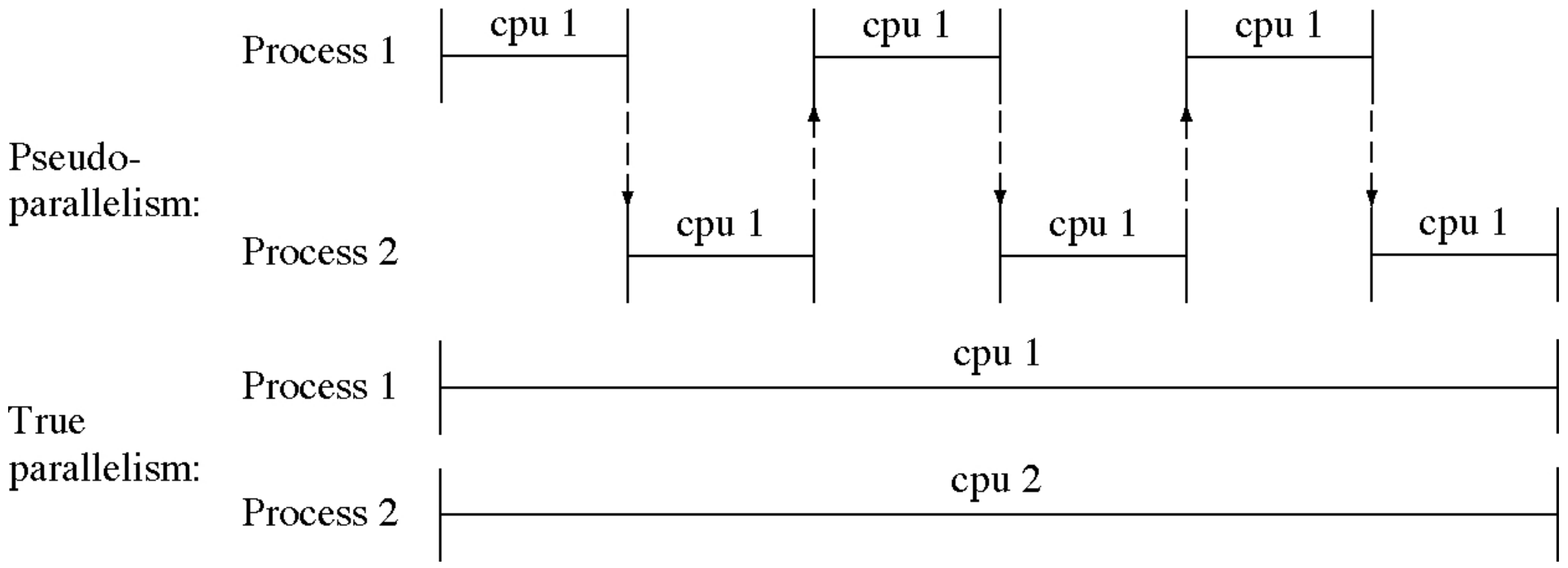
Parallel Systems

Chapter 6

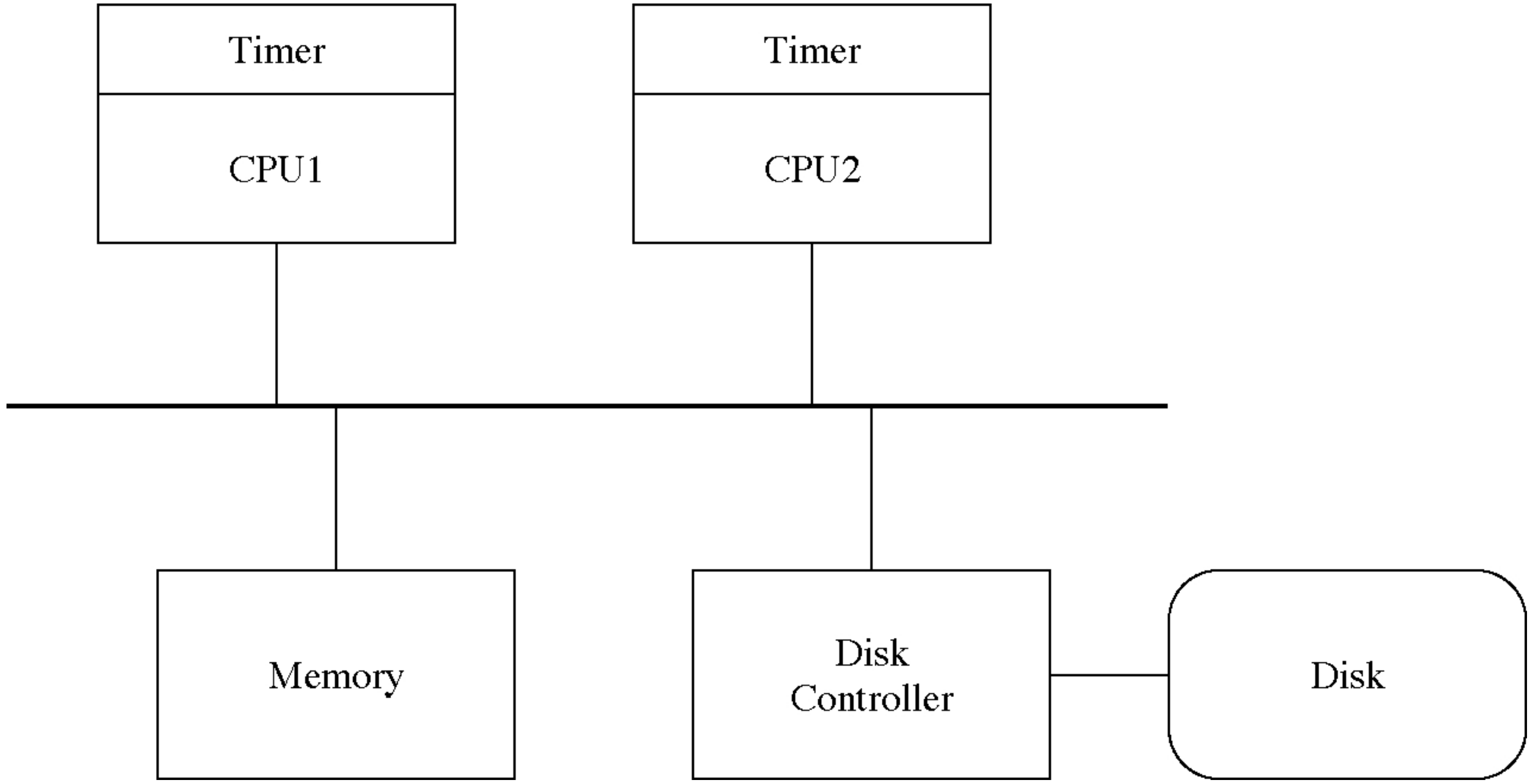
Key concepts in chapter 6

- Parallelism
  - physical (a.k.a. true) parallelism
  - logical (a.k.a. pseudo-) parallelism
- Multiprocessor OS (MPOS)
  - race conditions
  - atomic actions
  - spin locks
- Threads
- Kernel-mode processes
- Implementation of mutual exclusion

