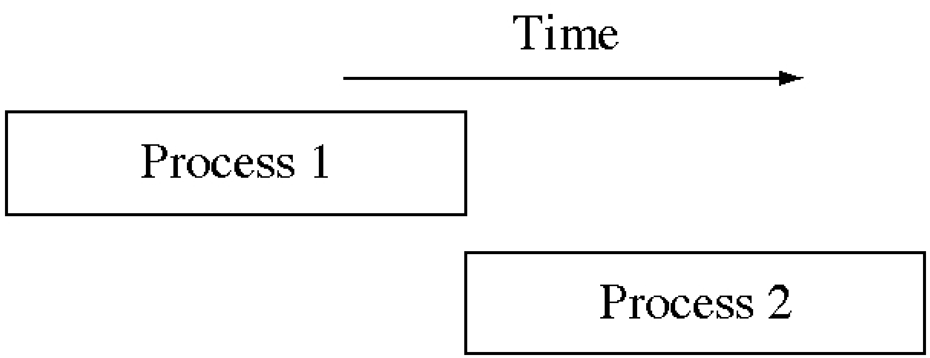
Implementing Processes

Chapter 5

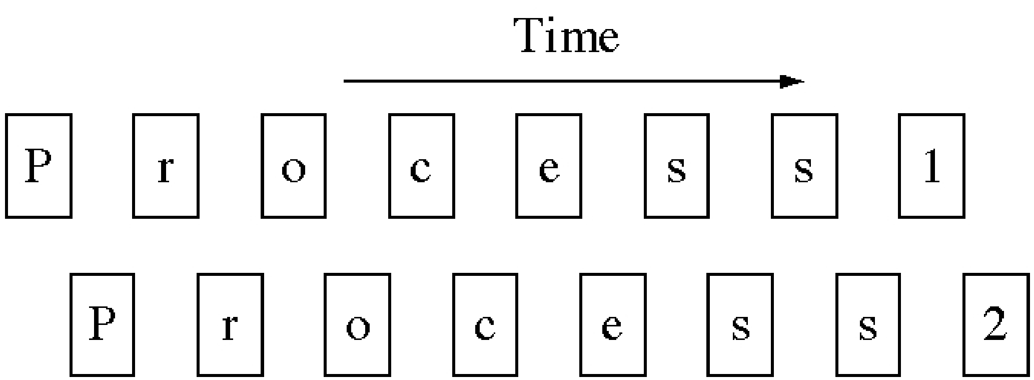
Key concepts in chapter 5

- Simple operating systems (SOS)
  - Implementation of processes
  - System initialization
  - Process switching
  - System call handling
  - Waiting in the OS
- Operating systems as table and event managers

Implementing processes by interleaving the processor



(a) Serial execution of processes 1 and 2

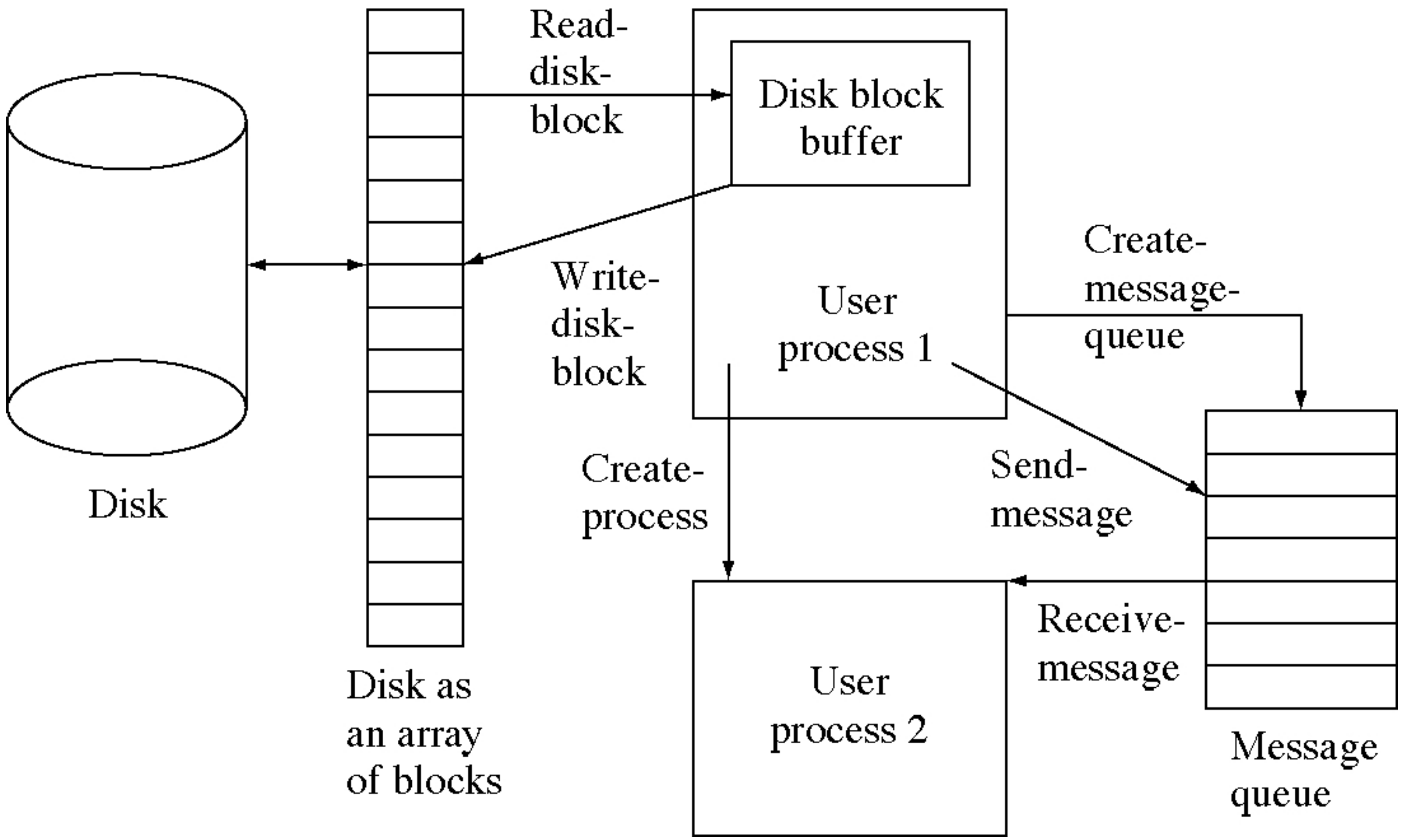


(b) Interleaved execution of processes 1 and 2

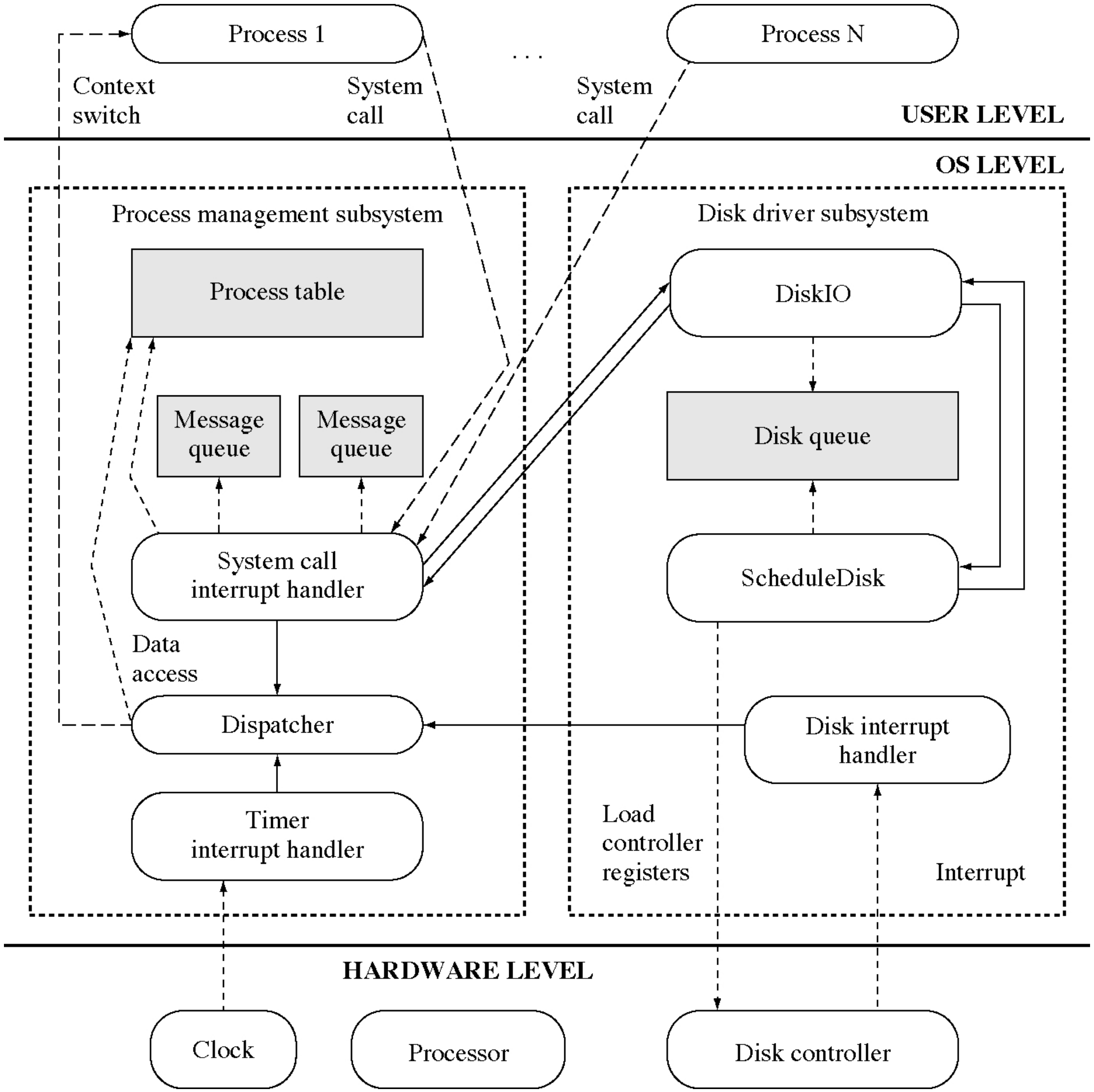
System call interface of the Simple OS (SOS)

- CreateProcess(int firstBlock, int nBlocks)
- ExitProcess(int exitCode)
- CreateMessageQueue()
- SendMessage(int mqid, int *msg)
- ReceiveMessage(int mqid, int *msg)
- ReadDiskBlock(int block, char *buffer)
- WriteDiskBlock(int block, char *buffer)

SOS objects and operations



SOS architecture



System constants (1 of 2)

- ```
  // Boolean values
  enum { False=0, True=1 };

  // hardware constants (determined by the hardware)
  const int DiskBlockSize     = 4096;
  const int NumberOfRegisters = 32;

  // system constants (we can change these constants
  //   to tune the operating system)
  const int SystemStackSize = 4096;  // bytes
  const int ProcessSize = 512*1024; // bytes
  const int TimeQuantum = 100000;
    // 100000 microseconds = 100 milliseconds
  const int MessageSize = 8; // 8 words = 32 bytes
  const int InitialProcessDiskBlock = 4341; //disk block #
  const int EndOfFreeList = -1;
  ```
  ```

System constants (2 of 2)

