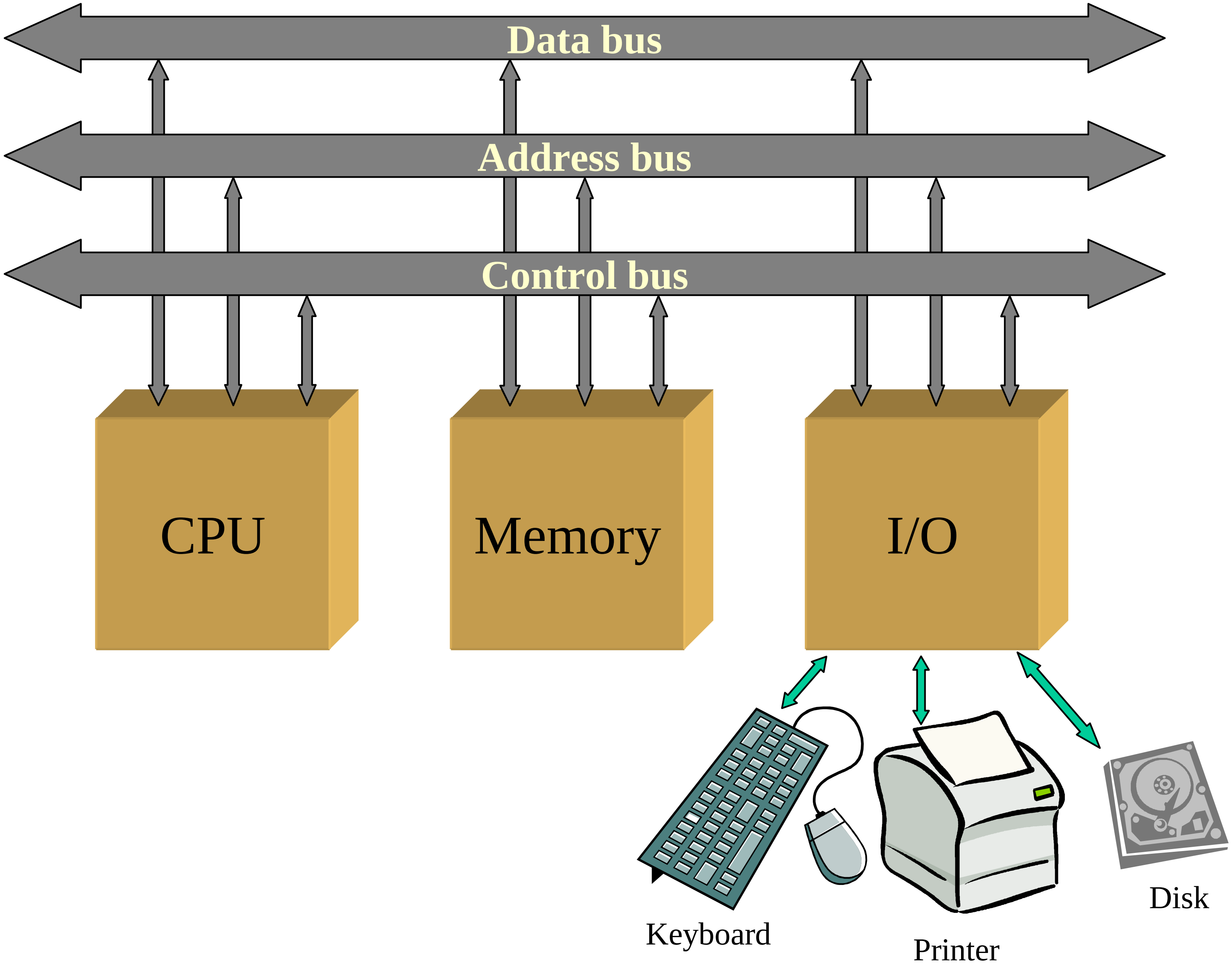