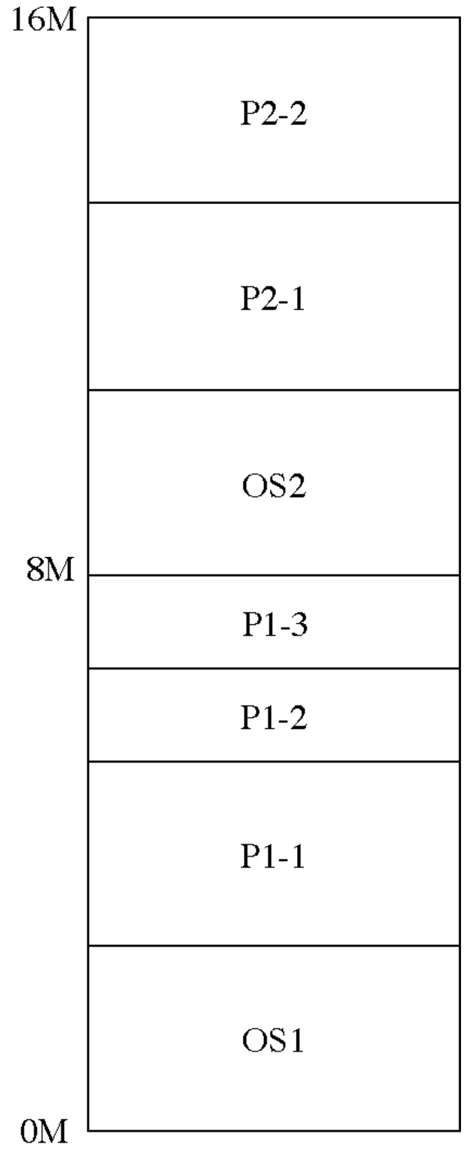
Parallel actions in a program



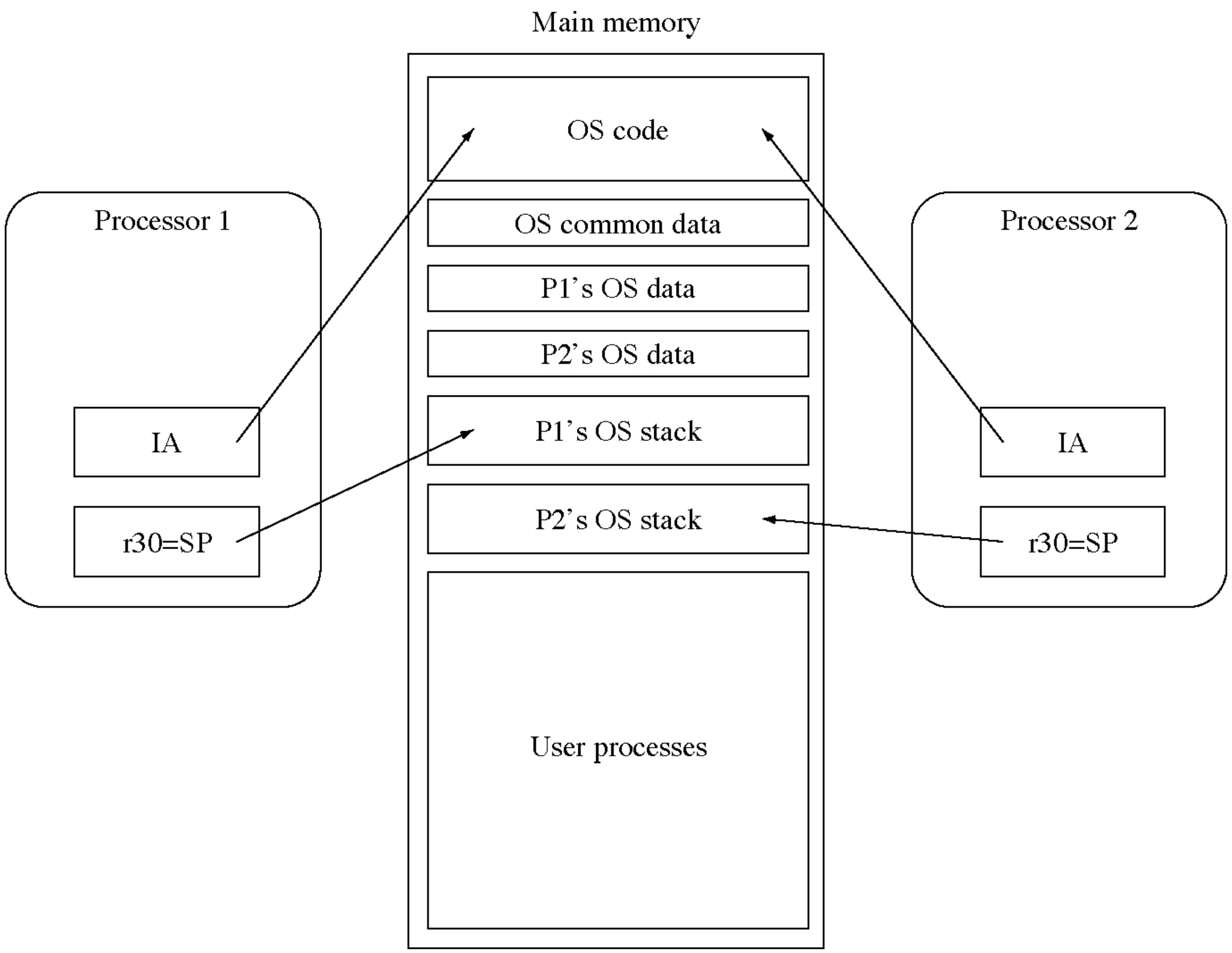
Parallel hardware



Two operating systems



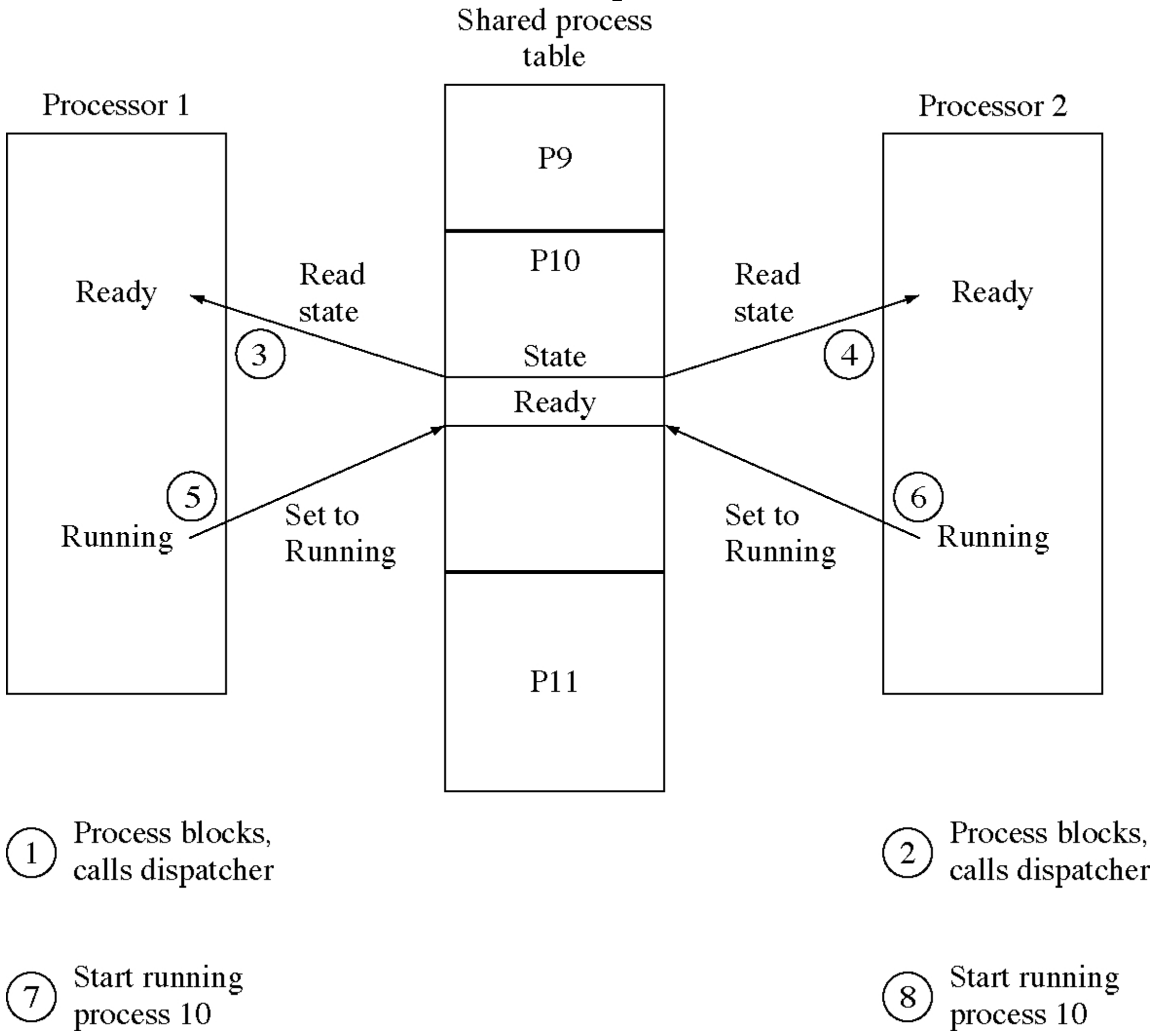
Sharing the OS between processors



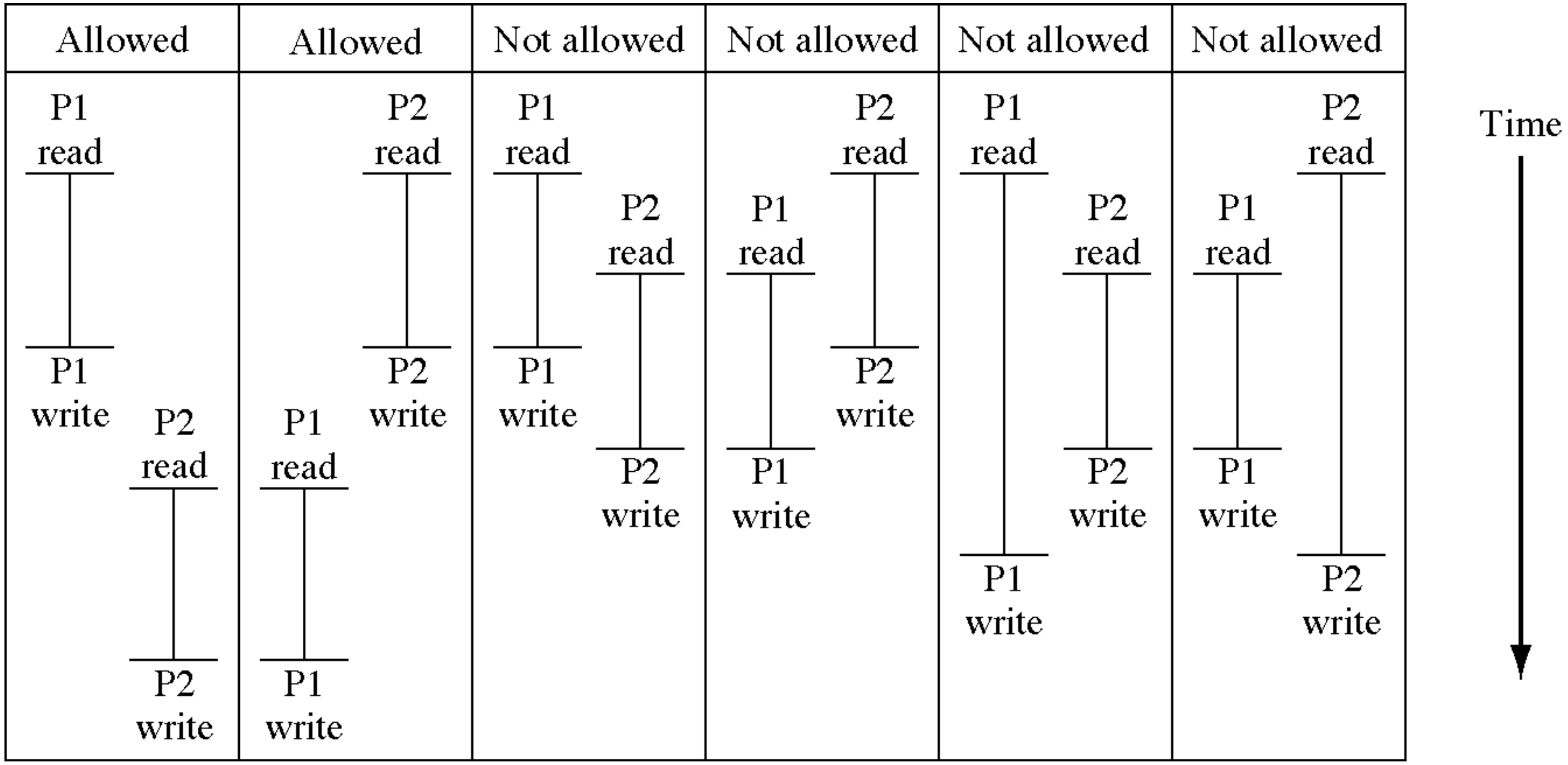
Global data

- Shared (between the processors)
  - process_using_disk
  - disk queue
  - process table (pd)
  - message queues
- Duplicated (one for each processor)
  - stack
  - current_process

Race condition with a shared process table


Shared process table
Processor 1
Processor 2
P9
P10
State
Ready
P11
Ready
Running
Read state
Set to Running
3
4
5
6
1 Process blocks, calls dispatcher
2 Process blocks, calls dispatcher
7 Start running process 10
8 Start running process 10

Atomic actions with the ExchangeWord instruction



Hardware implementation of atomic actions

- Only the “bus master” can access memory
- Solution
  - allow one processor to be the bus master for two bus cycles
  - one to read the word
  - one to write the word
  - Presto! An atomic read/write action

MPOS: global data

-
  ```
  // Each processor has a stack and a current process
  int p1_current_process;
  int p2_current_process;
  int p1_SystemStack[SystemStackSize];
  int p2_SystemStack[SystemStackSize];
  // Shared process table
  ProcessDescriptor pd[NumberOfProcesses];
  ```

-
  ```
  // Shared message queues
  MessageBuffer
  message_buffer[NumberOfMessageBuffers];
  int free_message_buffer;
  int message_queue_allocated[NumberOfMessageQueues];
  Queue<MessageBuffer *>
    * message_queue[NumberOfMessageQueues];
  Queue<WaitQueueItem *>
    * wait_queue[NumberOfMessageQueues];

  // Shared disk data
  int process_using_disk;
  Queue<DiskRequest *> disk_queue;
  ```

MPOS: system initialization

```
• int main( void ) { // Proc. 1 system initialization

    // Set up the system's stack
    asm{ load  p1_SystemStack+SystemStackSize,r30 }
