
    if( sema->queue->Empty() ) ++sema->busy;
    else {
      process = sema->queue->Remove();
      Unblock( process );
    }
    Unlock( sema->lock );
  }
  ```
  ```

N-buffer producer

- ```
  Const int MaxBuffers = 20;
  Queue<Buffer> buffer_queue;
  void main() { // The Producer
    int use_buffer_queue
      = AttachSemaphore(UseBufferQueueID);
    int empty_buffer = AttachSemaphore(EmptyBufferID);
    int full_buffer = AttachSemaphore(FullBufferID);
    Buffer buffer;
    while( 1 ) {
      FillBuffer( buffer );
      Wait( empty_buffer );
      Wait( use_buffer_queue );
      buffer_queue.Insert( buffer );
      Signal( use_buffer_queue );
      Signal( full_buffer );
    }
    DetachSemaphore( use_buffer_queue );
    DetachSemaphore( empty_buffer );
    DetachSemaphore( full_buffer );
  }
  ```
  ```

N-buffer consumer

- ```
  void main() { // The Consumer
    int use_buffer_queue
      = AttachSemaphore( UseBufferQueueID );
    int empty_buffer = AttachSemaphore(EmptyBufferID);
    int full_buffer = AttachSemaphore(FullBufferID);
    Buffer buffer;
    Signal( use_buffer_queue ); // Buffer queue free
    for( int i = 0; i < MaxBuffers; ++i )
      Signal( empty_buffer );
    while( 1 ) {
      Wait( full_buffer );
      Wait( use_buffer_queue );
      buffer = buffer_queue.Remove();
      Signal( use_buffer_queue );
      Signal( empty_buffer );
      ConsumeBuffer( buffer );
    }
    DetachSemaphore( use_buffer_queue );
    DetachSemaphore( empty_buffer );
    DetachSemaphore( full_buffer );
  }
  ```
  ```

Semaphores and messages

- Semaphores
  - are basically messages with no content
  - they handle synchronization only
  - data are transferred in shared memory
- Messages
  - are more appropriate when there is no shared memory

System constants and globals

- ```
  // system limits (we can change these)
  const int NumberOfSemaphores = 50;

  // system call numbers
  const int AttachSemaphoreSystemCall = 8;
  const int DetachSemaphoreSystemCall = 9;
  const int SignalSemaphoreSystemCall = 10;
  const int WaitSemaphoreSystemCall = 11;

  // semaphore data structures
  struct Semaphore {
    int allocated;
    int count;
    int use_count;
    int id;
    Queue<Pid> queue;
  };

  Semaphore sema[NumberOfSemaphores];
  ```
  ```

System initialization

- ```
  int main( void ) {
    // ... all what we had before, plus
    // initialize all semaphores to "not allocated”
    for( i = 0; i < NumberOfSemaphores; ++i ) {
      sema.allocated[i] = False;
    }
  ```
  ```

System call handler (1 of 3)

```
case AttachSemaphoreSystemCall:
  int sema_id; asm { store r9,sema_id }
  pd[current_process].sa.reg[1]
    = AttachSemaphore( sema_id );
  break;
case DetachSemaphoreSystemCall:
  int sid; asm { store r9,sid }
  pd[current_process].sa.reg[1]
    = DetachSemaphore( sid );
  break;
case SignalSemaphoreSystemCall:
  int sid; asm { store r9,sid }
  pd[current_process].sa.reg[1]
    = SignalSemaphore( sid );
  break;
case WaitSemaphoreSystemCall:
  int sid; asm { store r9,sid }
  pd[current_process].sa.reg[1]
    = WaitSemaphore( sid );
  break;
```

System call handler (2 of 3)

- ```
  case SendMessageSystemCall:
    // get the arguments
    int * user_msg; asm { store r9,user_msg }
    int to_q; asm { store r10,to_q }
    // check for an invalid queue identifier
    if( !message_queue_allocated[to_q] ) {
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    int msg_no = GetMessageBuffer();
    // make sure we have not run out of message
  buffers
    if( msg_no == EndOfFreeList ) {
      pd[current_process].sa.reg[1] = -2;
      break;
    }
    // copy the message vector from the system
  caller's
    // memory into the system's message buffer
    CopyToSystemSpace( current_process, user_msg,
      message_buffer[msg_no], MessageSize );
  ```
  ```

System call handler (3 of 3)