```
// system limits (we can change these)
const int NumberOfProcesses = 20;
const int NumberOfMessageQueues = 20;
// The total number of message buffers
const int NumberOfMessageBuffers = 100;
// event handler offsets (determined by the hardware)
const int SystemCallHandler = 0;
const int TimerHandler = 4;
const int DiskHandler = 8;
const int ProgramErrorHandler = 12;
// system call numbers (arbitrary numbers,
//   as long as they are all different)
const int CreateProcessSystemCall = 1;
const int ExitProcessSystemCall = 2;
const int CreateMessageQueueSystemCall = 3;
const int SendMessageSystemCall = 4;
const int ReceiveMessageSystemCall = 5;
const int DiskReadSystemCall = 6;
const int DiskWriteSystemCall = 7;
```

Process global data

- ```
  struct SaveArea {
      int ia, psw, base, bound,
  reg[NumberOfRegisters];
  };
  enum ProcessState { Ready, Running, Blocked };
  typedef int Pid;

  struct ProcessDescriptor {
    int slotAllocated;
    int timeLeft; // time left from the last time
  slice
    ProcessState state; // ready, running or blocked
    SaveArea sa; // register save area
  };

  int current_process
  int SystemStack[SystemStackSize];
  ProcessDescriptor pd[NumberOfProcesses];
      // pd[0] is the system
  ```
  ```

Message global data

- ```
  typedef int MessageBuffer[MessageSize];

  MessageBuffer
  message_buffer[NumberOfMessageBuffers];

  int free_message_buffer;

  int message_queue_allocated[NumberOfMessageQueues];

  Queue<int> * message_queue[NumberOfMessageQueues];

  struct WaitQueueItem {
    Pid pid;
    char * buffer;
  };

  Queue<WaitQueueItem *> *
    wait_queue[NumberOfMessageQueues];
  ```
  ```

Interrupt vector area

- ```
  char * SystemCallVector
    = &SystemCallInterruptHandler;

  char * TimerVector
    = &TimerInterruptHandler;

  char * DiskVector
    = &DiskInterruptHandler;

  char * ProgramErrorVector
    = &ProgramErrorInterruptHandler
  ```
  ```

Disk global data

- ```
  int process_using_disk;

  struct DiskRequest {
    int command;
    int disk_block;
    char * buffer;
    int pid;
  };

  Queue<DiskRequest *> disk_queue;
  ```
  ```

Message buffer allocation procedures

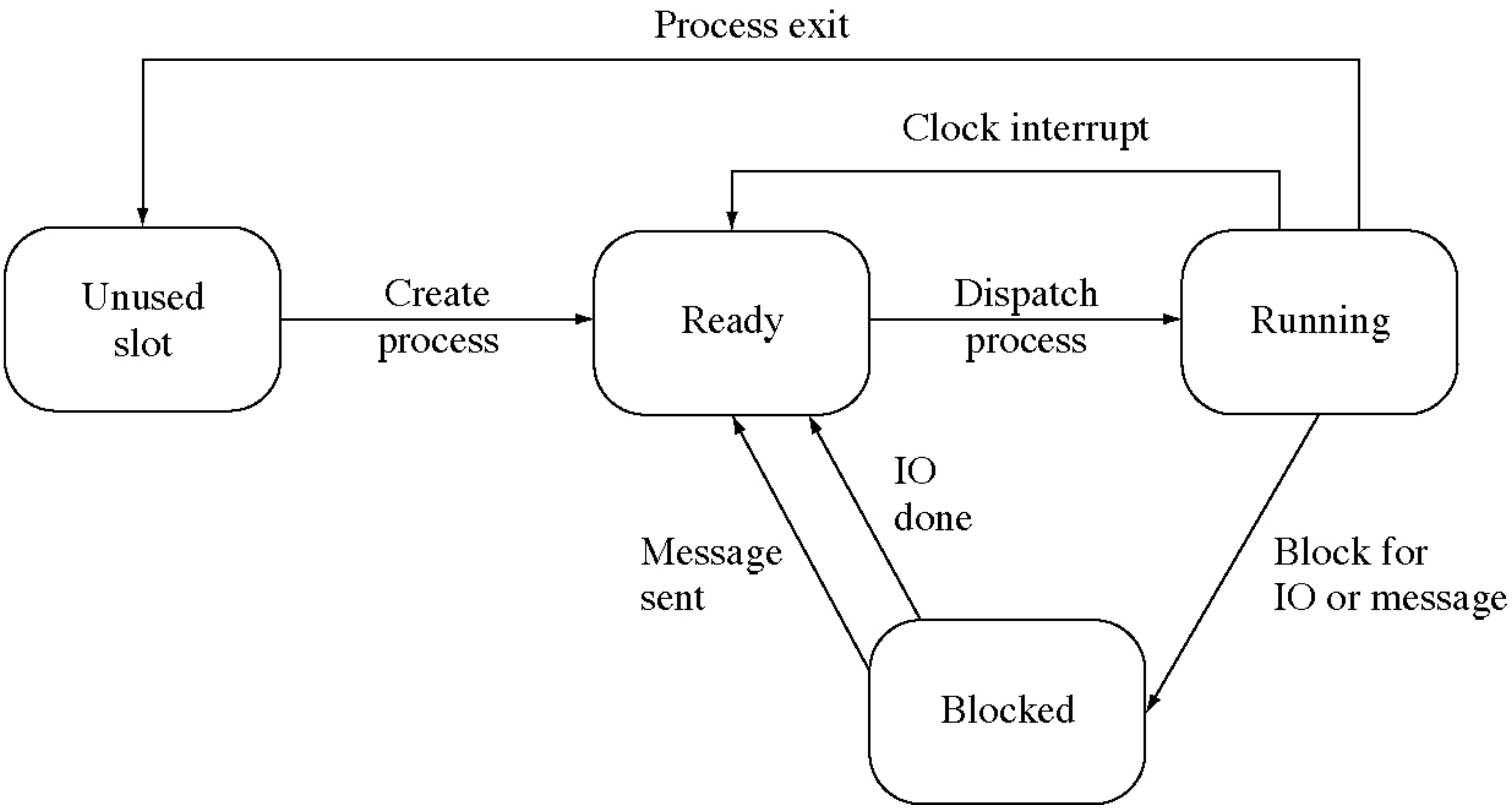
- ```
  int GetMessageBuffer( void ) {
    // get the head of the free list
    int msg_no = free_message_buffer;
    if( msg_no != EndOfFreeList ) {
      // follow the link to the next buffer
      free_message_buffer = message_buffer[msg_no][0];
    }
    return msg_no;
  }

  void FreeMessageBuffer( int msg_no ) {
    message_buffer[msg_no][0] = free_message_buffer;
    free_message_buffer = msg_no;
  }
  ```
  ```

Process creation

- ```
  int CreateProcessSysProc(int first_block,int n_blocks) {
    int pid;
    for( pid = 1; pid < NumberOfProcesses; ++pid ) {
      if( pd[pid].slotAllocated ) break;
    }
    if( pid >= NumberOfProcesses ) { return -1; }
    pd[pid].slotAllocated = True;
    pd[pid].state = Ready;
    pd[pid].sa.base = pid * ProcessSize;
    pd[pid].sa.bound = ProcessSize;
    pd[pid].sa.psw = 3; // user mode, interrupts enabled
    pd[pid].sa.ia = 0;
    char * addr = (char *)(pd[pid].sa.base);
    for( i = 0; i < n_blocks; ++i ) {
      while( DiskBusy() ) ;
      IssueDiskRead( first_block + i, addr, 0/*no int*/);
      addr += DiskBlockSize;
    }
    return pid;
  }
  ```
  ```

Process states



Dispatcher

- ```
  void Dispatcher( void ) {
    current_process = SelectProcessToRun();
    RunProcess( current_process );
  }
  ```
  ```

Select a process to run

- ```
  int SelectProcessToRun( void ) {
    static int next_proc = NumberOfProcesses;
    if( current_process > 0
        && pd[current_process].state == Ready
        && pd[current_process].timeLeft > 0 ) {
      pd[current_process].state = Running;
      return current_process;
    }
    for( int i = 1; i < NumberOfProcesses; ++i ) {
      if( ++next_proc >= NumberOfProcesses )
        next_proc = 1;
      if( pd[next_proc].slotAllocated
          && pd[next_proc].state == Ready ) {
        pd[next_proc].timeLeft = TimeQuantum;
        pd[next_proc].state = Running;
        return next_proc;
      }
    }
    return -1;
  }
  ```
  ```

Run a process

- ```
  void RunProcess( int pid ) {
    if( pid >= 0 ) {
      SaveArea * savearea = &(pd[pid].sa);
      int quantum = pd[pid].timeLeft;
      asm {
        load    savearea+0,iia
        load    savearea+4,ipsw
        load    savearea+8,base
        load    savearea+12,bound
        loadall savearea+16
        load    quantum,timer
        rti
      }
    } else {
      waitLoop: goto waitLoop;
    }
  }
  ```
  ```

The system stack

- All code needs a stack
  - the compiler expects to have one for us by the running program
- We play tricks on the C++ compiler and fiddle with its stack

Timer interrupt handler

```
• void TimerInterruptHandler( void ) {
    if( current_process > 0 ) {
      SaveArea * savearea = &(pd[current_process].sa);
      asm {
        store    iia,savearea+0
        store    ipsw,savearea+4
        store    base,savearea+8
        store    bound,savearea+12
        storeall savearea+16
        load     SystemStack+SystemStackSize,r30
      }
    }
    pd[current_process].timeLeft = 0;
    pd[current_process].state = Ready;
    Dispatcher();
  }
```