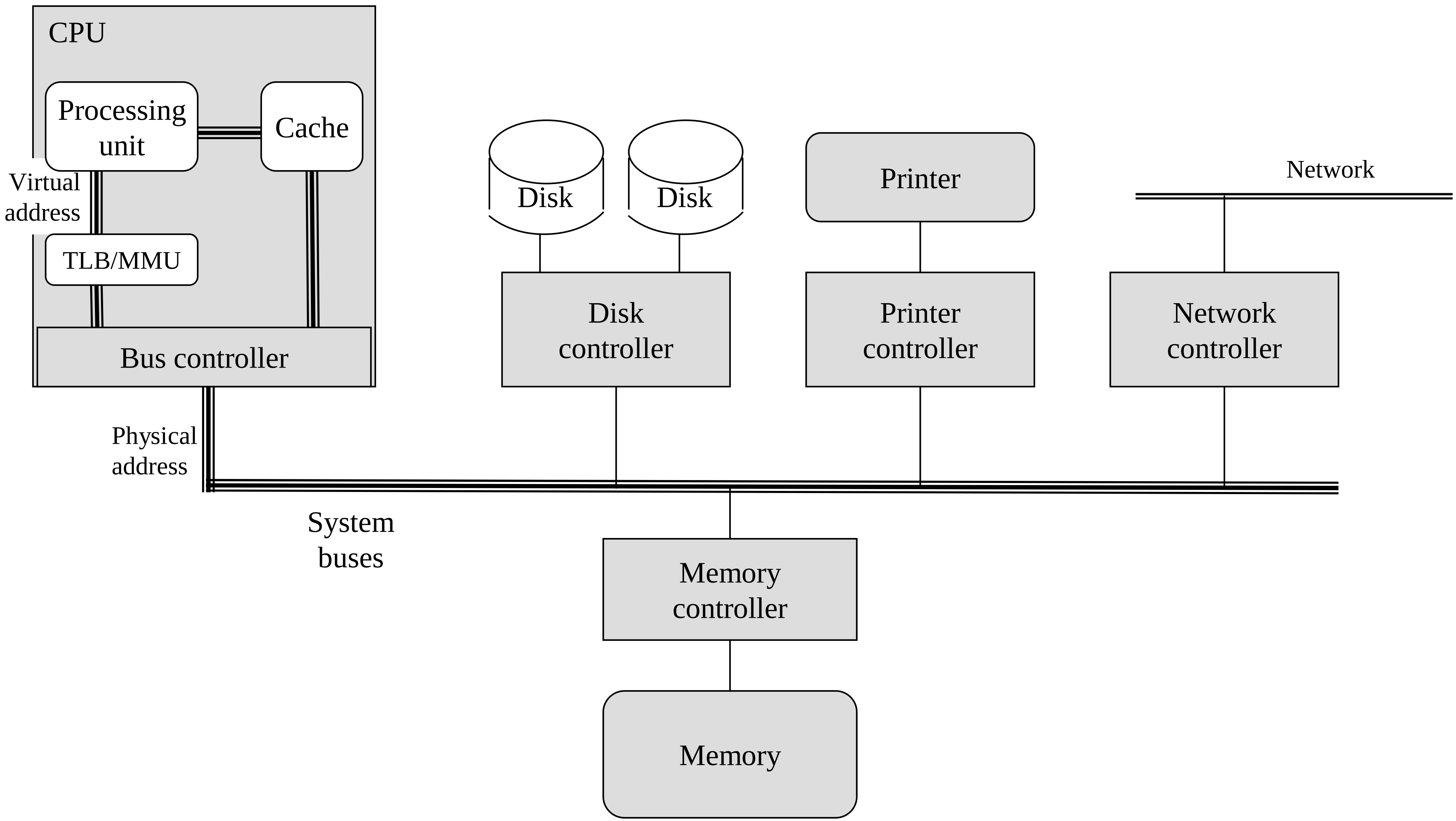
Topic 2. System calls