- ```
    // put it on the queue
    message_queue[to_q].Insert( msg_no );
    // notify any waiters that it is there
    SignalSemaphore( message_semaphore[to_q] ); // NEW
    pd[current_process].sa.reg[1] = 0;
    break;
  case ReceiveMessageSystemCall:
    int * user_msg; asm { store r9,user_msg }
    int from_q; asm { store r10,from_q }
    // check for an invalid queue identifier
    if( !message_queue_allocated[from_q] ) {
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    WaitSemaphore( message_semaphore[from_q] ); // NEW
    int msg_no = message_queue[from_q].Remove();
    TransferMessage( msg_no, user_msg );
    pd[current_process].sa.reg[1] = 0;
    break;
  ```
  ```

Attach semaphore

- ```
  int AttachSemaphore( int sema_id ) {
    int i, free_slot = -1;
    for( i = 0; i < NumberOfSemaphores; ++i )
      if( sema[i].allocated ) {
        if( sema[i].id == sema_id ) break;
        else free_slot = I
      }
    if( i >= NumberOfSemaphores ) { // found sema_id?
      if( free_slot == -1 ) // found free slot?
        return -1; // No, so return an error code.
      i = free_slot;
      sema[i].allocated = True;
      sema[i].count = 0;
      sema[i].use_count = 0;
      sema[i].id = sema_id;
      sema[i].queue = new Queue<Pid>;
    }
    ++sema[i].use_count;
  return i;
  }
  ```
  ```

Detach semaphore

- ```
  int DetachSemaphore( int sid ) {
    if( !sema[sid].allocated ) {
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    if( --sema[sid].use_count == 0 ) {
      sema[sid].allocated = False;
      delete sema[sid].queue;
    }
    return 0;
  }
  ```
  ```

Signal semaphore

- ```
  int SignalSemaphore( int sid ) {
    if( !sema[sid].allocated ) {
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    if( sema[sid].queue->Empty() )
      ++sema[sid].count;
    else {
      int pid = sema[sid].queue->Remove();
      pd[pid].state = Ready;
    }
    return 0;
  }
  ```
  ```

Wait semaphore

- ```
  int WaitSemaphore( int sid ) {
    if( !sema[sid].allocated ) {
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    if( sema[sid].count > 0 )
      --sema[sid].count;
    else {
      sema[sid].queue->Insert(current_process);
      pd[current_process].state = Blocked;
    }
    return 0;
  }
  ```
  ```

System initialization

- ```
  processTableSemaphore
    = AttachSemaphore( ProcessTableID );
  disk_queue = AttachSemaphore( DiskQueueID);
  disk_free = AttachSemaphore( DiskFreeID );
  ```
  ```

Dispatcher

- ```
  int SelectProcessToRun( void ) {
    static int next_proc = NumberOfProcesses;
    int i, return_value = -1;
    // Get exclusive access to the process table.
    WaitSemaphore(processTableSemaphore);     // NEW
    // ... use process table as before
    SignalSemaphore(processTableSemaphore);  // NEW
    return return_value;
  }
  ```
  ```

Disk I/O

- ```
  void DiskIO( int command, int disk_block,
      char * buffer ) {
    // Create a new disk request
    // and fill in the fields.
    DiskRequest * req = new DiskRequest;
    req->command = command;
    req->disk_block = disk_block;
    req->buffer = buffer;
    req->semaphore = pd[current_process].semaphore;
    // Then insert it on the queue.
    disk_queue.Insert( req );
    // Wake up the disk scheduler if it is idle.
    SignalSemaphore( disk_queue );
    WaitSemaphore( pd[current_process].semaphore );
  }
  ```
  ```

Disk scheduling