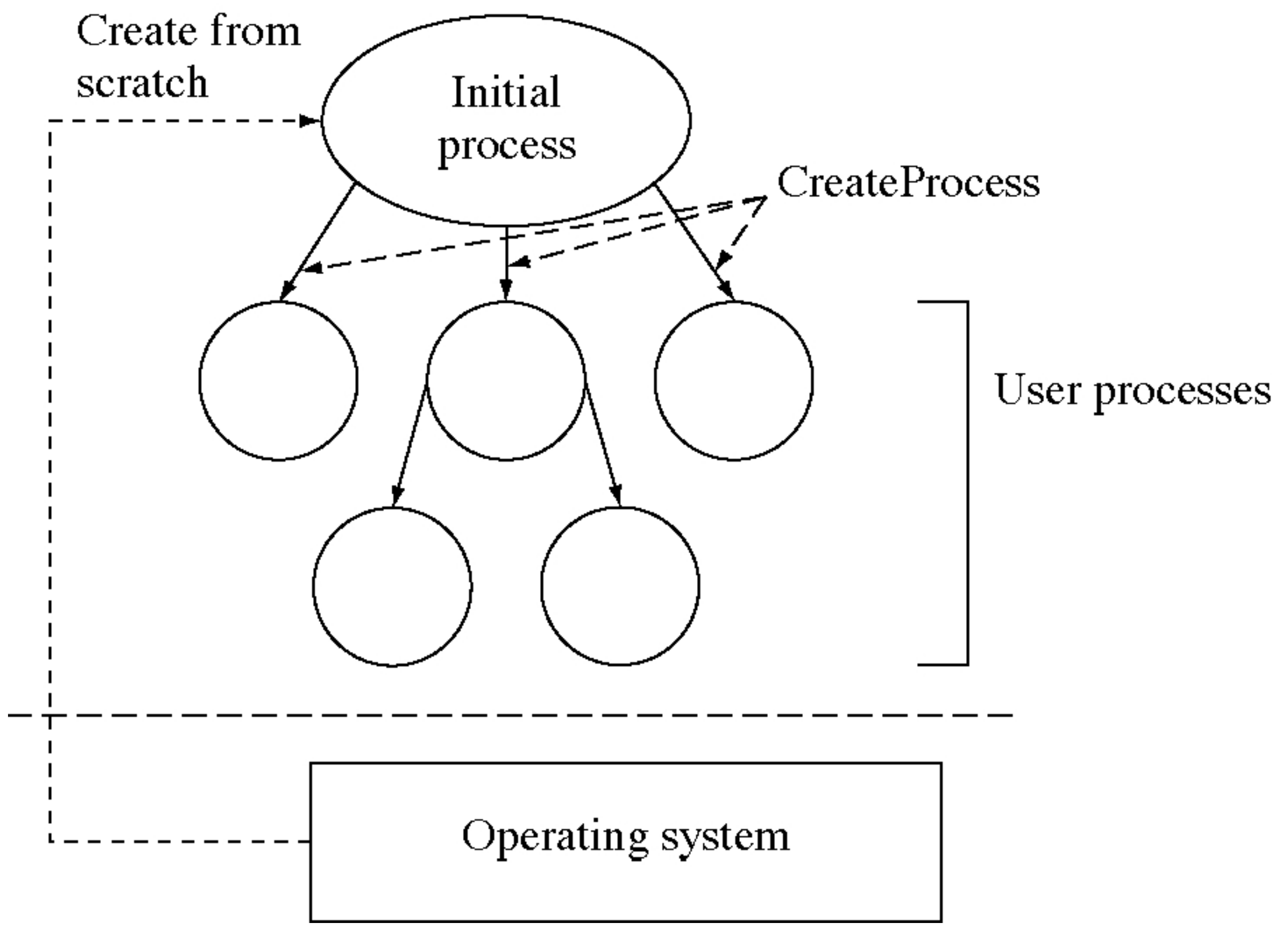
System initialization

- ```
  int main( void ) {
    asm{ load  SystemStack+SystemStackSize,r30 }
    asm{ load  &SystemCallVector,iva }
    pd[0].slotAllocated = True;
    pd[0].state = Blocked;
    for( i = 1; i < NumberOfProcesses; ++i )
      pd[i].slotAllocated = False;
    (void)CreateProcessSysProc(
       InitialProcessDiskBlock, 1 );
    for( i = 0; i < (NumberOfMessageBuffers-1); ++i )
      message_buffer[i][0] = i + 1;
    message_buffer[NumberOfMessageBuffers-1][0]
      = EndOfFreeList;
    free_message_buffer = 0;
    for( i = 0; i < NumberOfMessageQueues; ++i )
      message_queue_allocated[i] = False;
    process_using_disk = 0;
    Dispatcher();
  }
  ```
  ```

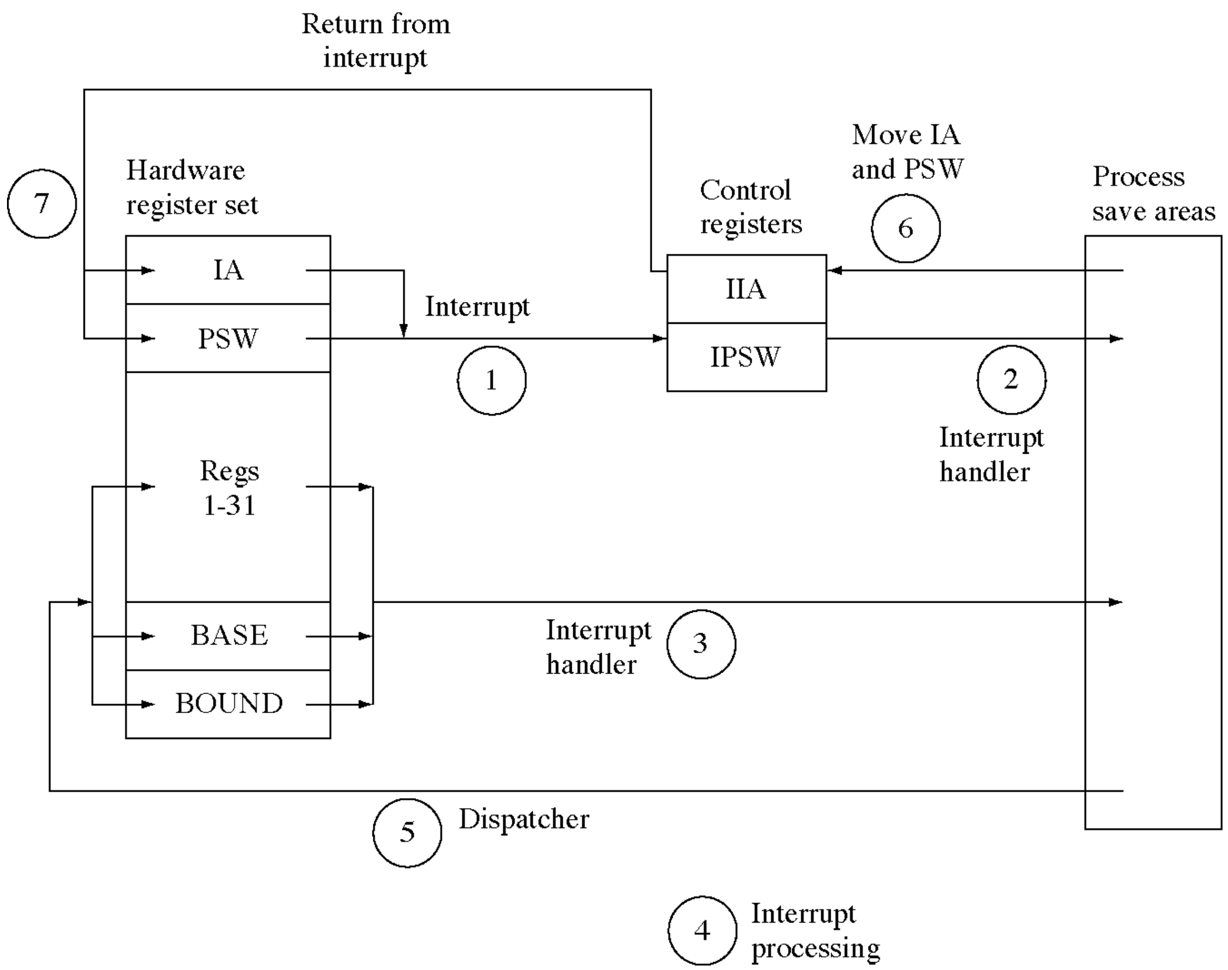
The initial process

- ```
  void main() {
    // start the two counting processes
    (void)CreateProcess(
       UserProcessA, UserProcessASize );
    (void)CreateProcess(
       UserProcessB, UserProcessBSize );
    // Nothing else for this process to do.
    // We haven't implemented a Wait system call,
    // so just exit.
    ExitProcess( 0 );
  }
  ```
  ```