    // set up the interrupt vectors ... as before
    // set up the pds ... as before
    // process 1 is the initialization process
    // ... as before
    // set up the pool of free message buffers
    // ... as before
    process_using_disk = 0;
    p1_current_process = 0;
    // start processor 2
    Dispatcher();
  }
  int main( void ) { // Proc. 2 system initialization
    // Set up the system's stack
    asm{ load  p2_SystemStack+SystemStackSize,r30 }
    p2_current_process = 0;
    Dispatcher();
  }
```

Protecting the process table

-
  ```
  // A global variable to restricts access to the
  // process table
  int ProcessTableFreeFlag = 1;
  void StartUsingProcessTable( void ) {
    int flag;
    while( 1 ) {
      asm{
        move  r0,r1
        exchangeword r1,ProcessTableFreeFlag
        store  r1,flag
      }
      // Busy wait until we get a 1 from the flag.
      if( flag == 1 )
        break;
    }
  }
  void FinishUsingProcessTable( void ) {
    ProcessTableFreeFlag = 1;
  }
  ```

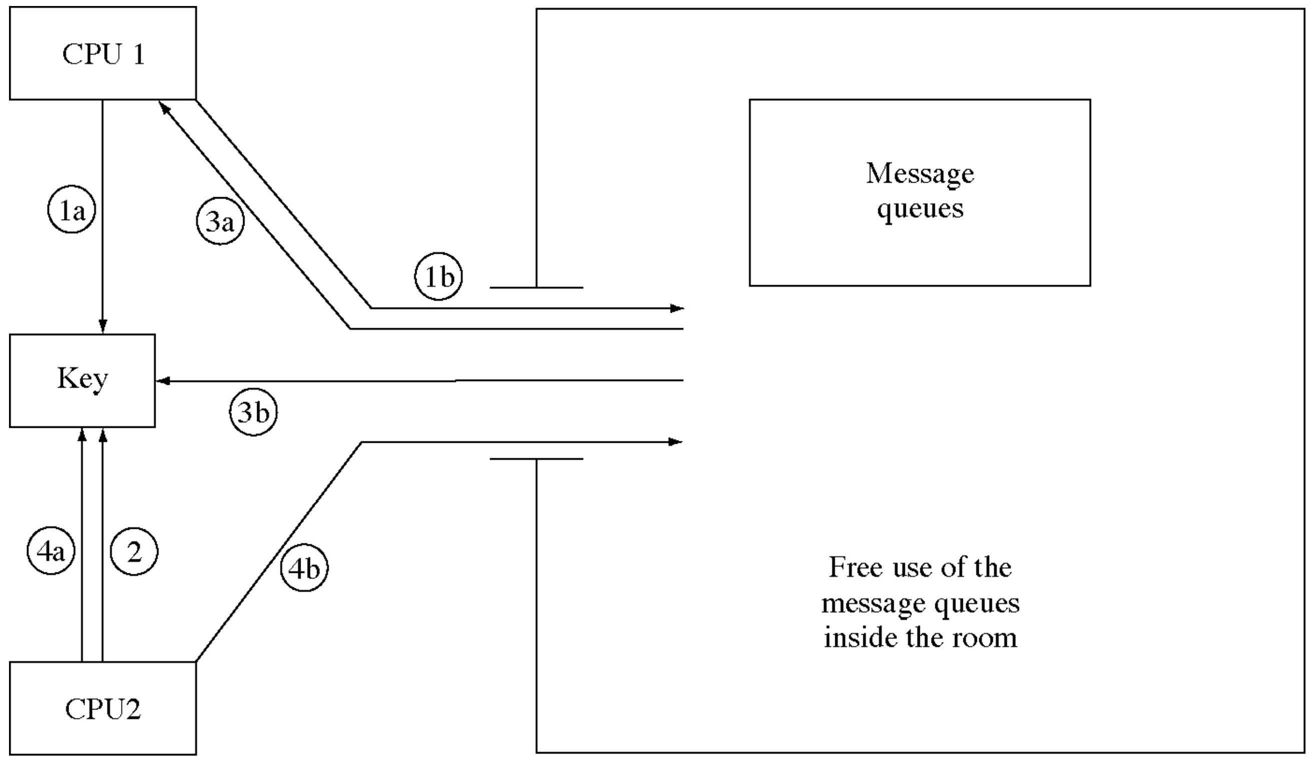
MPOS: dispatcher

-
  ```
  int SelectProcessToRun( void ) {
    static int next_proc = NumberOfProcesses;
    int i, return_value = -1; int current_process;
    StartUsingProcessTable();  // <-------- NEW CODE
    if( ProcessorExecutingThisCode() == 1 )
         current_process = p1_current_process;
    else current_process = p2_current_process;
    if(current_process>0 && pd[current_process].state==Ready
        && pd[current_process].timeLeft>0) {
      pd[current_proc].state = Running;
      return_value =  current_process;
    } else {
      for( 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].state = Running;
          pd[next_proc].timeLeft = TimeQuantum;
          return_value = next_proc;
          break;
        }
      }
    }
    FinishUsingProcessTable();  // <-------- NEW CODE
    return return_value;
  }
  ```