1. Basic architecture
2. Input/Output routine mechanism
3. Resident routines
4. Accessing OS services: system calls

Von Neumann architecture



Basic architecture: controllers

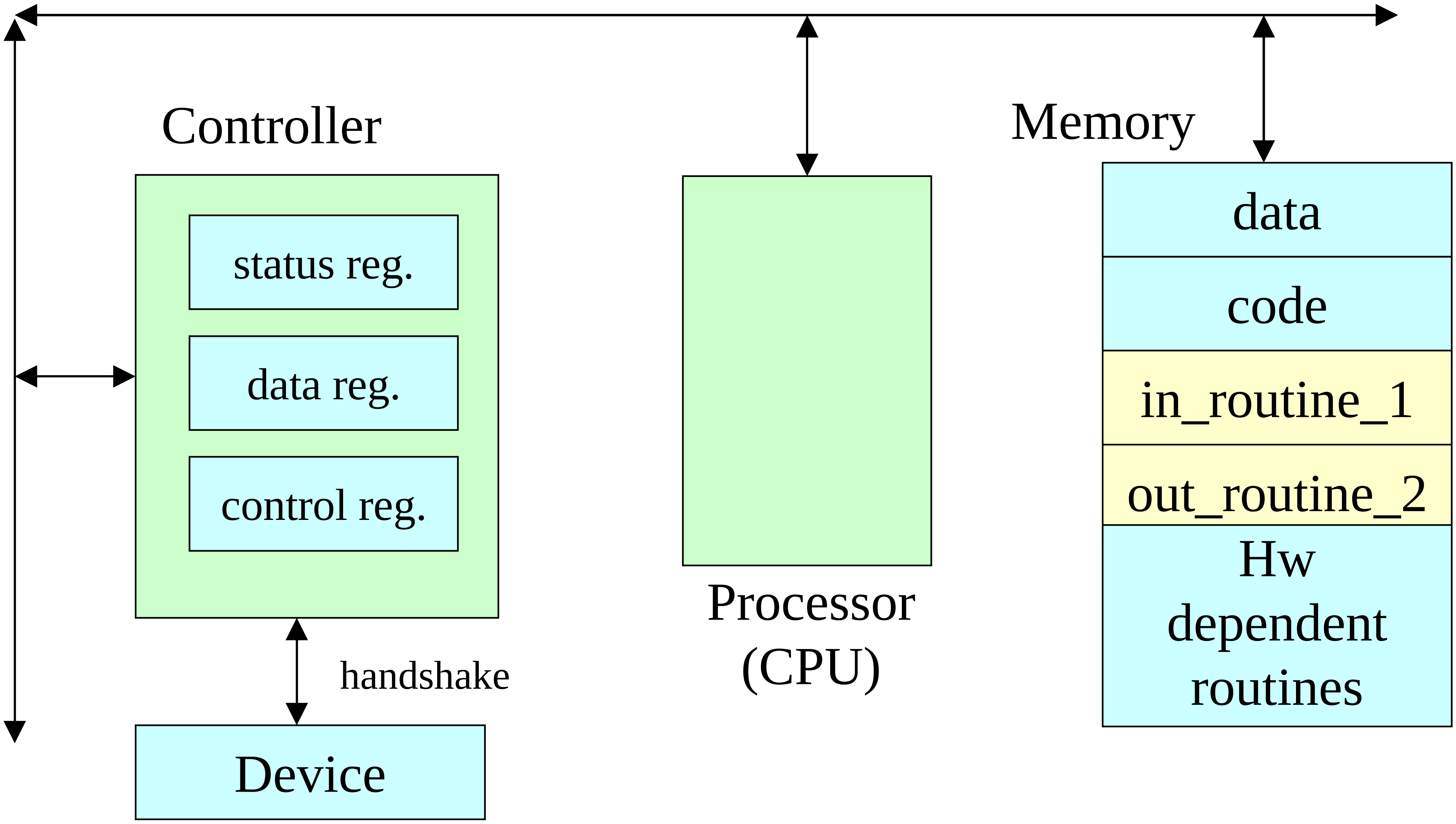


Basic architecture

- If all components (CPU, memory, bus, devices) are efficient, the whole computer can also be efficient, but not necessarily…
  - The **tuned management** of all resources will be largely responsible for the success or failure