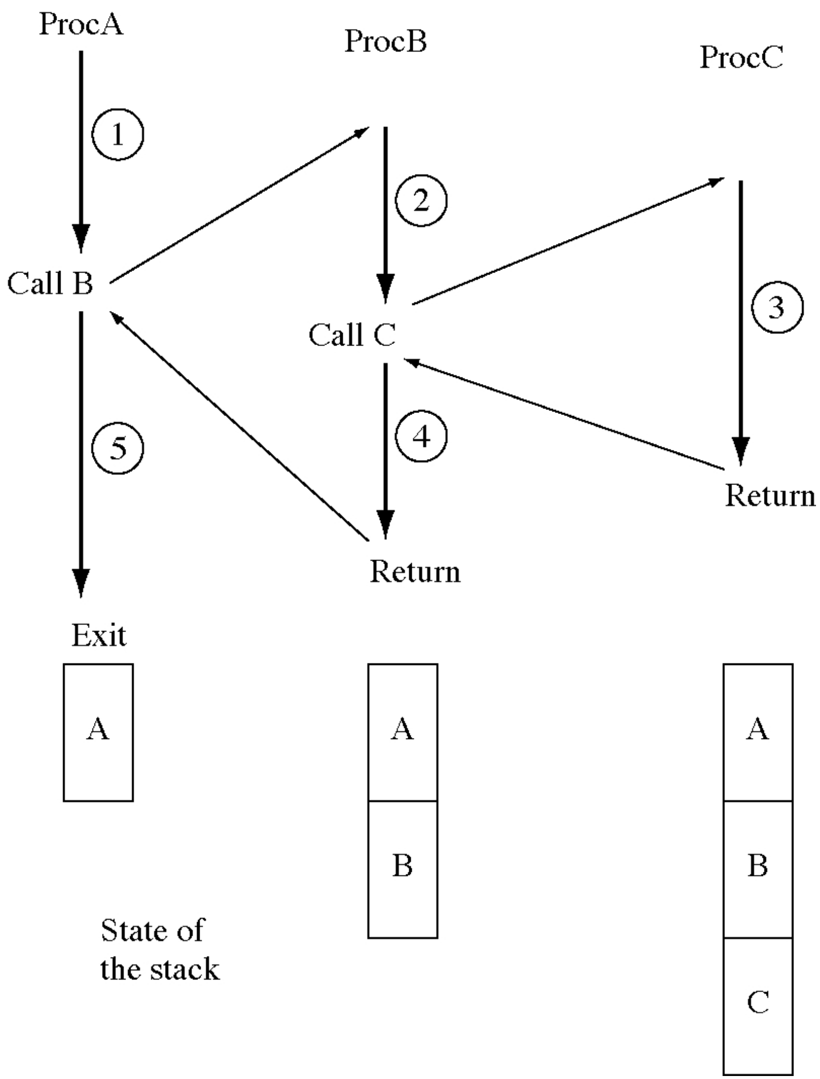
Initial process creates other processes



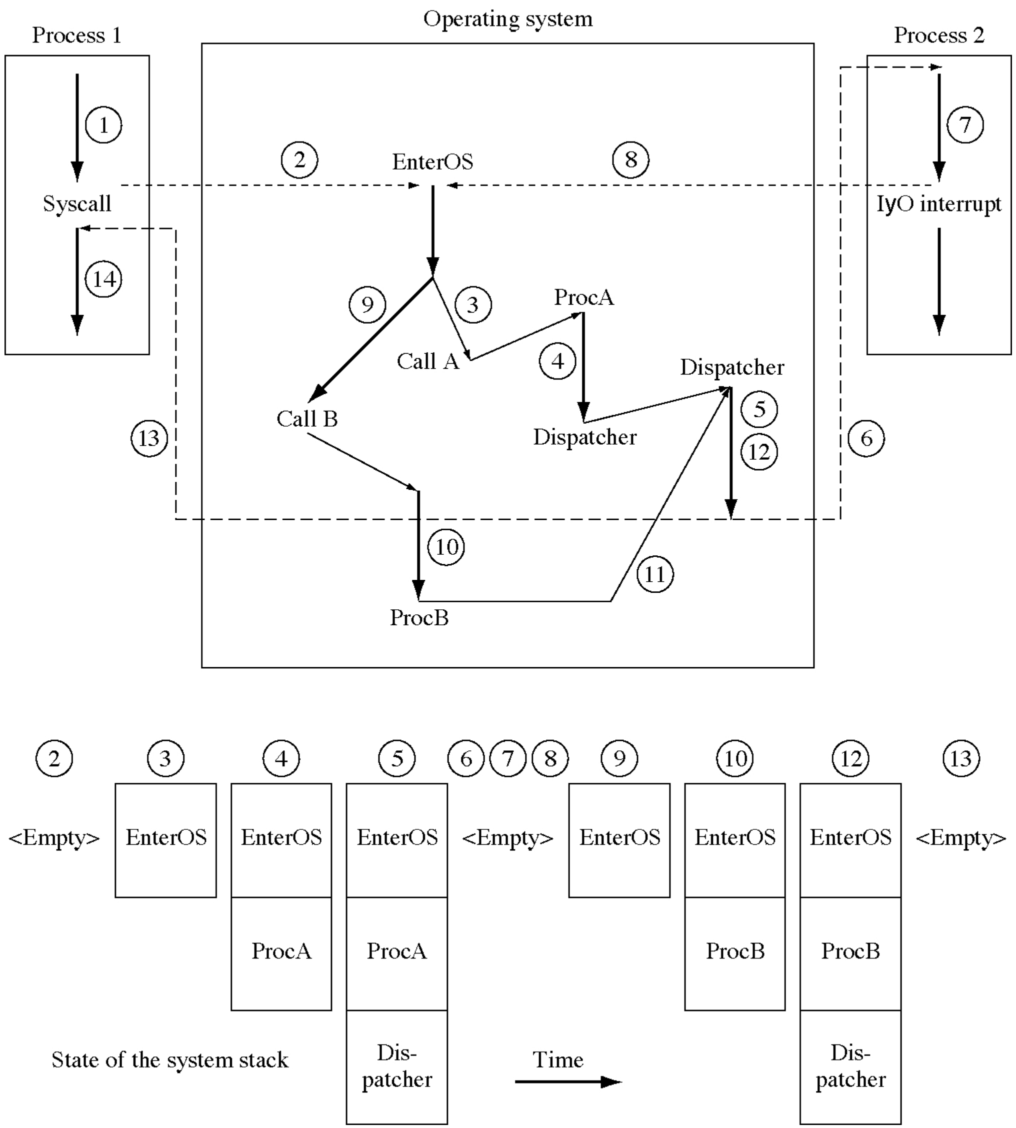
Process switching



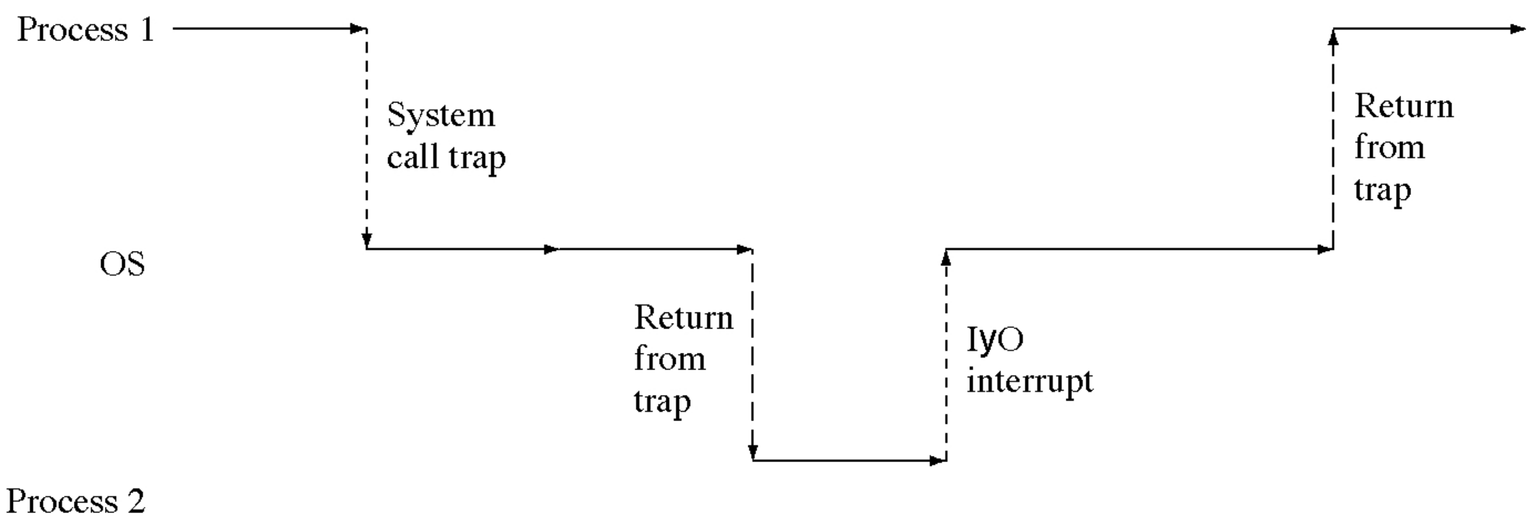
Flow of control within a process



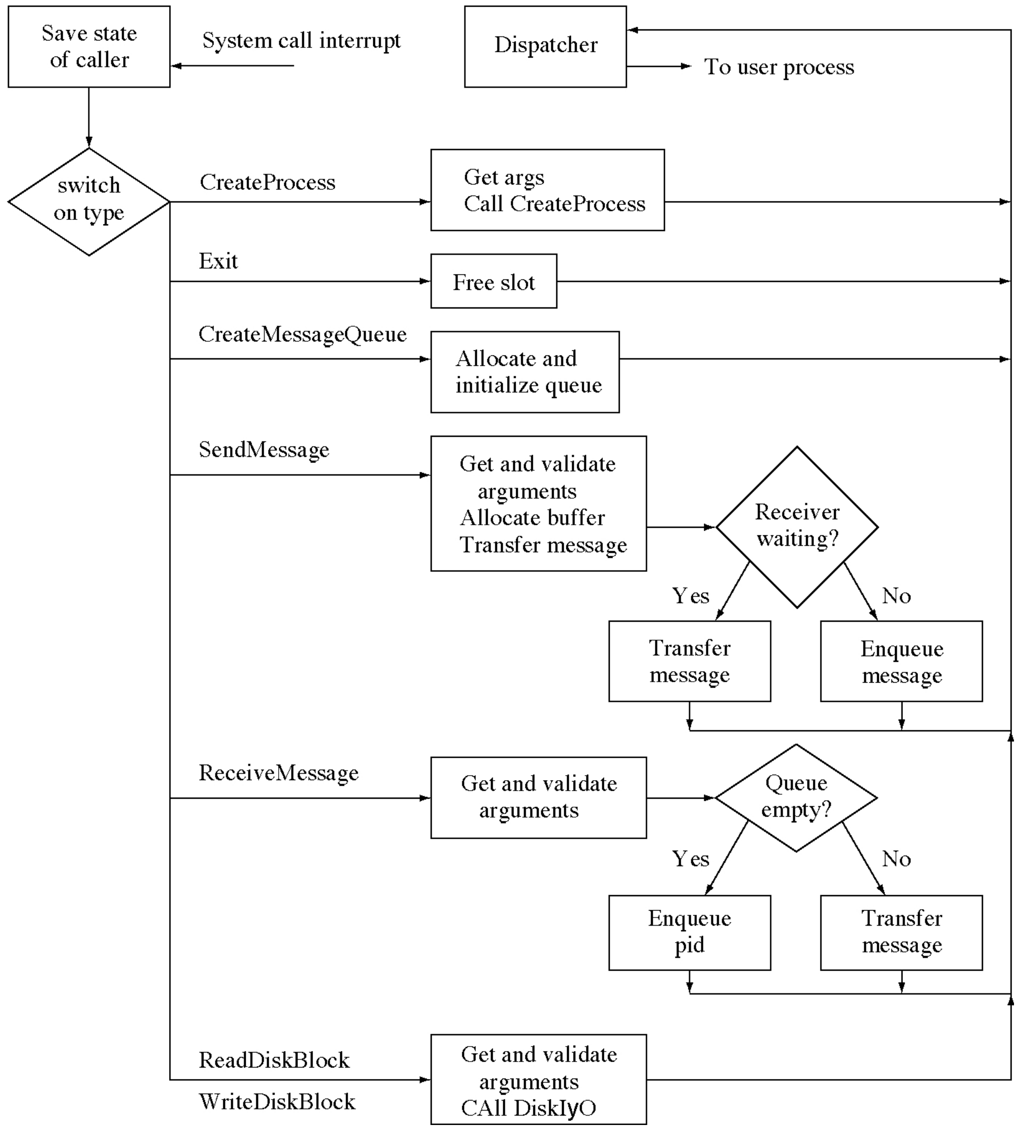
Process switching control flow



Flow of control during process switching (another view)



System call interrupt handler



System call interrupt handler (1 of 6)

```
void SystemCallInterruptHandler( void ) {
  SaveArea * savearea = &(pd[current_process].sa);
  int saveTimer;
  asm {
    store    timer,saveTimer
    load     #0,timer
    store    iia,savearea+0
    store    ipsw,savearea+4
    store    base,savearea+8
    store    bound,savearea+12
    storeall savearea+16
    load     SystemStack+SystemStackSize,r30
  }
  pd[current_process].timeLeft = saveTimer;
  pd[current_process].state = Ready;
  int system_call_number;
  asm { store r8,system_call_number }
  switch( system_call_number ) {
```

System call interrupt handler (2 of 6)

```
• case CreateProcessSystemCall:
    // get the system call arguments from the
  registers
    int block_number; asm { store r9,block_number }
    int number_of_blocks;
    asm { store r10,number_of_blocks }
    // put the return code in R1
    pd[current_process].sa.reg[1]
      = CreateProcessSysProc(
          block_number,number_of_blocks);
    break;
  case ExitProcessSystemCall:
    char * return_code; asm { store r9,return_code }
    // we don't save the return code in this OS so
    // just free up the pd slot
    pd[current_process].slotAllocated = False;
    break;
```

System call interrupt handler (3 of 6)

- ```
  case CreateMessageQueueSystemCall:
    // find a free message queue
    int i;
    for( i = 0; i < NumberOfMessageQueues; ++i ) {
      if( !message_queue_allocated[i] ) {
        break;
      }
    }
    if( i >= NumberOfMessageQueues ) {
      // signal the error, message queue overflow
      // return a value that is invalid
      pd[current_process].sa.reg[1] = -1;
      break;
    }
    message_queue_allocated[i] = True;
    message_queue[i] = new Queue<int>;
    wait_queue[i] = new Queue<WaitQueueItem *>;
    pd[current_process].sa.reg[1] = i;
    break;
  ```
  ```

System call interrupt handler (4 of 6)