Busy waiting

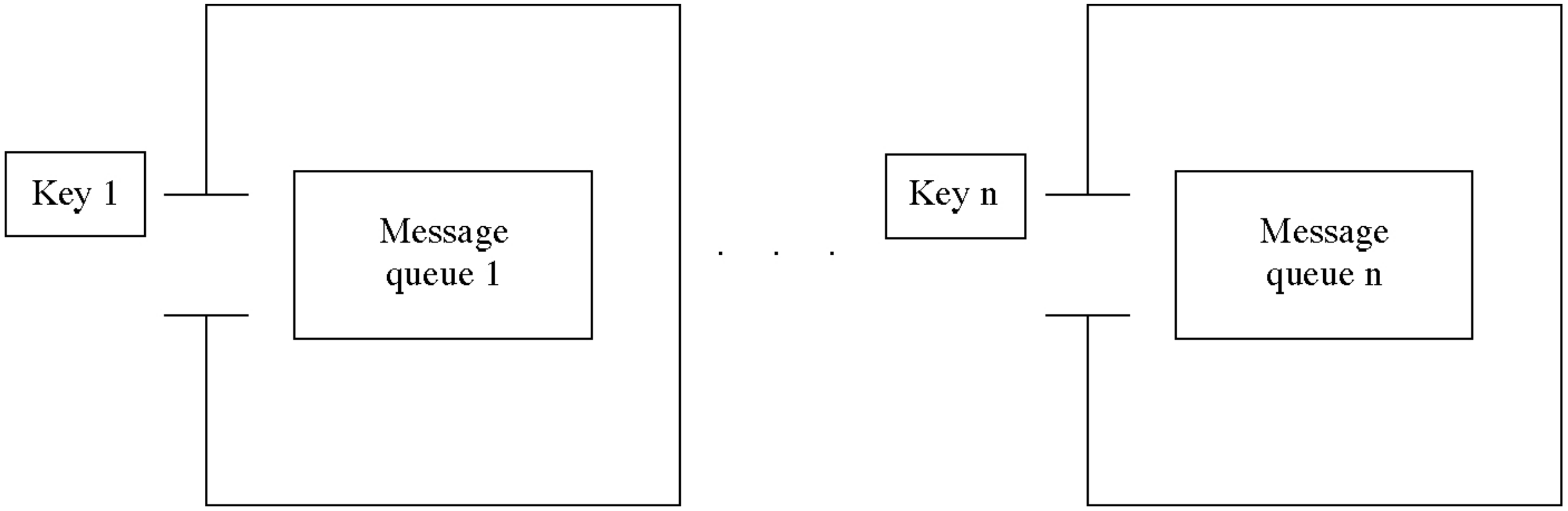
- *Processes* wait by being suspended and resumed when the desired event occurs
- *Processors* wait by continually checking the lock word
  - this is called busy waiting
  - the lock is called a spin lock

Protecting all the message queues with one lock



1. CPU1 (a) gets key and (b) enters the room.
2. CPU2 fails to get the key but keeps on trying.
3. CPU1 (a) leaves the room and (b) returns the key.
4. CPU2 (a) succeeds in getting the key and (b) enters the room.

Protecting each message queue with a separate lock



Protecting access to shared data

-
  ```
  void StartUsingSharedMemory(
          int * shared_memory_flag ) {
      int flag;
      while( 1 ) {
          // check first to see if we have a chance
          while( * shared_memory_flag == 0 )
              ;
          asm{
              move  r0,r1
              exchangeword r1,* shared_memory_flag
              store  r1,flag
          }
          // Busy wait until we get a 1 from the flag.
          if( flag == 1 ) break;
      }
  }
  void FinishUsingSharedMemory(
          int * shared_memory_flag ) {
      *shared_memory_flag = 1;
  }
  ```

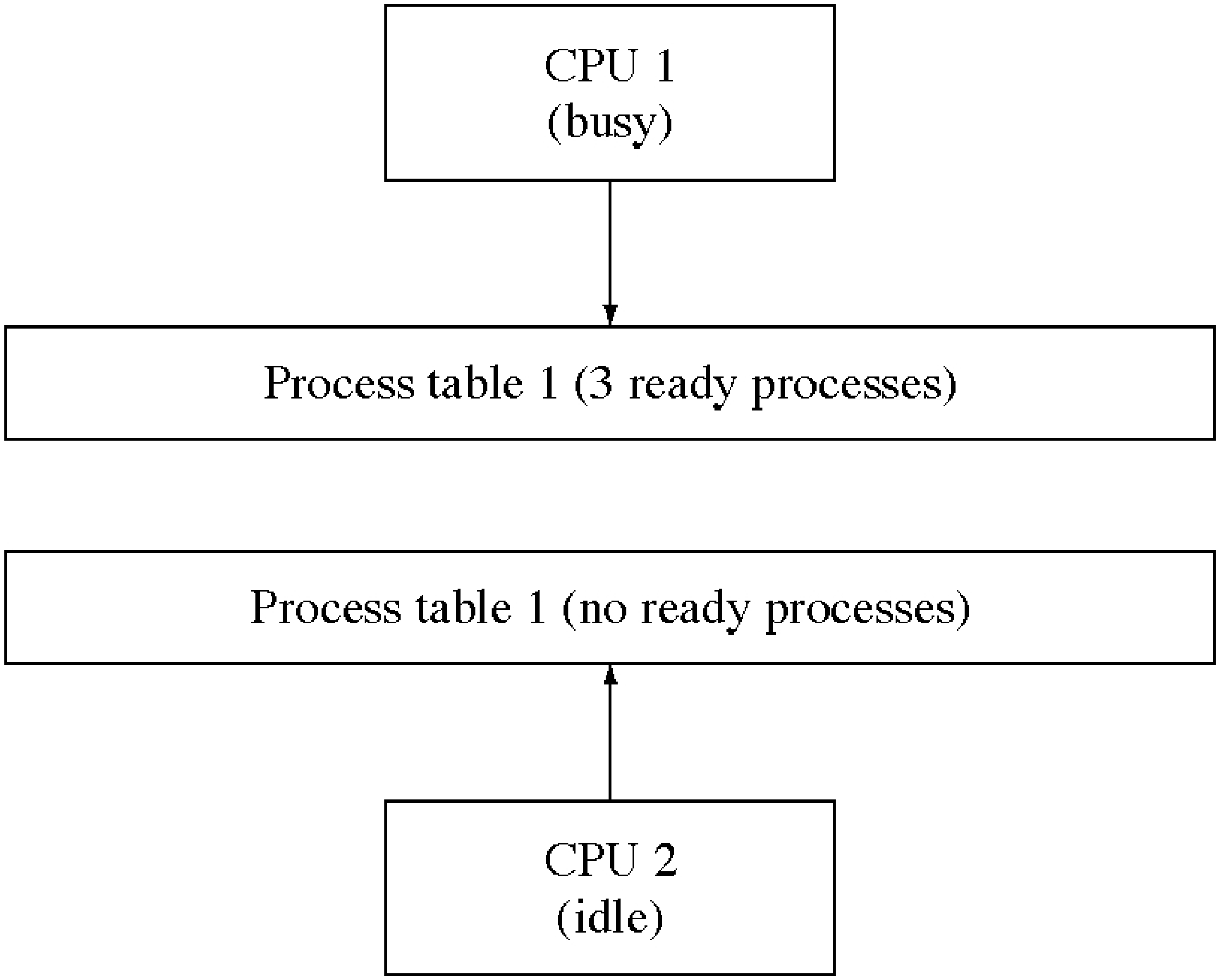
Protecting the message queues

-
  ```
  int message_queue_flag[NumberOfMessageQueues]
    = {1,1,1,...,1};
  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 )
    StartUsingSharedMemory(&message_queue_flag[to_q]);
    if( !wait_queue[to_q].Empty() ) {
      WaitQueueItem item = wait_queue.Remove();
      TransferMessage( msg_no, item.buffer );
      pd[item.pid].state = Ready;
    } else { message_queue[to_q].Insert( msg_no ); }
    FinishUsingSharedMemory(&message_queue_flag[to_q]);
    pd[current_process].sa.reg[1] = 0;
    break;
  ```

Protecting the message buffers

-
  ```
  int message_buffer_flag = 1; //

  int GetMessageBuffer( void ) {
    // get the head of the free list
    StartUsingSharedMemory( &message_buffer_flag );
    int msg_no = free_msg_buffer;
    if( msg_no != EndOfFreeList ) {
      // follow the link to the next buffer
      free_msg_buffer = message_buffer[msg_no][0];
    }
    FinishUsingSharedMemory( &message_buffer_flag );
    return msg_no;
  }

  void FreeMessageBuffer( int msg_no ) {
    StartUsingSharedMemory( &message_buffer_flag );
    message_buffer[msg_no][0] = free_msg_buffer;
    free_msg_buffer = msg_no;
    FinishUsingSharedMemory( &message_buffer_flag );
  }
  ```