- In this regard, the **Operating Systems** is the set of routines in charge of the **efficient management** of all the resources of the computer
  - The way this management is carried out will be responsible of the good/poor performance of the computer

Input/Output by busy waiting (spinning)



Input/Output by busy waiting (spinning)

- Procedure for using the device:
  1. Check if the device (controller) is ready for reading/writing
  2. When the device is ready, read/write the data
  3. Indicate to the controller that the data has been read/written

- Checking if the device controller is ready is done by busy waiting on reading the status register

- It is the responsibility of the user program to check the status register and call the appropriate error handling routine when an error occurs

Input/Output by busy waiting - Example

Example: read 80 characters from an input device *DEV1* and write them in an output device *DEV2*

Routine types:

Hardware dependent routines:
access the registers of the controllers (assumed to be coded)

Input/Output routines:
perform the input and output operations

error routine:
checks if there is an error, in which case it finishes the execution of the program (assumed to be coded)

⇨ Cooked/raw Input/Output: control-characters

Line feed (LF), Backspace (BS), End of file (EOF)…

Input/Output by busy waiting - Example

Input routine for device

```
in_routine_1(char *vector, int count)
{
   int j, status;

   for (j = 0; j < count; j++)
   {
      do {
         status = read_status_register(DEV1);
      } while (status == BUSY);
      error(DEV1, status);
      vector[j] = read_data_register(DEV1);
      write_control_register(DEV1, READ);
   }
}
```

Output routine for device 2

```
out_routine_2(char *vector, int count)
{
   int j, status;

   for (j = 0; j < count; j++)
   {
      do {
         status = read_status_register(DEV2);
      } while (status == BUSY);
      error(DEV2, status);
      write_data_register(DEV2, vector[j]);
      write_control_register(DEV2, WRITTEN);
   }
}
```

User program

(synchronous I/O)

```
main()
{
   char buff[80];

   while (TRUE) {
      in_routine_1(buff, 80);
      out_routine_2(buff, 80);
   }
}
```

Input/Output by busy waiting

- Issue:
  - During Input/Output operations the CPU is most of the time doing nothing else than just waiting…
  - Waiting times can be “extreeeeeeeeeemely” long
    - I/O devices are slow (compared to the CPU)

- Solution:
  - Interrupt driven Input/Output

Memory hierarchy

	Access time	Size
Register	0,5 ns	512 Byte
Cache On-chip (L1)	2 ns	32 KB - 64 KB
Cache Off-chip (L2)	5 - 10 ns	512 KB - 2 MB
Main memory	20 - 50 ns	2 GB
Magnetic disk	10 ms	100 GB
CD ROM	300 ms	700 MB
Magnetic tape	1 s - 1 min	GB / TB