- ```
  case SendMessageSystemCall:
    int * user_msg; asm { store r9,user_msg }
    int to_q; asm { store r10,to_q }
    if( !message_queue_allocated[to_q] ) {
      pd[current_process].sa.reg[1] = -1;
      break; }
    int msg_no = GetMessageBuffer();
    if( msg_no == EndOfFreeList ) {
      pd[current_process].sa.reg[1] = -2;
      break
    }
    CopyToSystemSpace( current_process, user_msg,
      message_buffer[msg_no], MessageSize );
    if( !wait_queue[to_q].Empty() ) {
      WaitQueueItem item = wait_queue[to_q].Remove();
      TransferMessage( msg_no, item.buffer );
      pd[item.pid].state = Ready;
    } else
      message_queue[to_q].Insert( msg_no );
    pd[current_process].sa.reg[1] = 0;
    break;
  ```
  ```

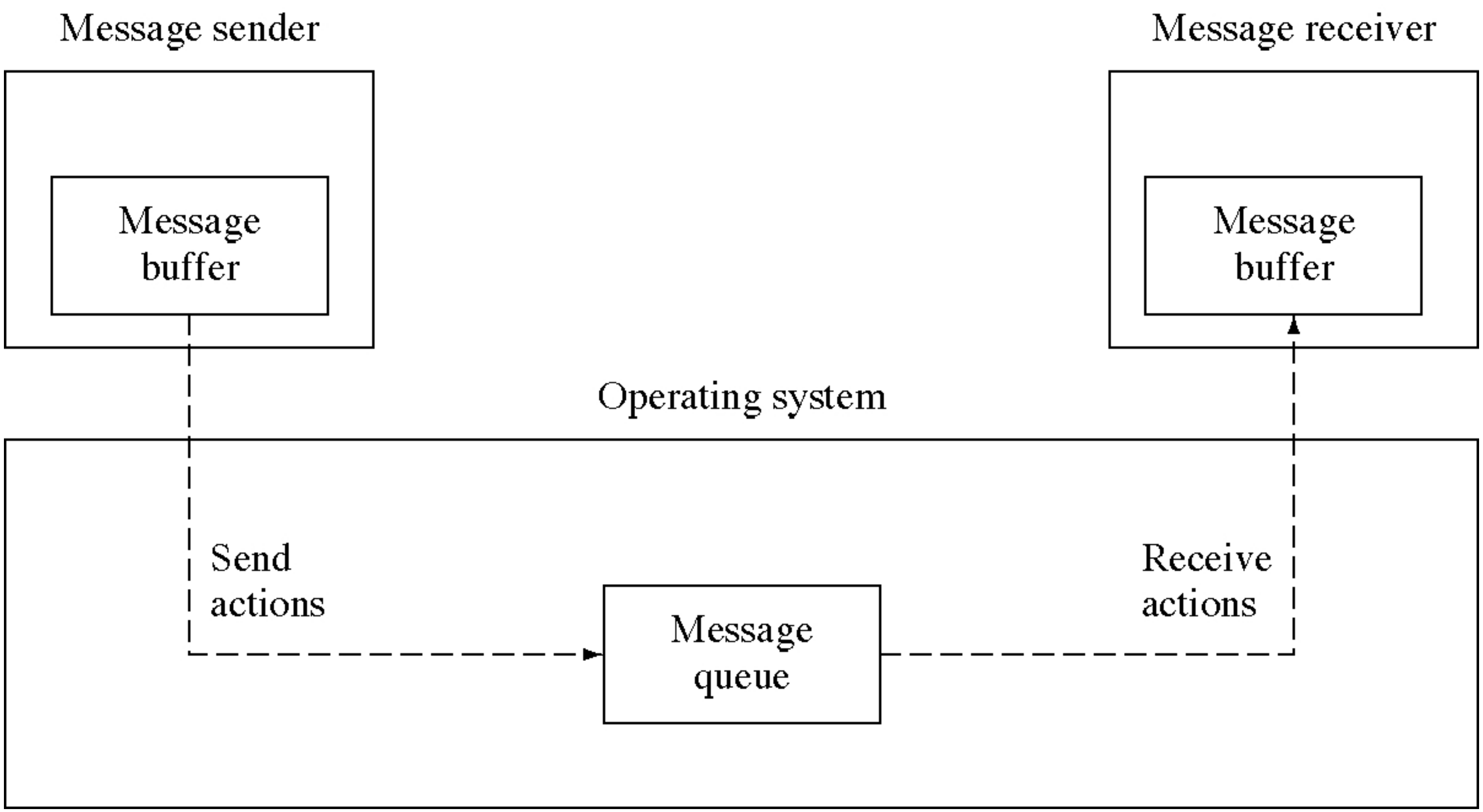
System call interrupt handler (5 of 6)

- ```
  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;
    }
    if( message_queue[from_q].Empty() ) {
      pd[current_process].state = Blocked;
      WaitQueueItem item;
      item.pid = current_process;
      item.buffer = user_msg;
      wait_queue[from_q].Insert( item );
    } else {
      int msg_no = message_queue[from_q].Remove();
      TransferMessage( msg_no, user_msg );
    }
    pd[current_process].sa.reg[1] = 0;
    break;
  ```
  ```

System call interrupt handler (6 of 6)

```
•   case DiskReadSystemCall:
    case DiskWriteSystemCall:
      char * buffer; asm { store r9,buffer }
      buffer += pd[current_process].sa.base;
      // convert to physical address
      int disk_block; asm { store r10,disk_block }
      DiskIO( system_call_number, disk_block,
  buffer );
      pd[current_process].sa.reg[1] = 0;
      break;
    }
    Dispatcher();
  }
```

Send and receive cctions



Transfer between system and user memory

- ```
  void CopyToSystemSpace(
          int pid, char * from, char * to, int len ) {
      from += pd[pid].sa.base;
      while( len-- > 0 )
          *to++ = *from++;
  }

  void CopyFromSystemSpace(
          int pid, char * to, char * from, int len ) {
      to += pd[pid].sa.base;
      while( len-- > 0 )
          *to++ = *from++;
  }
  ```
  ```

Program error interrupt handler

- ```
  void ProgramErrorInterruptHandler( void ) {
    asm {
      // stop the interval timer
      //   and clear any pending timer interrupt
      load     #0,timer
      // no need to save the processor state
      //
      // set up the stack
      load     SystemStack+SystemStackSize,r30
    }
    pd[current_process].slotAllocated = False;
    Dispatcher();
  }
  ```
  ```

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->pid = current_process;
    // Then insert it on the queue.
    disk_queue.Insert( req );
    pd[current_process].state = Blocked;
    // Wake up the disk scheduler if it is idle.
    ScheduleDisk();
  }
  ```
  ```

Disk scheduling

- ```
  void ScheduleDisk( void ) {
    // If the disk is already busy
    if( DiskBusy() ) return;
    DiskRequest * req = disk_queue.Remove();
    // no disk request to service so return.
    if( req == 0 ) return;
    // remember process waiting for the disk operation
    process_using_disk = req->pid;
    // issue read or write, disk interrupt enabled
    if( req->command == DiskReadSystemCall )
      IssueDiskRead( req->disk_block, req->buffer,
  1 );
    else
      IssueDiskWrite(
        req->disk_block, req->buffer, 1 );
  }
  ```
  ```

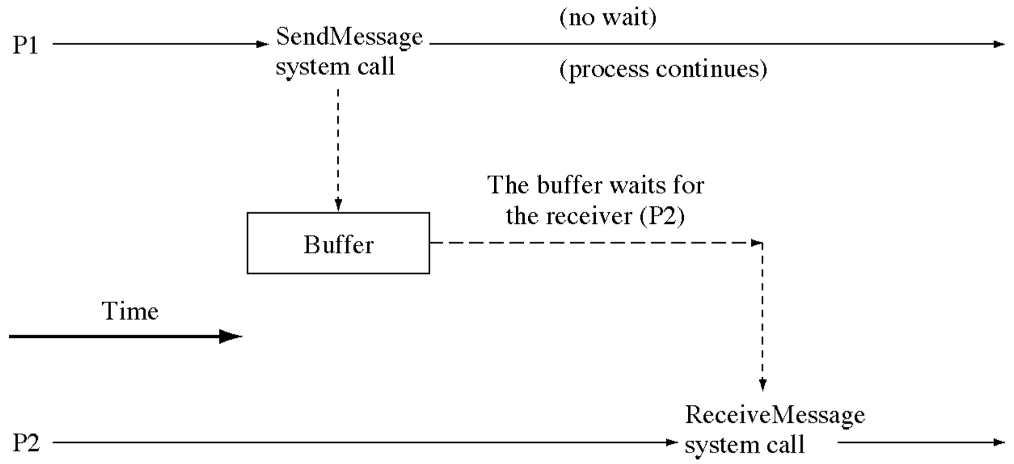
Disk interrupt handler

- ```
  void DiskInterruptHandler( void ) {
    if( current_process > 0 ) {
      SaveArea * savearea = &(pd[current_process].sa);
      int saveTimer;
      asm { store    timer,saveTimer
            load     #0,timer
            store    iia,savearea+0
            store    ipsw,savearea+4
            store    base,savearea+8
            store    bound,savearea+12
            storeall savearea+16
            load     SystemStack+SystemStackSize,r30 }
      pd[current_process].timeLeft = saveTimer;
      pd[current_process].state = Ready; }
    pd[process_using_disk].state = Ready;
    process_using_disk = 0;
    ScheduleDisk();
    Dispatcher();
  }
  ```
  ```

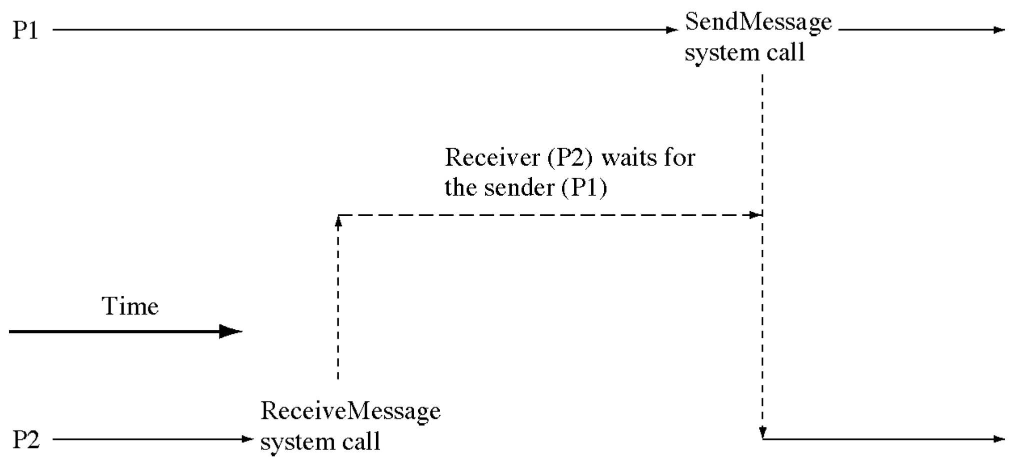
Disk interface implementation