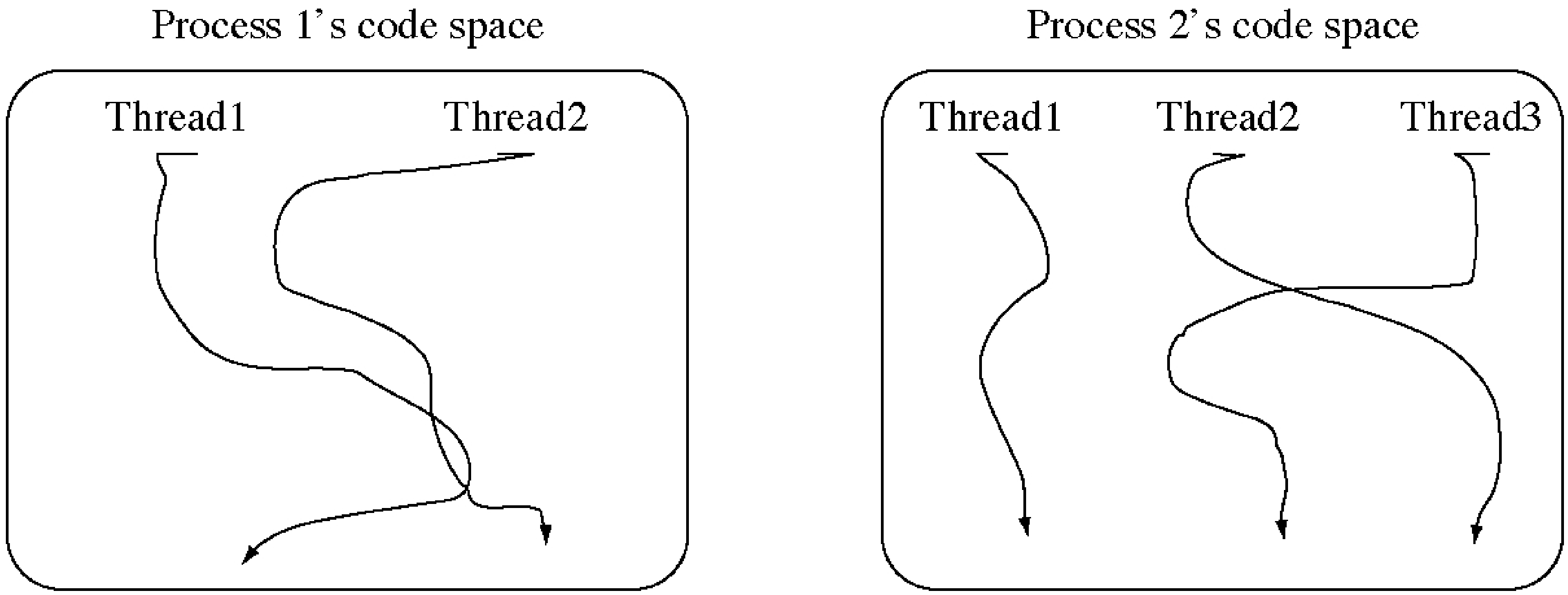
Using two process tables


CPU 1
(busy)
Process table 1 (3 ready processes)
Process table 1 (no ready processes)
CPU 2
(idle)

Threads

- A **process** has two characteristics
  - *resources*: a process can hold resources such as an address space and open files.
  - *dispatchability*: a process can be scheduled by the dispatcher and execute instructions.
- We can split these two characteristics into:
  - a (new) process which holds resources
  - a **thread** which can be dispatched and execute instructions.
  - several threads can exist in the same process, that is, in the same address space.

Threads executing in two processes



Process and thread analogies

- A process is the software version of a standalone computer
  - One processor (for each computer or process), no shared memory
  - Communicate (between computers or processes) with messages
- A thread is the software version of a processor in a multiple-processor, shared-memory computer
  - Multiple processors (threads), shared memory
  - Communicate (between processors or threads) through shared memory

Thread-related system calls

- int CreateThread(
    char *startAddress, char *stackAddress)
- int ExitThread(int returnCode)
- WaitThread(int tid)
- Basically the same as for processes

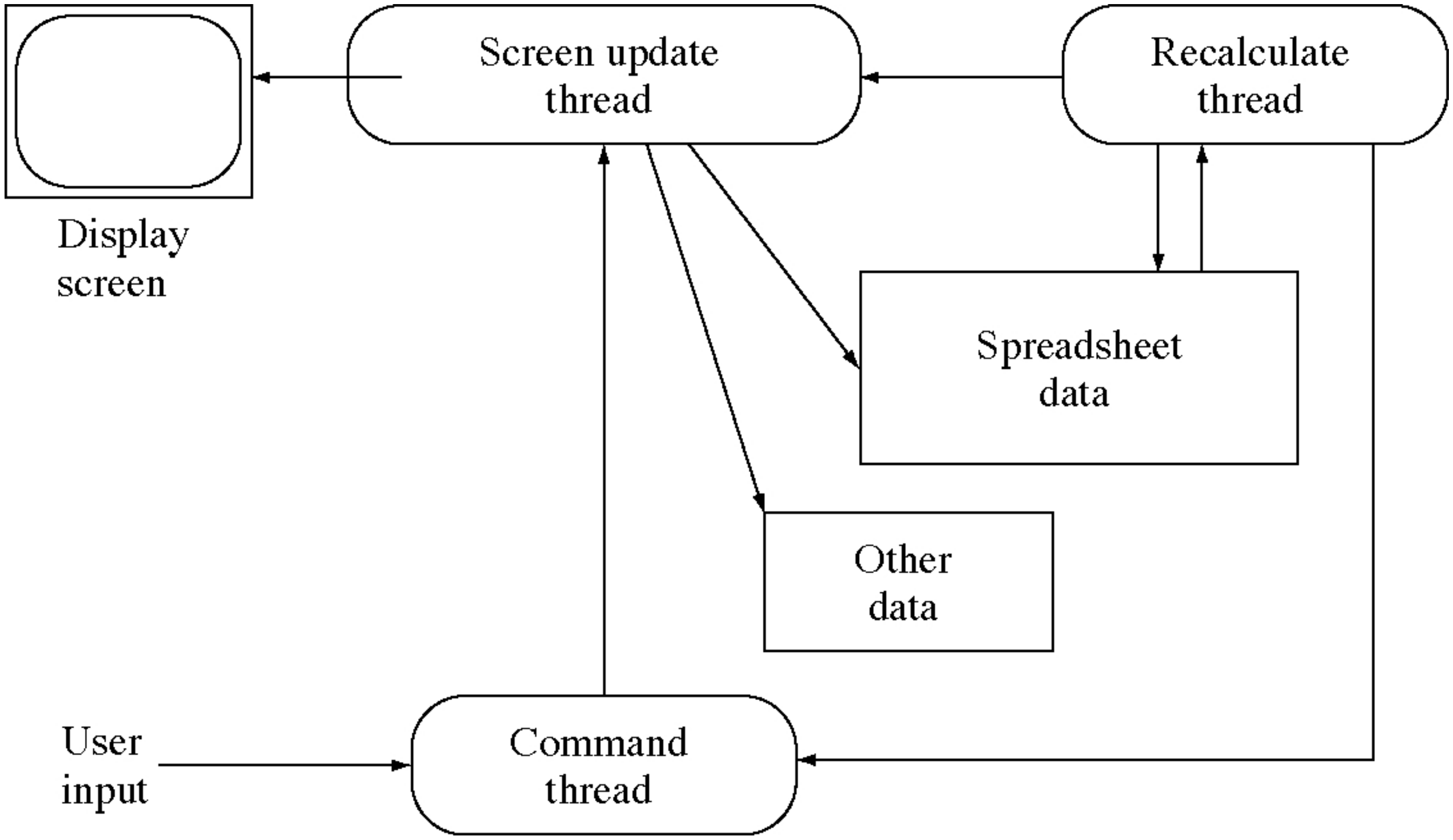
Advantages of threads

- Threads allows parallel activity inside a single address space
  - Inter-thread communication and synchronization is very fast
- Threads are cheap to create
  - mainly because we do not need another address space
  - They are also called *lightweight processes* (LWPs)