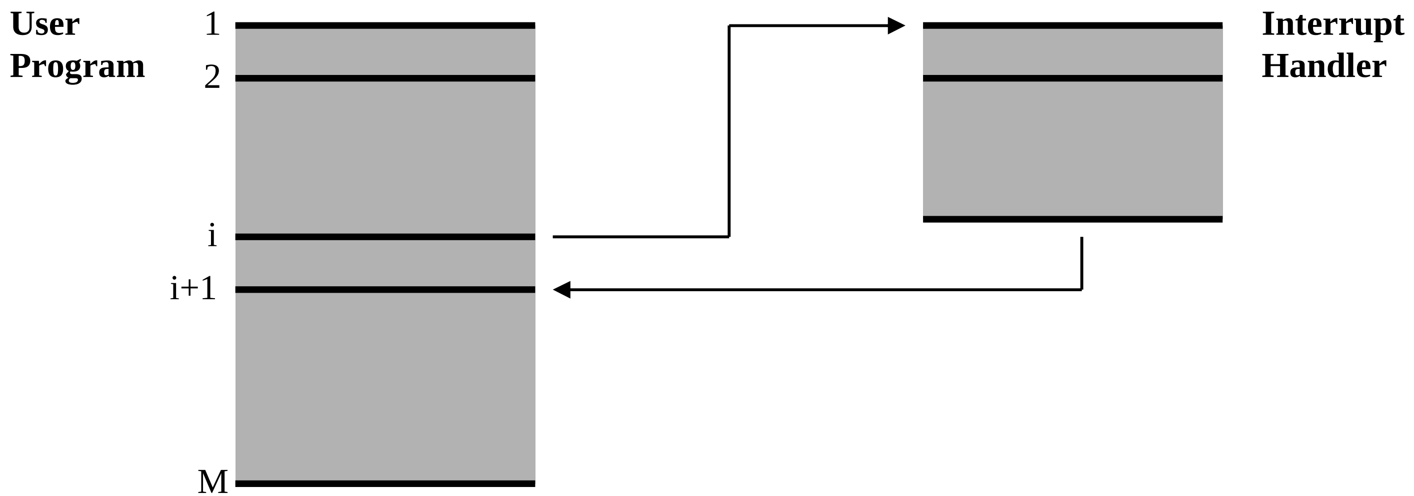
Interrupts

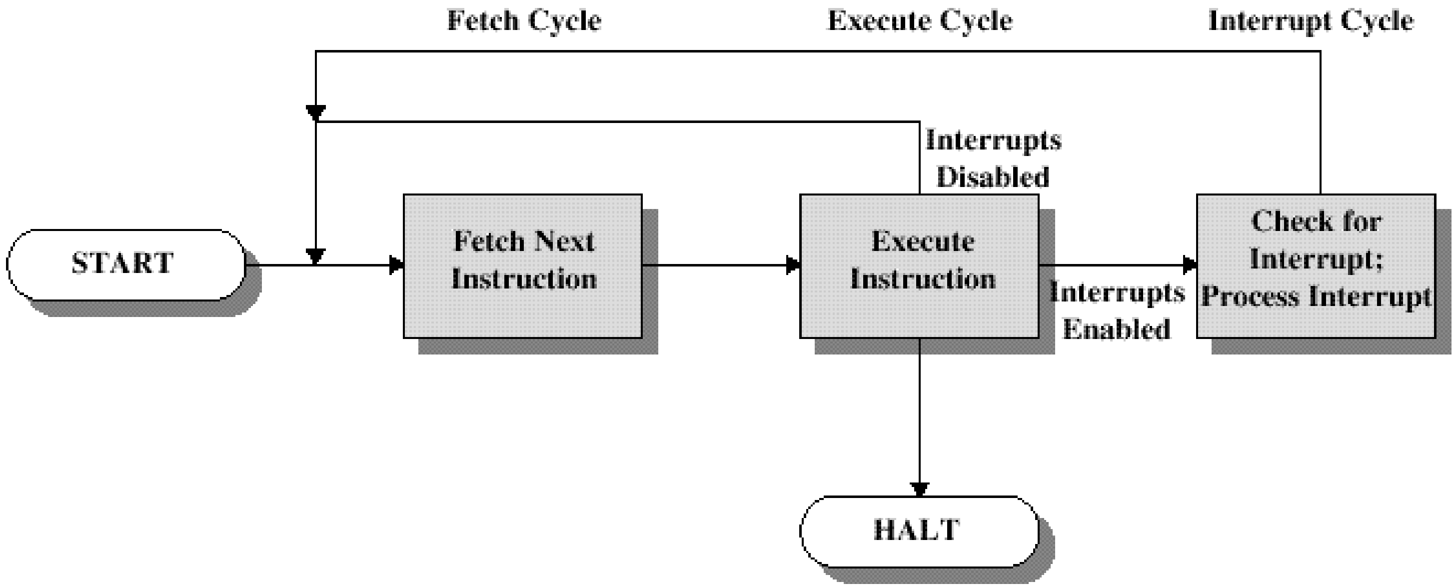
- Interrupt – event that occurs in a computing system, which affects the execution flow

- Types of interrupts:
  - Hardware interrupts
    - Clock interrupt (periodic)
    - I/O device interrupt (asynchronous)
    - Software error (arithmetic overflow, unknown instruction, wrong memory address…)
    - Hardware error (bus error)