- ```
  int DiskBusy( void ) {
    disk_status_reg stat = *Disk_status;
    return stat.busy;
  }
  void IssueDiskRead( int block_number, char * buffer,
      int enable_disk_interrupt ) {
    disk_control_reg control_reg;
    // assemble the necessary control word
    control_reg.command = 1;
    control_reg.disk_block = block_number;
    control_reg.interrupt_enabled
      = enable_disk_interrupt;
    // store the control words
    //   in the disk control register
    *Disk_memory_addr = buffer;
    *Disk_control_reg = control_reg;
  }
  ```
  ```

Waiting for messages



(a) SendMessage occurs before the ReceiveMessage



(b) ReceiveMessage occurs before the SendMessage

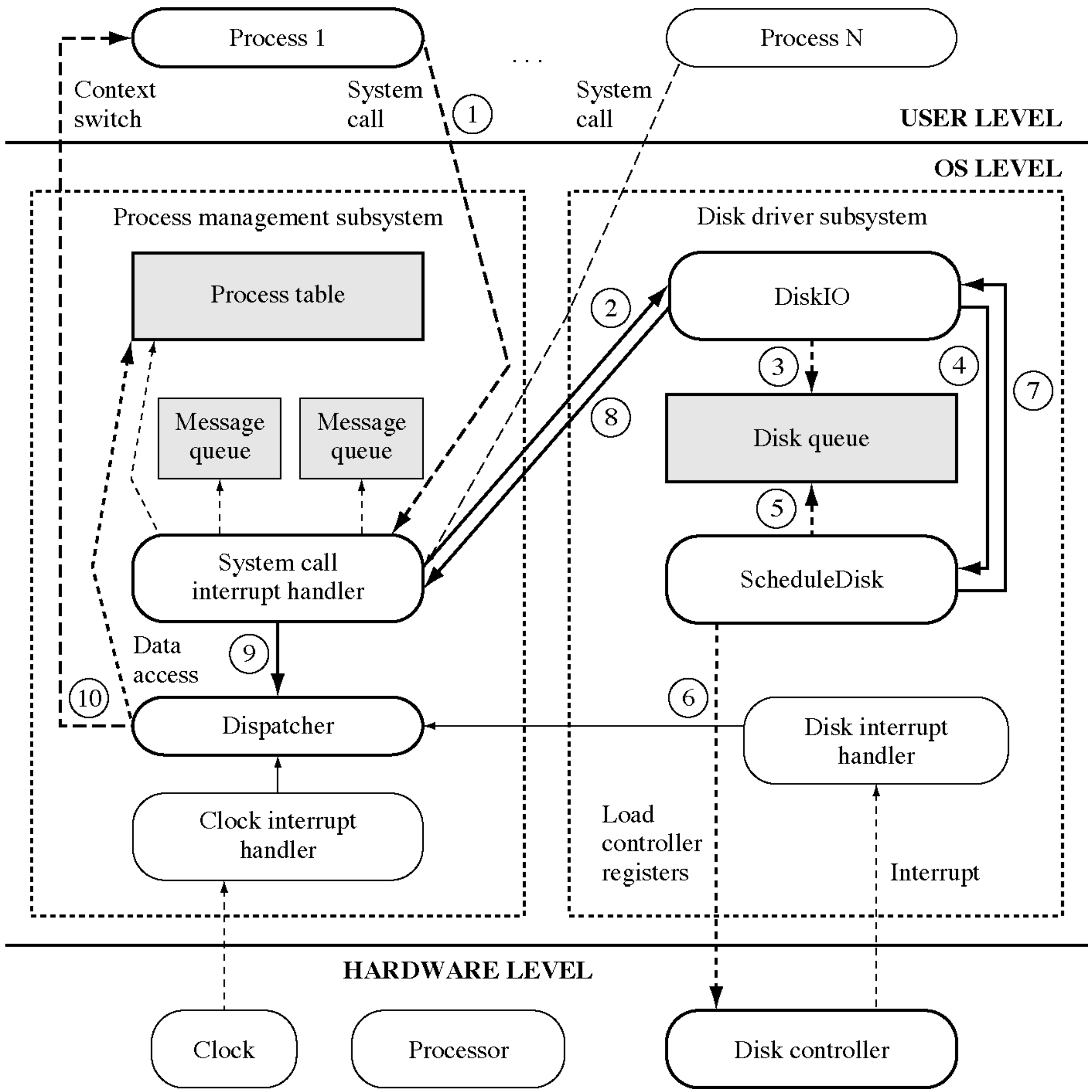
Waiting inside a system call

- Some systems calls must wait
  - E.g. ReceiveMessage, ReadDiskBlock
- The OS suspends the process and saves its state
  - but how does the state of the OS processing the system call get saved?
  - Special provision must be made for this

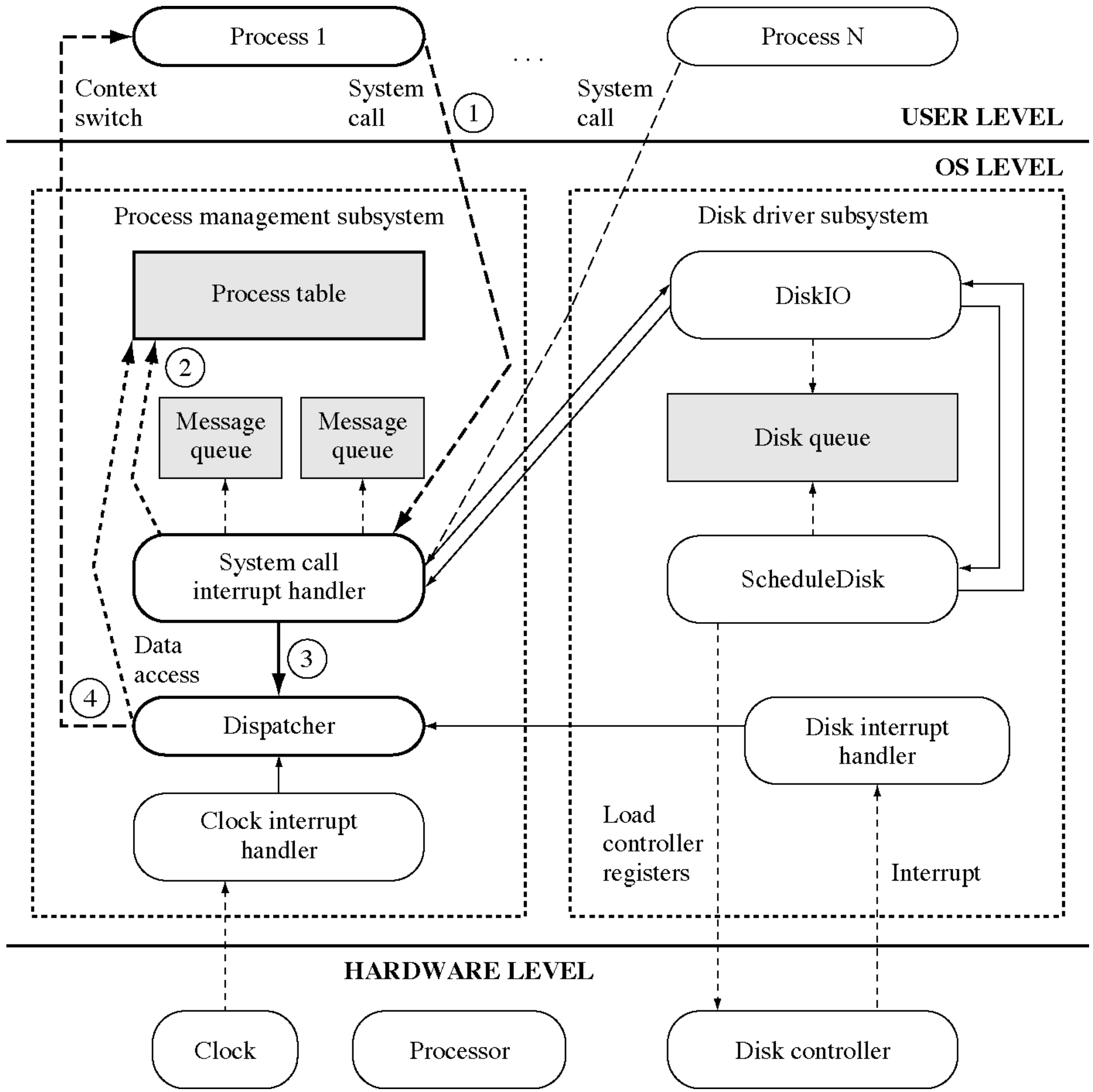
Suspending system calls

- Find a place to save the state that will be needed when the system call resumes
  - usually this is in a waiting queue
- Arrange to be resumed when the event you are waiting for occurs
  - the OS component that handles the event will also handle this duty