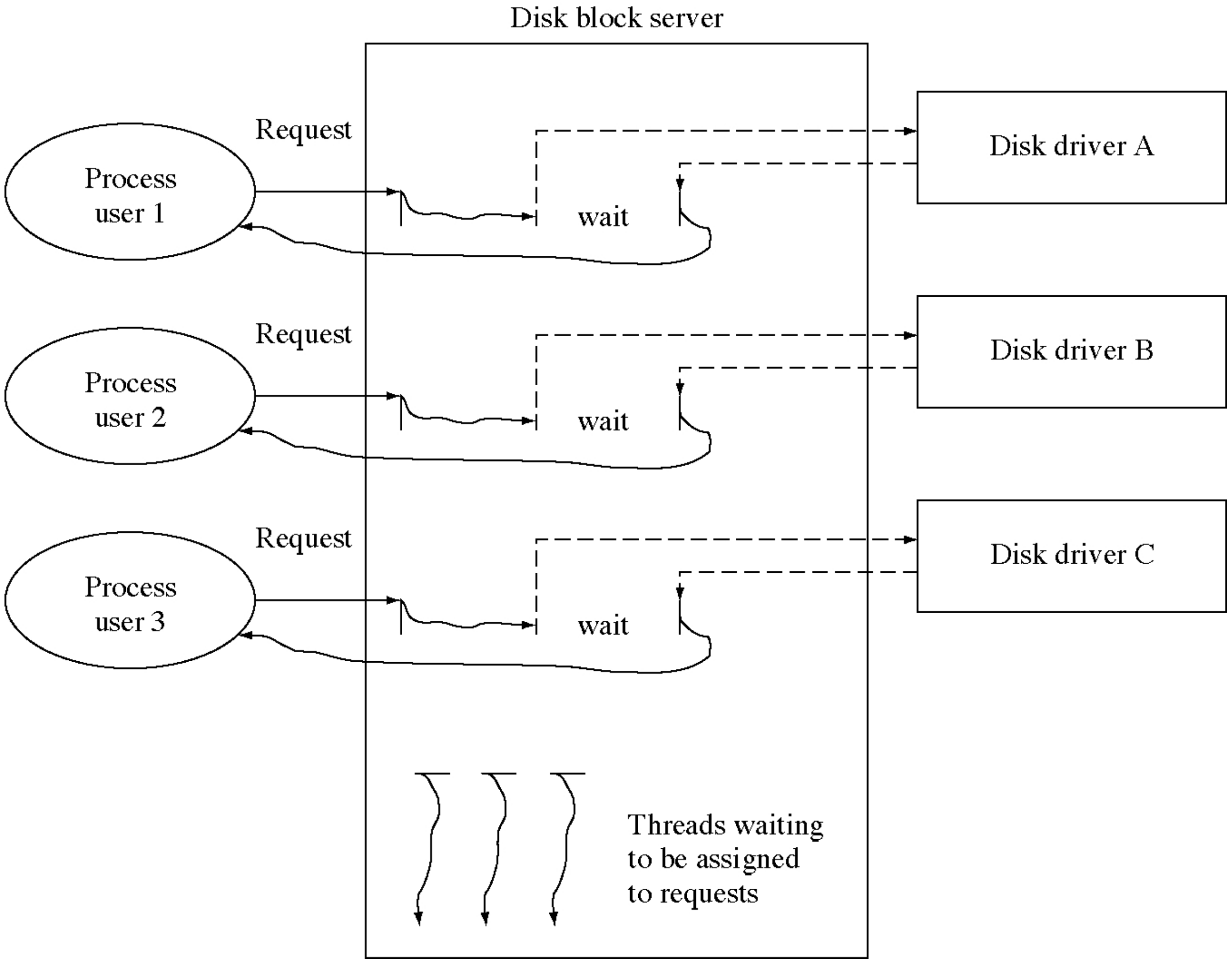
Uses of threads

- Partition process activities
  - each part of the process has its own thread
- Waiting
  - threads can wait for events without holding up other threads
- Replicate activities
  - servers can allocate a thread for each request

Threads in a spreadsheet program



Disk server using threads



Threads: global data

-
  ```
  enum ThreadState { Ready, Running, Blocked };
  struct ThreadDescriptor {
    int slotAllocated;
    int timeLeft;
    int nextThread, prevThread;
    int pid;
    ThreadState state;
    SaveArea sa;
  };
  struct ProcessDescriptor {
    int slotAllocated;
    int threads; void * base, bound;
  };
  // This is the thread currently running.
  int current_thread;
  // We need a process table and a thread table.
  ProcessDescriptor pd[NumberOfProcesses];
  // pd[0] is the systemThreadDescriptor
  td[NumberOfThreads];
  int thread_using_disk;
  ```

Threads: system initialization

-
  ```
  int main( void ) {
    // Other initialization is the same …
    // The thread slots start out free.
    for( i = 1; i < NumberOfThreads; ++i )
      td[i].slotAllocated = False;
    // Other initialization is the same …
  }
  ```

Threads: create process

-
  ```
  int CreateProcess( 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].base = pid * ProcessSize;
    pd[pid].bound = ProcessSize;
    char * addr = (char *)(pd[pid].sa.base);
    for( i = 0; i < n_blocks; ++i ) {
      while( DiskBusy() ) ;    // Busy wait for I/O.
      IssueDiskRead( first_block + i, addr, 0 );
      addr += DiskBlockSize;
    }
    int tid = CreateThread( pid, 0, ProcessSize );
    pd[pid].threads = tid;
    if( tid == -1 ) return -1;
    td[tid].nextThread = td[tid].prevThread = tid;
    return pid;
  }
  ```

Threads: create thread

-
  ```
  int CreateThread(
    int pid, char * startAddress, char *stackBegin ) {
    // startAddress = address in the process
    //   address space to begin execution.
    // stackBegin = initial value of r30
    int tid;
    for( tid = 1; tid < NumberOfThreads; ++tid )
      if( !(td[tid].slotAllocated) ) break;
    if( tid >= NumberOfThreads ) {
      return -1; }
    td[tid].slotAllocated = True;
    td[tid].pid = pid;
    td[tid].state = Ready;
    td[tid].sa.base = pd[pid].base;
    td[tid].sa.bound = pd[pid].bound;
    td[tid].sa.psw = 3;
    td[tid].sa.ia = startAddress;
    td[tid].sa.r30 = stackBegin;
    return tid;
  }
  ```

Threads: dispatcher

-
  ```
  void Dispatcher( void ) {
    current_thread = SelectThreadToRun();
    RunThread( current_thread ); }
  int SelectThreadToRun( void ) {
    static int next_thread = NumberOfThreads;
    if( current_thread > 0 && td[current_thread].slotAllocated
        && td[current_thread].state == Ready
        && td[current_thread].timeLeft > 0 ) {
      td[current_thread].state = Running;
      return current_thread; }
    for( int i = 0; i < NumberOfThreads; ++i ) {
      if( ++next_thread >= NumberOfThreads ) next_thread = 0;
      if( td[next_thread].slotAllocated
          && td[next_thread].state == Ready ) {
        td[next_thread].state = Running;
        return next_thread;
      }
    }
    return -1;    // No thread is ready to run
  }
  void RunThread( int tid ) {
    /* ... same except it uses tid instead of pid */}
  ```

Threads: system calls (1 of 2)

-
  ```
  void SystemCallInterruptHandler( void ) {
    // The Exit system call is eliminated.  You exit a
    // process by exiting from all its threads.
    case CreateProcessSystemCall:
      int block_number; asm { store r9,block_number }
      int number_of_blocks;
        asm { store r10, number_of_blocks }
      td[current_thread].sa.reg[1]
        = CreateProcess( block_number, number_of_blocks);
      break;
    case CreateThreadSystemCall:
      int start_addr; asm { store r9,start_addr }
      int stack_begin; asm { store r10,stack_begin }
      int pid = td[current_thread].pid;
      int new_thread
        = CreateThread( pid, start_addr, stack_begin );
      td[current_thread].sa.reg[1] = new_thread;
      int next_thread = td[current_thread].nextThread;
      td[new_thread].nextThread = next_thread;
      td[new_thread].prevThread = current_thread;
      td[current_thread].nextThread = new_thread;
      td[next_thread].prevThread = new_thread;
      break;
  ```

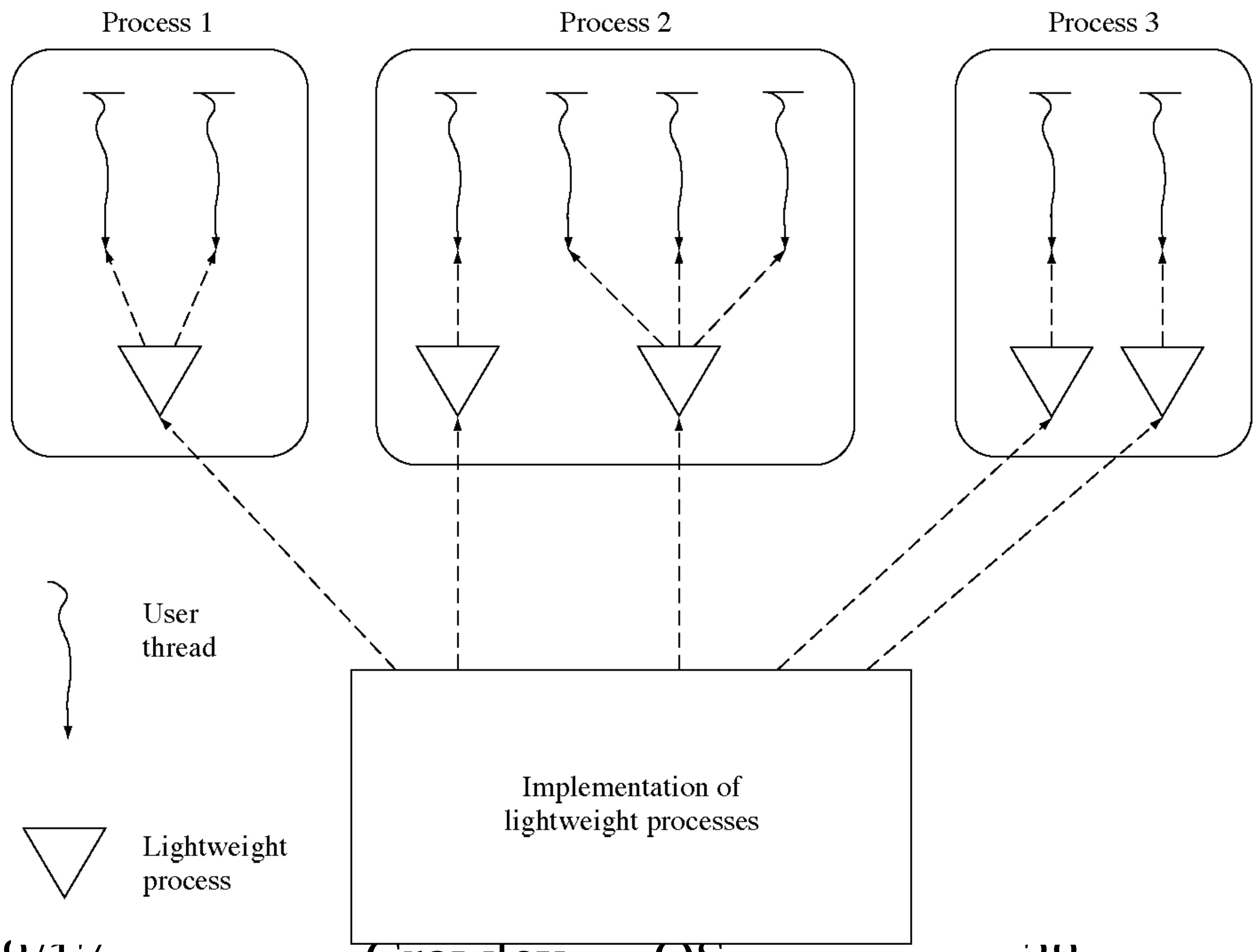
Threads: system calls (2 of 2)

```
•   case ExitThreadSystemCall:
      td[current_thread].slotAllocated = False;
      int next_thread = td[current_thread].nextThread;
      int pid = td[current_thread].pid;
      if( next_thread == current_thread ) {
        pd[pid].slotAllocated = False;
        // No other exit process processing. In a real OS
        // we might free memory, close unclosed files,etc.
      } else {
        // Unlink it from the list and make sure pd[pid]
        // is not pointing to the deleted thread.
        int prev_thread = td[current_thread].prevThread;
        td[next_thread].prevThread = prev_thread;
        td[prev_thread].nextThread = next_thread;
        pd[pid].threads = next_thread;
      }
      break;
    }
    Dispatcher();
  }
```