- ```
  void RealScheduleDisk( void ) {
    while( 1 ) { // NEW CODE
      WaitSemaphore( disk_free ); // NEW CODE
      WaitSemaphore( disk_queue ); // NEW CODE
      // Get the first disk request
      // from the disk request queue.
      DiskRequest * req = disk_queue.RemoveFirst();
      // remember which process is waiting
      // for the disk operation
      disk_completion_semaphore = req->semaphore;
      // issue the read or write,
      // with disk interrupt enabled
      if( req->command == DiskReadSystemCall )
        IssueDiskRead(req->disk_block,req->buffer,1);
      else
        IssueDiskWrite(req->disk_block,req->buffer,1);
    }
  }
  ```
  ```

Disk interrupt handler

- ```
  void DiskInterruptHandler( void ) {
    if( current_process > 0 ) {
      // was there a running process?
      // Save the processor state of the system
  caller.
      // ... as before
    }
    // Notify the waiting process that
    // the disk transfer is complete
    SignalSemaphore( disk_completion_semaphore );
    // Notify any waiters that the disk is free
    SignalSemaphore( disk_free );
    // now run a process
    Dispatcher();
  }
  ```
  ```

Monitors

- Another synchronization primitive
  - more structured, monitors are modules
- Components of a monitor module
  - variables: of any kind
  - condition variables: for monitor signaling
  - procedures: can be called from outside the monitor, they comprise the monitor’s interface

Counter monitor

- ```
  monitor Counter {
  private:
    int count = 0;
  public:
    void Increment( void ) { count = count + 1; }
    int GetCount( void ) { return count; }
  }
  int main() { // one process
    while( 1 ) {
      // ... Do other things
      Counter.Increment();
      // ... continue on
      int n = Counter.GetCount();
    }
  }
  int main() { // another process
    while( 1 ) {
      // ... Do other things
      Counter.Increment();
    }
  }
  ```
  ```

Signal monitor

```
monitor Signal {
private:
  int IsSignaled = 0;
  condition SendSignal;
public:
  void SendSignal( void ) {
    IsSignaled = 1;
    signal( SendSignal );
  }
  void WaitForSignal( void ) {
    if( !IsSignaled )
      wait( SendSignal );
    }
  }
}
```

Using the signal monitor

```
int main() { // the Signal Sender
  // ... Do things up to the signal point
  Signal.SendSignal();
  // ... continue on
}
int main() { // the Signal Receiver
  // ... Do things up to the signal point
  Signal.WaitForSignal();
  // ... continue on when the signal is receive
}
```

Bounded buffer monitor (1 of 2)

- ```
  monitor BoundedBufferType {
  private:
    BufferItem * buffer;
    int NumberOfBuffers;
    int next_in, nextout;
    int current_size;
    condition NotEmpty, NotFull;
  public:
    BoundedBufferType( int size ) {
      buffers = new BufferItem[size];
      NumberOfBuffers = size;
      next_in = 0; next_out = 0; current_size = 0;
    }
  ```
  ```

Bounded buffer monitor (2 of 2)

```
•     void Put( BufferItem item ) {
        if( current_size == NumberOfBuffers )
          wait( NotFull );
        buffer[next_in] = item;
        next_in = (next_in+1) % NumberOfBuffers;
        if( ++current_size == 1 )
          signal( NotEmpty );
      }
      BufferItem Get( void ) {
        if( current_size == 0 )
          wait( NotEmpty );
        BufferItem item = buffer[next_out];
        next_out = (next_out+1) % NumberOfBuffers;
        if( --current_size == NumberOfBuffers-1 )
          signal( NotFull );
        return item;
      }
    }
```

Using a bounded buffer monitor

- ```
  BoundedBufferType BoundedBuffer;

  int main() { // the Producer
    while( 1 ) {
      BufferItem item = ProduceItem();
      BoundedBuffer.Put( item );
    }
  }

  int main() { // the Consumer
    while( 1 ) {
      BufferItem item = BoundedBuffer.Get();
      ConsumeItem( item );
    }
  }
  ```
  ```