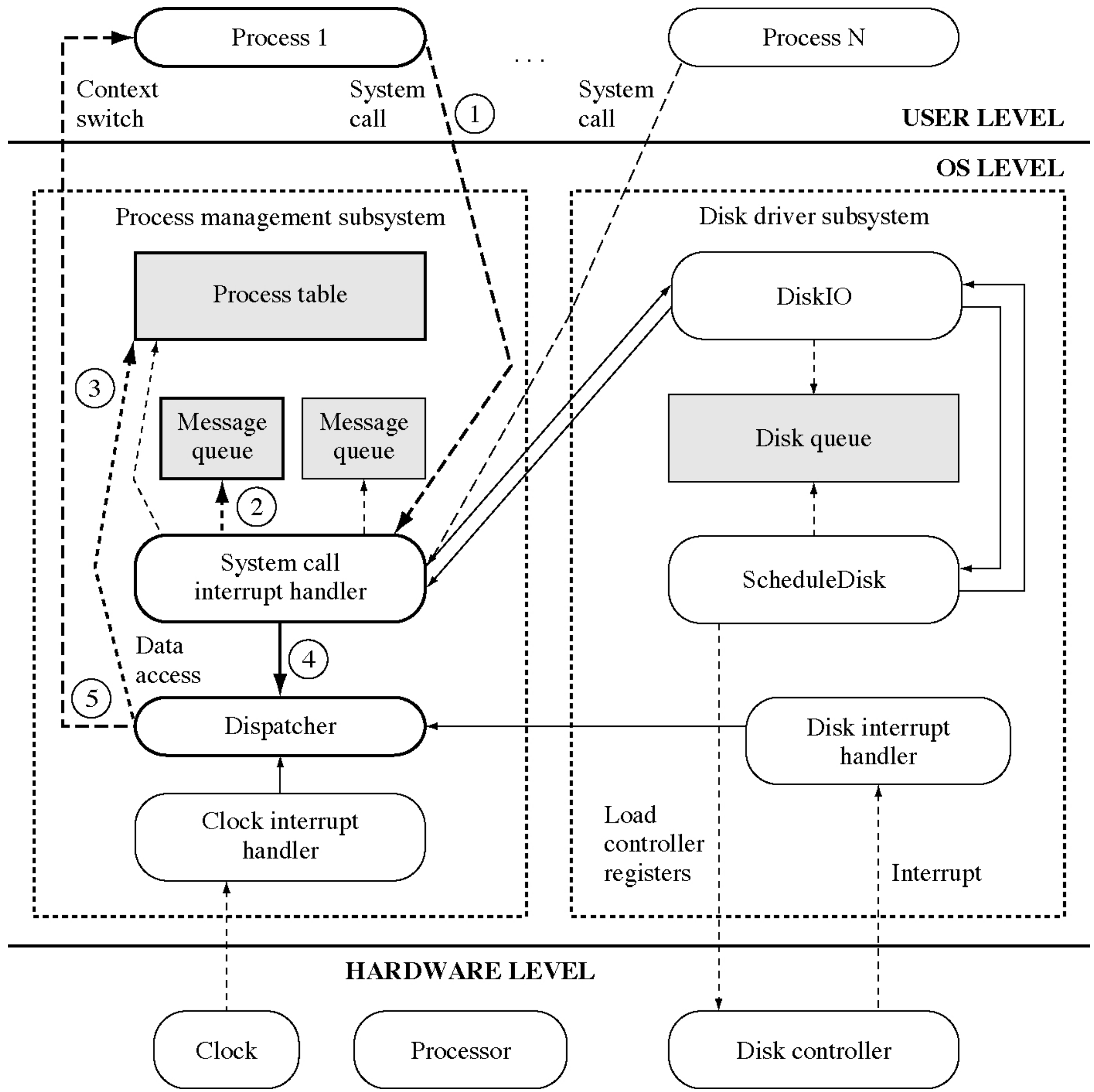
Disk read flow of control



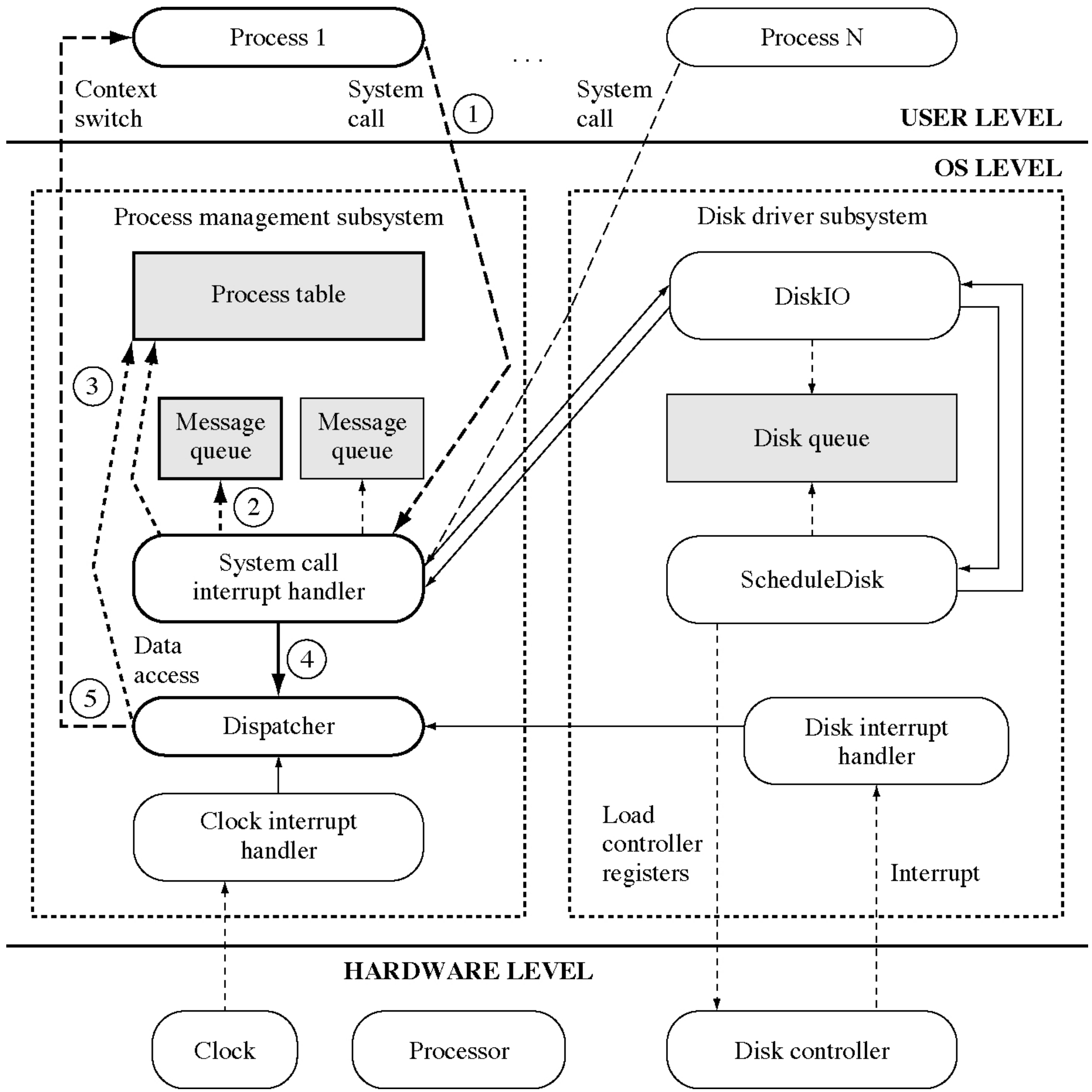
Create process flow of control



Create message queue control flow



Send message flow of control



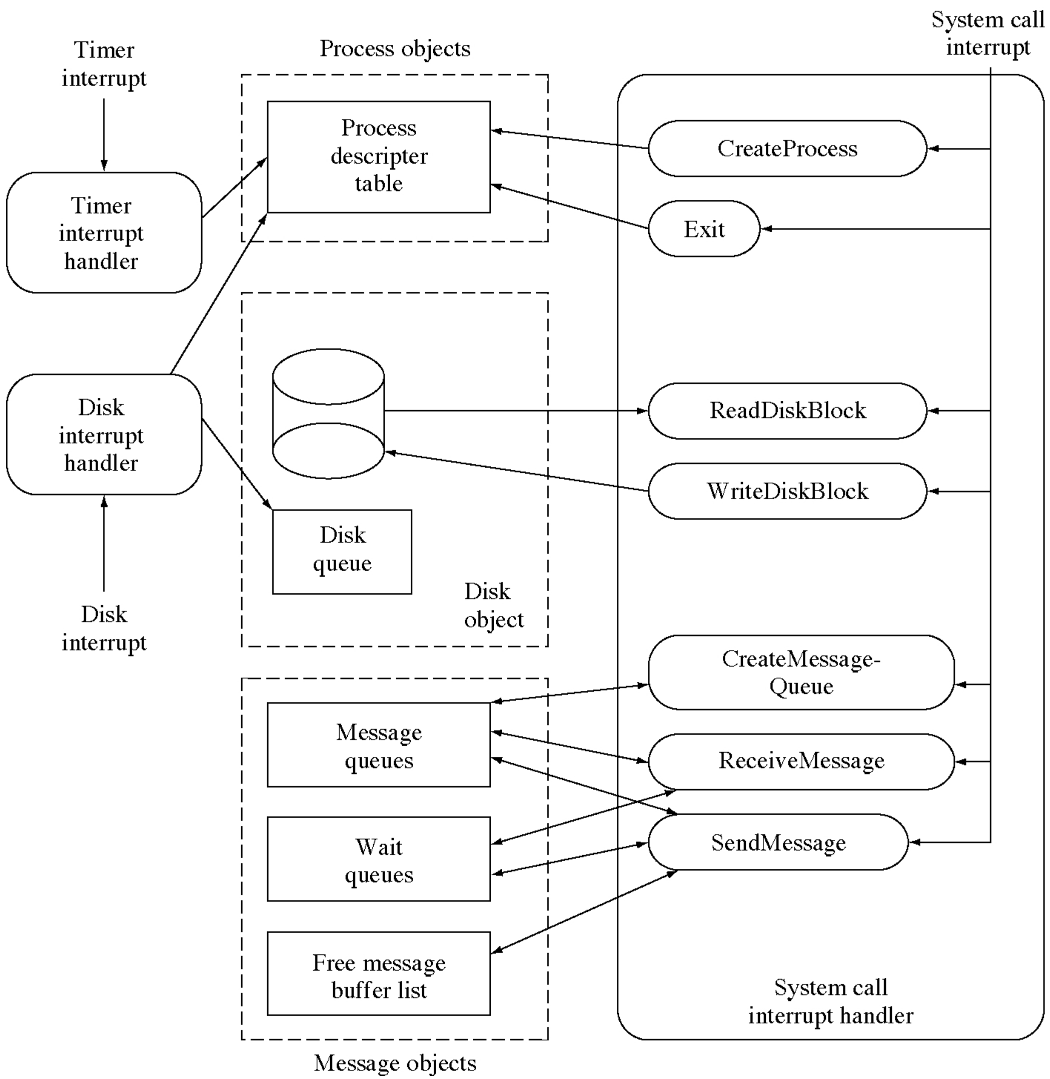
Interrupts in the OS

- The OS is not set up to handle this
  - data save areas will be destroyed
  - so we don’t allow interrupts in system code
- Chapter 6 shows how to handle this problem

The OS as an event and table manager

• Interrupt	*Data Updated*	Processing Done
– Timer	*pd*	Switch processes
– Disk	*pd*	Unblock process and start next I/O
• System Call	*Data Updated*	Processing Done
– Createprocess	*pd*	Initialize process table slot
– Exit	*pd*	Free process table slot
– CreateMsgQueue	*pd, message_queue*	Initialize message queue
– SendMessage	*pd, message_queue message_buffer*	Queue or transfer message
– ReceiveMessage	*pd, message_queue message_buffer*	Block or transfer message
– ReadDiskBlock	*pd, disk_queue*	Queue disk request
– WriteDiskBlock	*pd, disk_queue*	Queue disk request