Kinds of threads

- We have described *lightweight processes*
  - threads implemented by the OS
- A process can implement *user threads*
  - threads implemented (solely) by the user
  - these are more efficient
  - but if the OS blocks one of them, they are all blocked
- *Kernel threads* are threads that run inside the OS (a.k.a. the kernel)

Combining user threads and lightweight processes



Threads in real OSs

- Almost all modern OSs implement threads
- Plan 9 has a very flexible form of process creation instead of threads
- Several user thread packages are available
- Some programming languages implement threads in the language
  - Ada, Modula 3, Java

Design technique: Separation of concepts

- We separated resource holding and dispatchability into separate concepts: process and thread
  - This allowed us to have several threads inside a single process
- If two concepts can be used separately it is often useful to separate them in the software

Design technique: Reentrant programs

- Threads allow two threads of control to be executing in the same code at the same time
- This leads to sharing problems
  - such code must be reentrant
  - this is the same problem we had with two processor sharing OS global data

Kernel-mode processes

- Many UNIX implementations have long allowed processes to execute inside the kernel (the OS) during system calls
  – this is actually an example of kernel threads
  – it neatly solves the problem of suspending a system call

Kernel-mode: globals

-
  ```
  struct ProcessDescriptor {
  int slotAllocated;
  int timeLeft; // time left from the last time slice
  ProcessState state;
  // SaveArea sa; <---- ELIMINATED
  int inSystem; // set to 0 in CreateProcess <--- NEW
  int * lastsa; // most recent save area on the stack
                // <-- NEW
  char sstack[SystemStackSize]; // system mode stack
                                //<---- NEW};
  // We don't need a common system stack any more
  // int SystemStack[SystemStackSize]; <----
  ELIMINATED
  // CHANGED: this is now a queue of ints.
  Queue<int> * wait_queue[NumberOfMessageQueues];
  ```

Kernel-mode: create process

-
  ```
  int CreateProcessSysProc(
      int first_block, int n_blocks ) {
    // ... the beginning part is the same as it was
    // before except this is added:
    pd[pid].inSystem = 0;
    // Read in the image of the process.
    char * addr = (char *)(pd[pid].sa.base);
    for( i = 0; i < n_blocks; ++i ) {
      DiskIO(DiskReadSystemCall, first_block+i, addr);
      addr += DiskBlockSize;
    }
  return pid;
  }
  ```

System calls (1 of 5)

-
  ```
  void SystemCallInterruptHandler( void ) {
    // *** BEGIN NEW CODE ***
    //     All this initial interrupt handling is new.
    int * savearea;
    int saveTimer;
    if( pd[current_process].inSystem == 0 ) {
      // This is the first entry into system mode
      savearea = &(pd[current_process].sstack
                  +SystemStackSize-sizeof(SaveArea)-4);
    } else {
      // we were already in system mode so the system
      // stack already has some things on it.
      asm { store r30,savearea }
      // allocate space on the stack for the save area
      savearea -= sizeof(SaveArea)+4;
    }
  ```

System calls (2 of 5)

```
•   asm {
      store    timer,saveTimer
      load     #0,timer
      store    iia,savearea+4
      store    ipsw,savearea+8
      store    base,savearea+12
      store    bound,savearea+16
      storeall savearea+20
      load     savearea,r30
    }
    pd[current_process].timeLeft = saveTimer;
    pd[current_process].state = Ready;
    *savearea = pd[current_process].lastsa;
    // link to previous save area
    pd[current_process].lastsa = savearea;
    ++(pd[current_process].inSystem);
    // state saved, so enable interrupts
    asm { load #2,psw }
    // *** END NEW CODE *** for interrupt handling
  code
```

System calls (3 of 5)

-
  ```
  // fetch the system call number and switch on it
  int system_call_number;
  asm { store r8,system_call_number }
  switch( system_call_number ) {
  case CreateProcessSystemCall:
    // ... the same
  case ExitProcessSystemCall:
    // ... the same
  case CreateMessageQueueSystemCall:
    // ... the same
  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;
    }
  ```

System calls (4 of 5)

```
•     int msg_no = GetMessageBuffer();
      // make sure we are not out of message buffers
      if( msg_no == EndOfFreeList ) {
        pd[current_process].sa.reg[1] = -2;
        break;
      }
      // copy message vector from the system caller's
      // memory into the system's message buffer
      CopyToSystemSpace( current_process, user_msg,
        message_buffer[msg_no], MessageSize );
      if( !wait_queue[to_q].Empty() ) {
        // process is waiting for a message, unblock
  it
        int pid = wait_queue.Remove();
        pd[pid].state = Ready;
      }
      // put it on the queue
      message_queue[to_q].Insert( msg_no );
      pd[current_process].sa.reg[1] = 0;
      break;
```

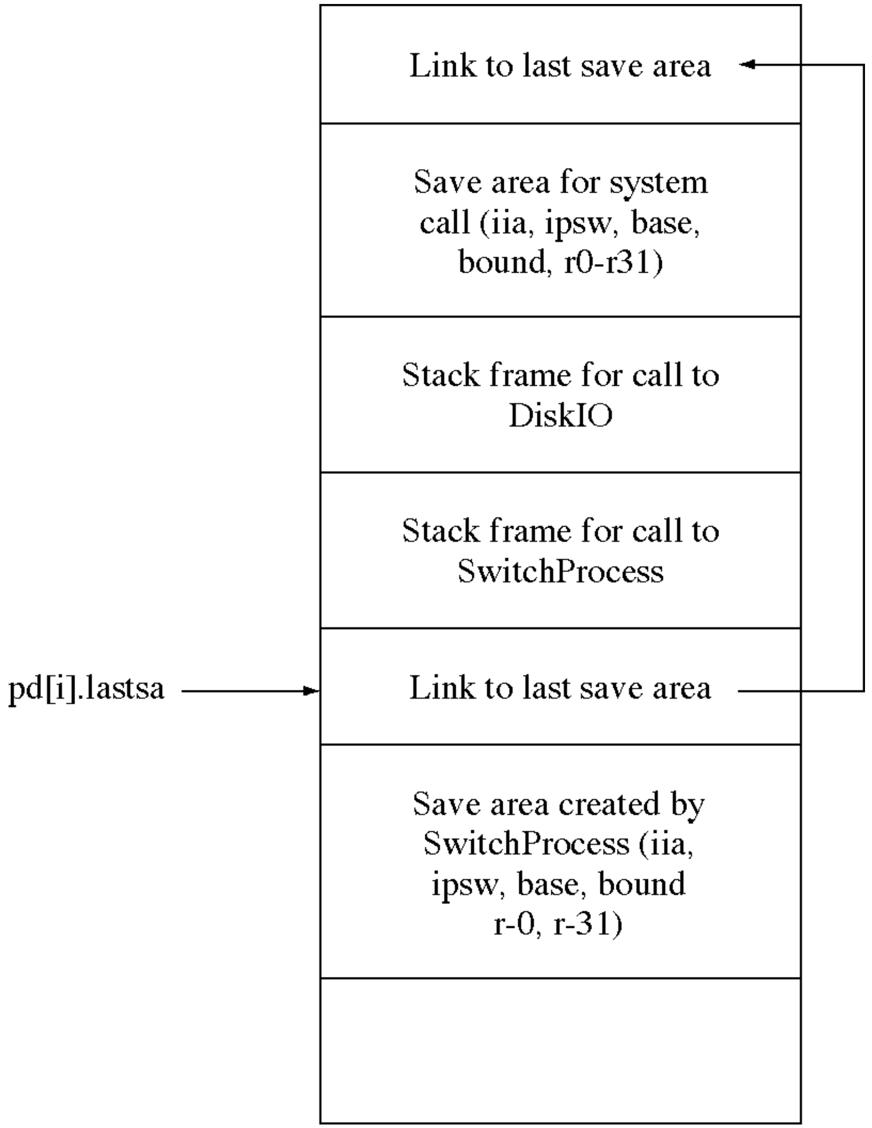
System calls (5 of 5)

```
• case ReceiveMessageSystemCall:
      int * user_msg; asm { store r9,user_msg }
      int from_q; asm { store r10,from_q }
      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;
        wait_queue[from_q].Insert( current_process );
        SwitchProcess(); }
      int msg_no = message_queue[from_q].Remove();
      TransferMessage( msg_no, user_msg );
      pd[current_process].sa.reg[1] = 0;
      break;
    case DiskReadSystemCall:
    case DiskWriteSystemCall:
      // ... the same
    }
    Dispatcher();
  }}
```