Counting semaphore monitor

- ```
  monitor Semaphore {
  private:
    int count = 0;
    condition NotBusy;
  public:
    void Signal( void ) {
      if( ++count > 0 )
        signal( NotBusy );
    }

    void Wait( void ) {
      while( count <= 0 )
        wait( NotBusy );
      --count;
    }
  }
  ```
  ```

Using a semaphore monitor

- ```
  int main() {       // one process
    while( 1 ) {
      // do other stuff
      // enter critical section
      Semaphore.Wait();
      // do critical section
      Semaphore.Signal();
    }
  }

  int main() {       // another process
    while( 1 ) {
      // do other stuff
      // enter critical section
      Semaphore.Wait();
      // do critical section
      Semaphore.Signal();
    }
  }
  ```
  ```

Protected counter in Ada95

- ```
  -- declare the interface to the task
  task Counter is
    entry GetCount( count : out integer );
    entry Increment;
  private
    count : integer;
  end Counter;
  -- the implementation of the protected variable
  task body Counter is
    loop
      select
        accept GetCount( count_out : out integer ) do
          count_out := count;
        end;
      or
        accept Increment do
          count := count + 1;
        end;
      end select;
    end loop;
  end Counter;
  ```
  ```

Using a protected counter

- ```
  task body OneProcess is begin
    loop
      -- do other things than incrementing the counter
      Counter.Increment;
      -- do other things
      Counter.GetCount( n );
    end loop;
  end OneProcess;

  task body AnotherProcess is begin
    loop
      -- do other things than incrementing the counter
      Counter.Increment;
      -- do other things
    end loop;
  end AnotherProcess;
  ```
  ```

Signal in Ada95

- ```
  task Signal is
    entry SendSignal;
    entry WaitForSignal;
  private
    IsSignaled : boolean := False;
  end Counter;

  task body Signal is
    accept SendSignal do
      IsSignaled := True;
    end;
    accept WaitForSignal do
      null;
    end;
  end Signal;
  ```
  ```

Using signal in Ada95

- ```
  task body SignalSender is begin
    -- get to point where event occurs
    Signal.SendSignal;
    -- go on to other things
  end SignalSender;

  task body SignalReceiver is begin
    -- get to the point where you need
    -- to wait for the event
    Signal.WaitForSignal;
    -- respond to event
  end SignalReceiver;
  ```
  ```

Bounded buffer in Ada95 (1 of 2)

- ```
  task BoundedBuffer is
    entry Put( x : in Item );
    entry Get( x : out Item );
  end BoundedBuffer;
  task body BoundedBuffer is
    NumberOfBuffers : constant integer := 20;
    buffers : array(1 .. NumberOfBuffers) of Item;
    current_size :
      integer range 0 .. NumberOfBuffers := 0;
    next_in, next_out
      : integer range 1 .. NumberOfBuffers := 1;
  ```
  ```

Bounded buffer in Ada95 (2 of 2)

```
• begin
    loop
      select
        when current_size < NumberOfBuffers =>
          accept Put( x : in item ) do
            buffers( next_in ) := x;
          end;
          next_in := (next_in mod NumberOfBuffers) + 1;
          current_size := current_size + 1;
        or when current_size > 0 =>
          accept Get( x : out Item do
            x := buffers(next_out);
          end;
          next_out := (next_out mod NumberOfBuffers)+1;
          current_size := current_size - 1;
        or
          terminate;
      end select;
    end loop;
  end BoundedBuffer;
```

Producer-consumer in Ada95

- ```
  task body Producer is begin
    loop
      item := ProduceItem;
      BounderBuffer.Put( item );
    end loop;
  end Producer;

  task body Consumer is begin
    loop
      BounderBuffer.Get( item );
      ConsumeItem( item );
    end loop;
  end Consumer;
  ```
  ```