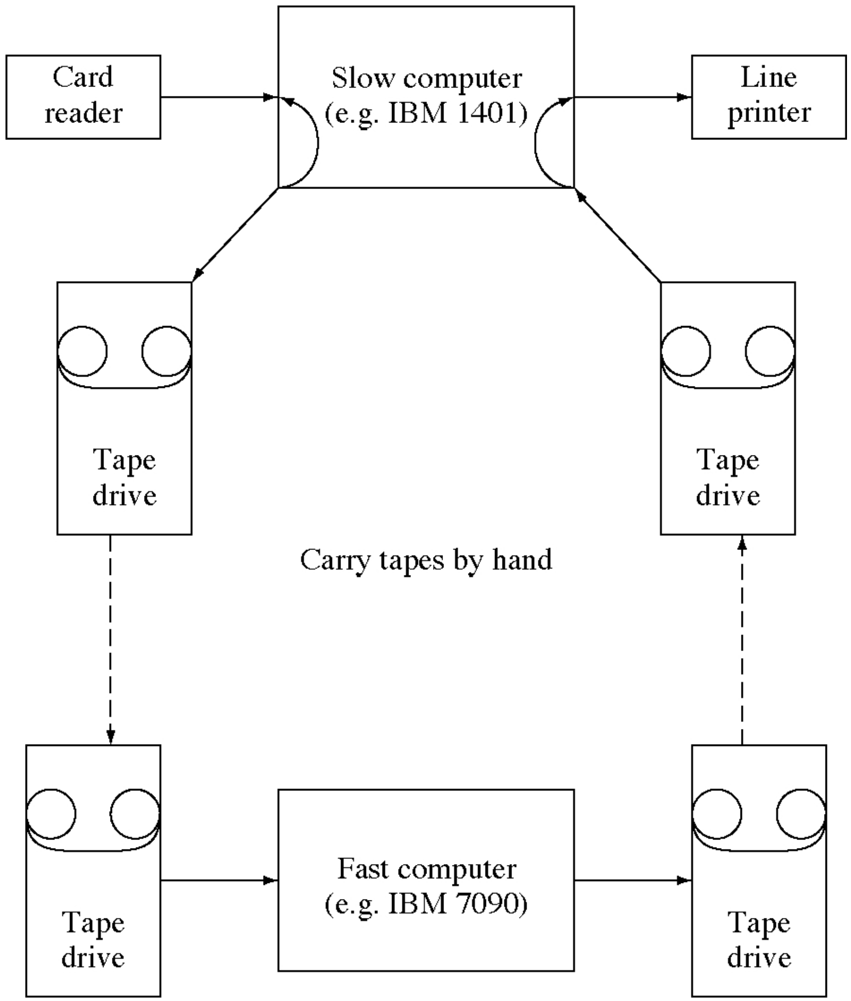
Interrupt event handling



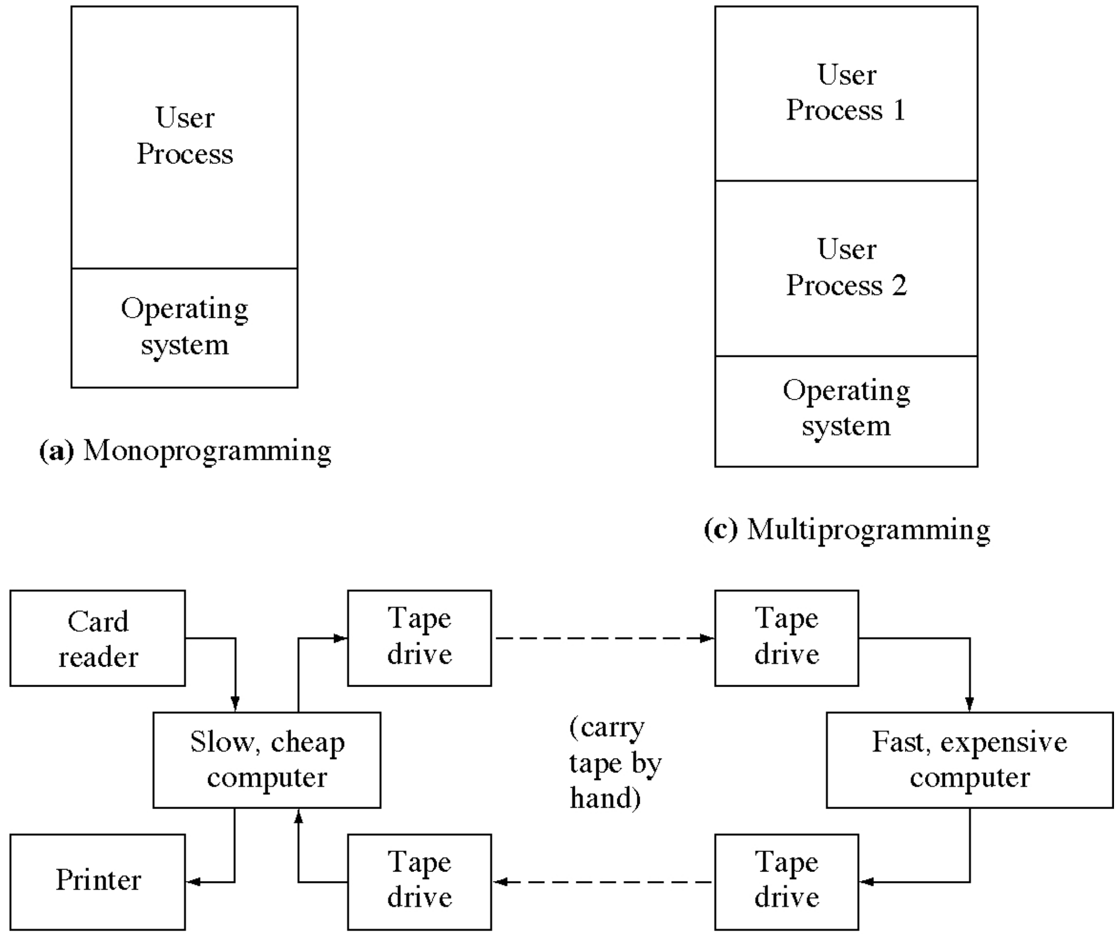
Typical process descriptor fields

- Process ID and name
- Memory allocated to the process
- Open files
- Process state
- User name and protection privileges
- Register save area
- CPU time user
- Pending software interrupts
- Parent process
- User ID

A batch operating system



Speeding up I/O (3 methods)

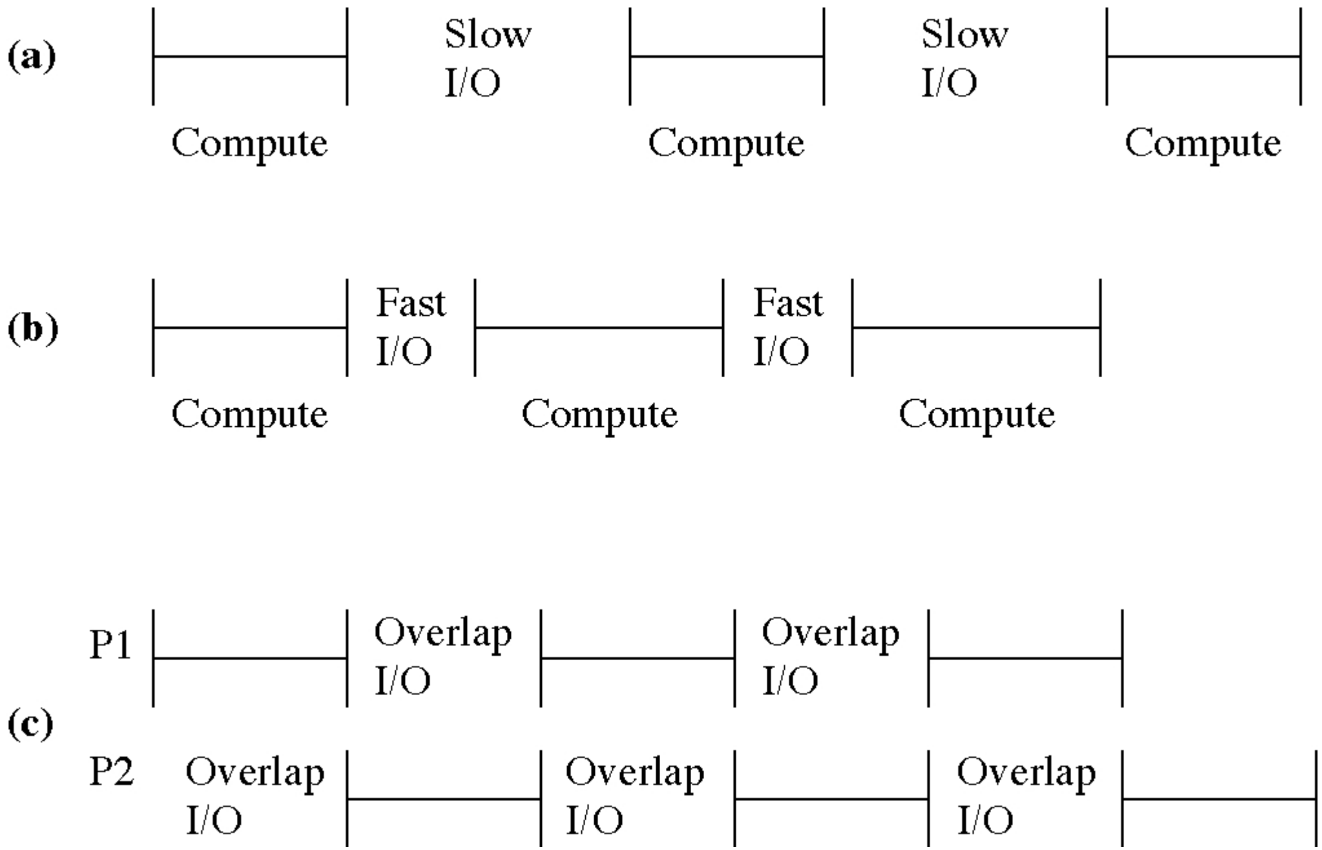


(a) Monoprogramming

(c) Multiprogramming

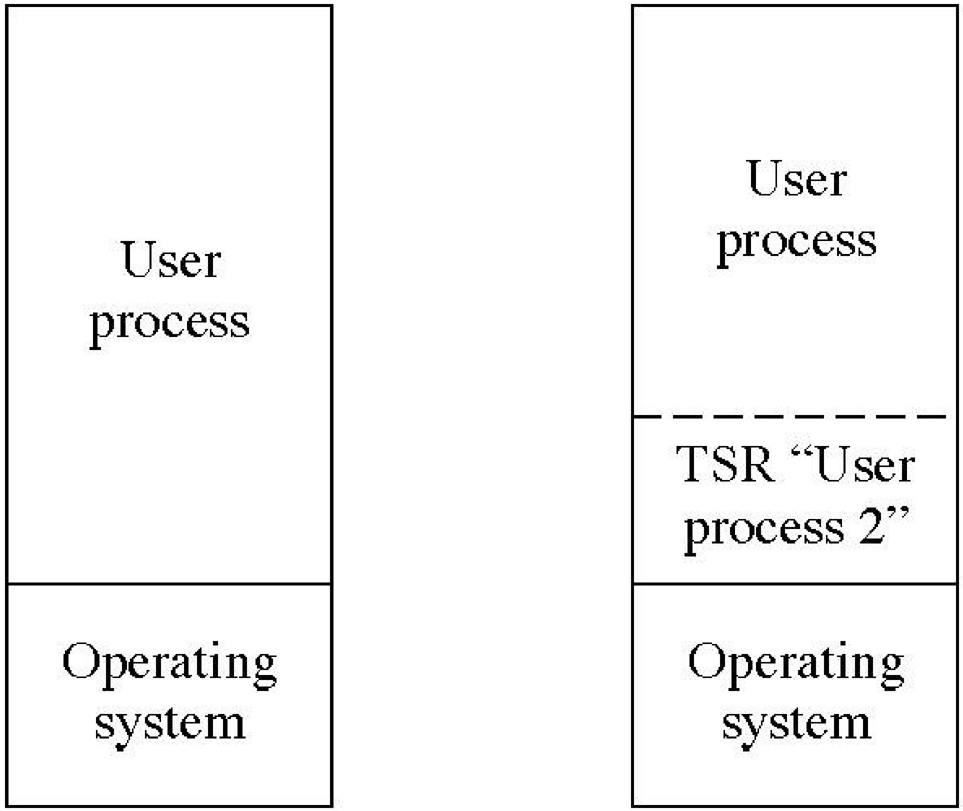
(b) Faster I/O

I/O overlap



(a)
Slow I/O
Slow I/O
Compute
Compute
Compute
(b)
Fast I/O
Fast I/O
Compute
Compute
Compute
P1
Overlap I/O
Overlap I/O
(c)
P2
Overlap I/O
Overlap I/O
Overlap I/O

TSRs in PCs



(a) PC with monoprogramming—slow task switching

(b) PC with TSRs