Kernel-mode: SwitchProcess

-
  ```
  void SwitchProcess() {
    // Called when a system mode process wants
    // to wait for something
    int * savearea; asm { store r30,savearea }
    savearea -= sizeof(SaveArea)+4;
    asm {
      // save the registers.
      // arrange to return from the procedure call
      // when we return.
      store    r31,savearea+4
      store    psw,savearea+8
      store    base,savearea+12
      store    bound,savearea+16
      storeall savearea+20
    }
    pd[current_process].state = Blocked;
    Dispatcher();
  }
  ```

Kernel-mode process system stack


Link to last save area
Save area for system call (iia, ipsw, base, bound, r0-r31)
Stack frame for call to DiskIO
Stack frame for call to SwitchProcess
pd[i].lastsa
Link to last save area
Save area created by SwitchProcess (iia, ipsw, base, bound r-0, r-31)

Kernel-mode: DiskIO

-
  ```
  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( (void *)req );
    pd[current_process].state = Blocked;

    // Wake up the disk scheduler if it is idle.
    ScheduleDisk();
    SwitchProcess(); // NEW CODE
  }
  ```

Kernel-mode: dispatcher

-
  ```
  void RunProcess( int pid ) {
    if( pid >= 0 ) {
      asm { move   #0,psw } // Disable interrupts
      int * savearea = pd[pid].lastsa;
      pd[pid].lastsa = *savearea;
      --(pd[pid].inSystem);
      int quantum = pd[pid].timeLeft
      asm {
        load    savearea+4,iia
        load    savearea+8,ipsw
        load    savearea+12,base
        load    savearea+16,bound
        loadall savearea+20
        load    quantum,timer
        rti
      }
    } else {
      waitLoop: goto waitLoop;
    }
  }
  ```

Kernel-mode: schedule disk

-
  ```
  void ScheduleDisk( void ) {
    StartUsingProcessTable();
    if( pd[scheduleDiskPid].state == Blocked )
      pd[scheduleDiskPid].state = Ready;
    FinishUsingProcessTable();
  }
  ```

Kernel-mode: real schedule disk

-
  ```
void RealScheduleDisk( void ) {
  while( 1 ) { // NEW CODE
    // If the disk is already busy, wait for it.
    if( DiskBusy() )
      SwitchProcess(); // NEW CODE
    // Get the first disk request
    // from the disk request queue.
    DiskRequest * req = disk_queue.Remove();
    // Wait, if there is no disk request to service.
    if( req == 0 )
      SwitchProcess(); // NEW CODE
    // record which process is waiting for the disk
    process_using_disk = req->pid;
    // issue read or write, with interrupt enabled
    if( req->command == DiskReadSystemCall )
      IssueDiskRead(req->disk_block,req->buffer,1);
    else
      IssueDiskWrite(req->disk_block,req->buffer,1);
  }
}
```

Kernel-mode process tradeoffs

- Waiting inside the kernel is easy
- Interrupts inside the kernel are easy
- But every process needs a kernel-mode stack
  - this adds up to a lot of memory allocated to stacks
- UNIX implementations have used kernel-mode processes from the beginning
  - but newer versions are getting away from it to save memory

Implementation of mutual exclusion in the OS

- Three low-level solutions
  - disable interrupts
    - easy but only possible for single processor systems
  - special hardware instruction (exchangeword)
    - the preferred solution
  - software mutual exclusion
    - no special hardware required

Using mutual exclusion

-
  ```
  void Process1( void ) {
    while( 1 ) {
      DoSomeStuff();
      EnterCriticalSection( 0 );
      DoCriticalSectionStuff();
      LeaveCriticalSection( 0 );
    }
  }
  void Process2( void ) {
    while( 1 ) {
      DoSomeStuff();
      EnterCriticalSection( 1 );
      DoCriticalSectionStuff();
      LeaveCriticalSection( 1 );
    }
  }
  ```

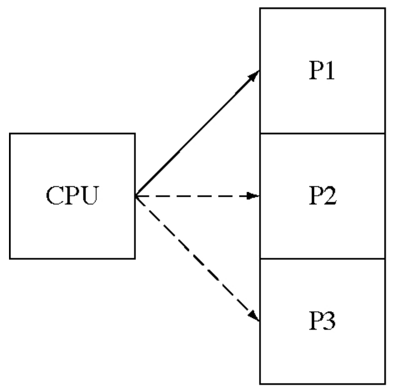
Implementing mutual exclusion

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

  // This is global data available to both processes
  int interested[2] = {False, False};
  int turn = 0;

  void EnterCriticalSection( int this_process ) {
    int other_process = 1 - this_process;
    interested[this_process] = True;
    turn = this_process;
    while( turn == this_process
           && interested[other_process] ) {
      // do nothing, just wait for this loop to exit
    }
  }

  void LeaveCriticalSection( int this_process ) {
    interested[this_process] = False;
  }
  ```

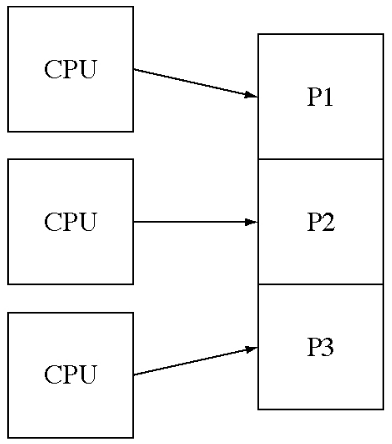
Comparing the three solutions

- Disabling interrupts
  - is fast and does not involve busy waiting
  - is the best solution for a single processor
- Using ExchangeWord
  - requires hardware assistance and busy waiting
  - is the best solution for multiprocessors which share memory
- Peterson’s solution
  - requires busy waiting but no special hardware
  - is the best solution for distributed systems with no central control

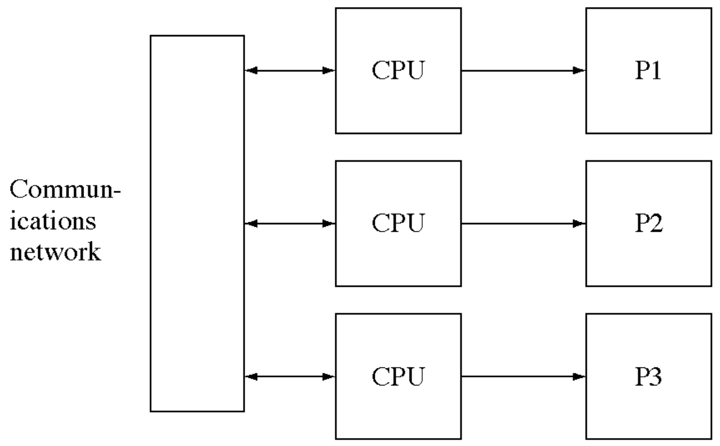
Varieties of multiple processors



(a) Multiprogramming — one CPU rapidly switched between processes



(b) Multiprocessing (shared memory) — multiple CPUs but one memory



(c) Multicomputers (no shared memory) — multiple CPU+memory units

Mutual exclusion (Dekker's)

-
  ```
  // This is global data available to both processes
  int interested[2] = {False, False};
  int turn = 0;
  void EnterCriticalSection( int this_process ) {
    int other_process = 1 - this_process;
    interested[this_process] = True;
    while( interested[other_process] ) {
      if( turn == other_process ) {
        interested[this_process] = False;
        while( turn == other_process )
          /* do nothing*/ ;
        interested[this_process] = True;
      }
    }
  }
  void LeaveCriticalSection( int this_process ) {
    int other_process = 1 - this_process;
    turn = other_process;
    interested[this_process] = False;
  }
  ```