Semaphore in Ada95

```
• CountingSemaphore( StartCount : Integer := 1 ) is
    entry Wait;
    entry Signal;
    entry Count( count_out : out integer );
  private
    CurrentCount : Integer := StartCount;
  end CountingSemaphore;
  task body CountingSemaphore is begin
    loop
      select
        when CurrentCount > 0 =>
          accept Wait do
            CurrentCount := CurrentCount - 1;
          end;
      or
        accept Signal do
          CurrentCount := CurrentCount + 1;
        end;
      or
        accept Count( count_out : out integer ) do
          count_out := CurrentCount;
        end;
      end select;
    end loop;
  end CountingSemaphore;
```

Protected counter in Ada95 using protecting variables

- ```
  -- the interface to the protected variable
  protected Counter is
    function GetCount return integer;
    procedure Increment;
  private
    count : integer;
  end Counter;
  -- the implementation of the protected variable
  protected body Counter is
    function GetCount return integer is begin
      return count;
    end GetCount;
    procedure Increment is begin
      count := count + 1;
    end Increment;
  end Counter;
  ```
  ```

Using a protected counter

- ```
  task body OneProcess is begin
    loop
      -- do other things than incrementing the counter
      Counter.Increment;
      -- do other things
      n := Counter.GetCount;
    end loop;
  end OneProcess;

  task body AnotherProcess is begin
    loop
      -- do other things than incrementing the counter
      Counter.Increment;
      -- do other things
    end loop;
  end AnotherProcess;
  ```
  ```

Signaling in Ada95 using protected variables

- ```
  protected Signal is
    procedure SendSignal;
    entry WaitForSignal;
  private
    IsSignaled : boolean := False;
  end Signal;

  protected body Signal is
    procedure SendSignal is begin
      IsSignaled := True;
    end SendSignal;
    entry WaitForSignal when IsSignaled is begin
      null;
    end WaitForSignal;
  end Signal;
  ```
  ```

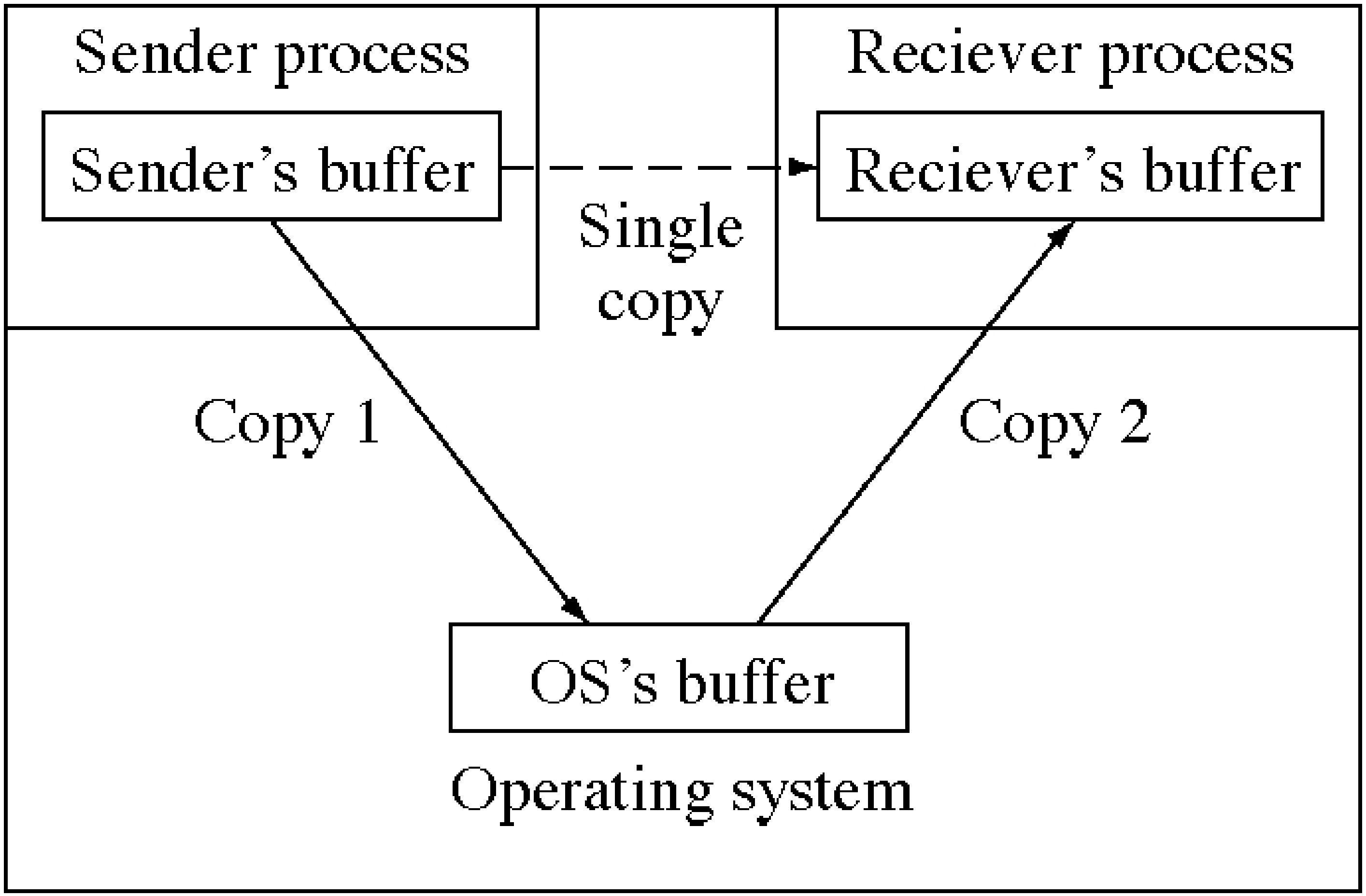
Using signals in Ada95

- ```
  task body SignalSender is begin
    -- get to point where event occurs
    Signal.SendSignal;
    -- go on to other things
  end SignalSender;

  task body SignalReceiver is begin
    -- get to the point where you need to wait
    -- for the event
    Signal.WaitForSignal;
    -- respond to event
  end SignalReceiver;
  ```
  ```

Bounder buffer in Ada95

- ```
  protected type BoundedBuffer is
    entry Put( x : in Item );
    entry Get( x : out Item );
  private
    buffers : ItemArray(1..NumberOfBuffers);
    next_in, next_out
      : integer range 1..NumberOfBuffers := 1;
    current_size
      : integer range 0..NumberOfBuffers := 0;
  end BoundedBuffer;
  protected body BoundedBuffer is
    entry Put( x : in Item )
      when current_size < NumberOfBuffers is begin
        buffers(next_in) := x;
        next_in := (next_in mod NumberOfBuffers) + 1;
        current_size := current_size + 1;
    end Put;
    entry Get( x : out Item )
      when current_size > 0 is begin
        x := buffers(next_out);
        next_out := (next_out mod NumberOfBuffers) + 1;
        current_size := current_size - 1;
    end Get;
  end BoundedBuffer;
  ```
  ```

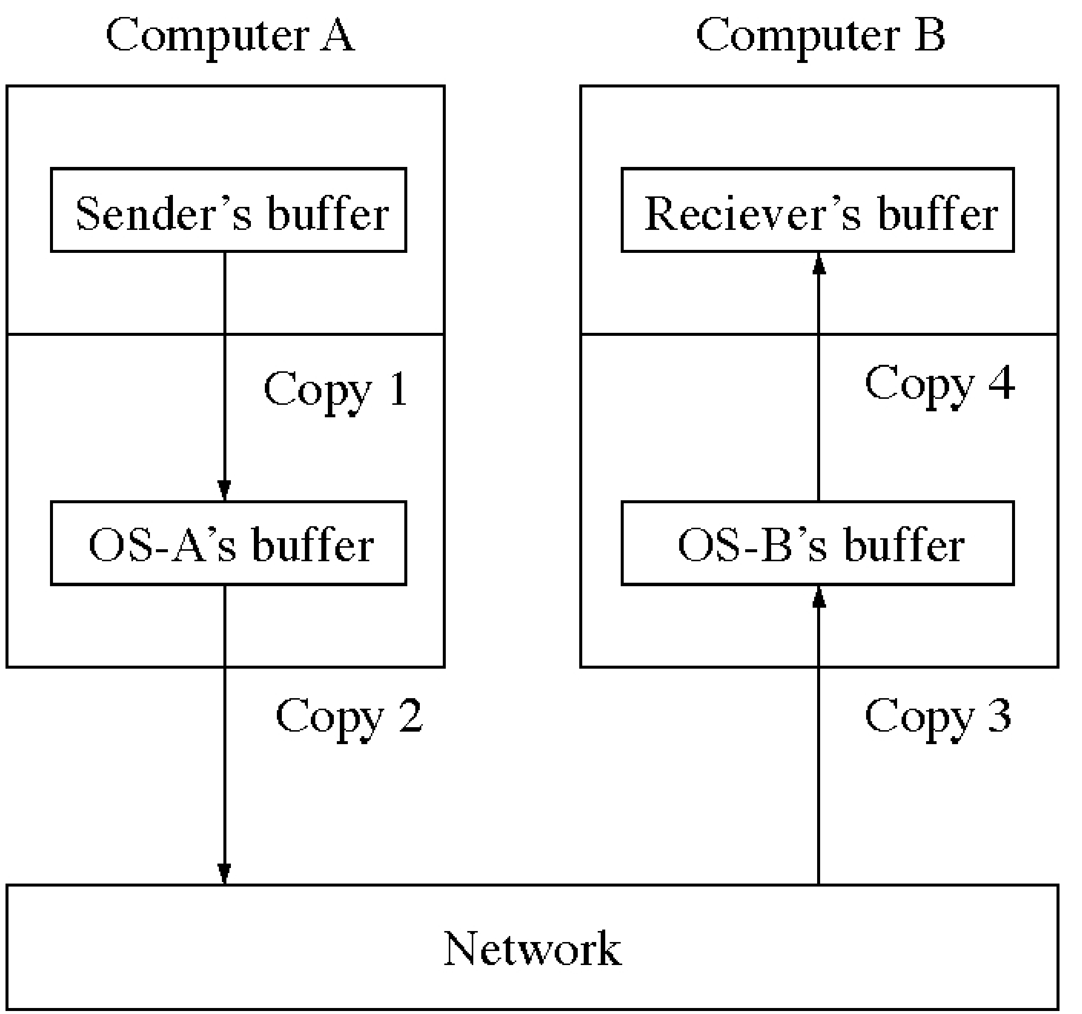
Producter-consumer in Ada95

- ```
  task body Producer is begin
    loop
      item := ProduceItem;
      BounderBuffer.Put( item );
    end loop;
  end Producer;

  task body Consumer is begin
    loop
      BounderBuffer.Get( item );
      ConsumeItem( item );
    end loop;
  end Consumer;
  ```
  ```

Counting semaphore in Ada95 using protected variables

- ```
  protected type
      CountingSemaphore( StartCount : Integer := 1 ) is
    entry Wait;
    procedure Signal;
    function Count return Integer;
  private
    CurrentCount : Integer := StartCount;
  end CountingSemaphore;
  protected body CountingSemaphore is
    entry Wait when CurrentCount > 0 is begin
      CurrentCount := CurrentCount - 1;
    end Wait;
    procedure Signal is begin
      CurrentCount := CurrentCount + 1;
    end Signal;
    function Count return Integer is begin
      return CurrentCount;
    end Count;
  end CountingSemaphore;
  ```
  ```

Copying messages in an OS



Copying messages in a network



Longer messages

- Eight words is too small
- Paging (chapter 11) will allow us to pass long messages with no copying

IPC in Mach

- Task: Mach terminology for process
- Thread: execute in a task
- Port: a message queue
  - only one receiver
  - protected with access rights and capabilities
- Message: contains three types of information
  - data: any number of bytes, copied
  - out-of-line-data: part of the sender's address space, not copied
  - ports

Signals

- Used in UNIX system for IPC
- An event notification
  - 30 or so fixed signal (event) types
- Basically a software interrupt
  - process defines the interrupt handler