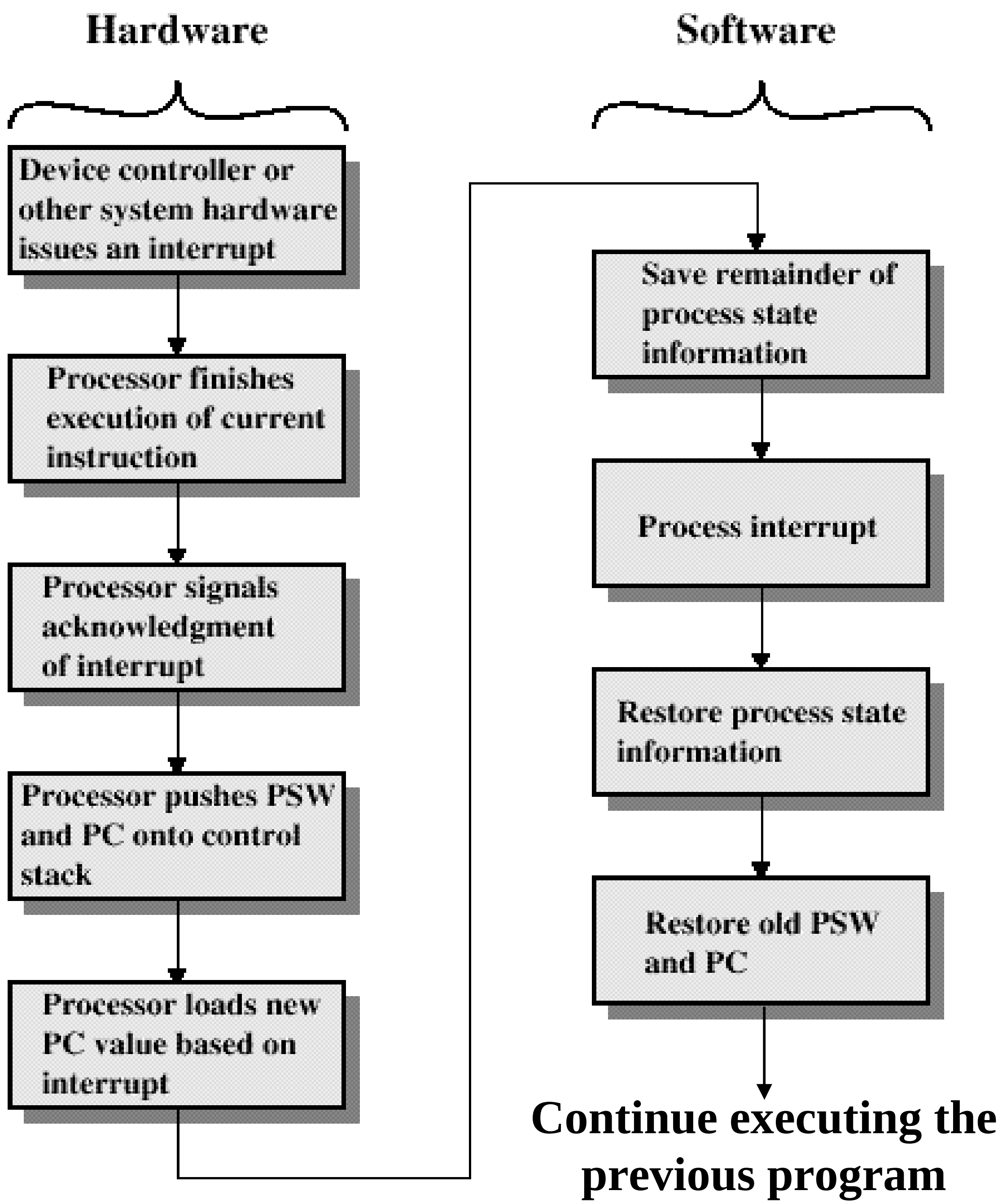
  - Software interrupt or *trap*
    - Special instruction of the processor
    - Synchronous with respect to the instruction sequence of the (calling) program

Instruction Control Flow and Instruction Cycle with Interrupts

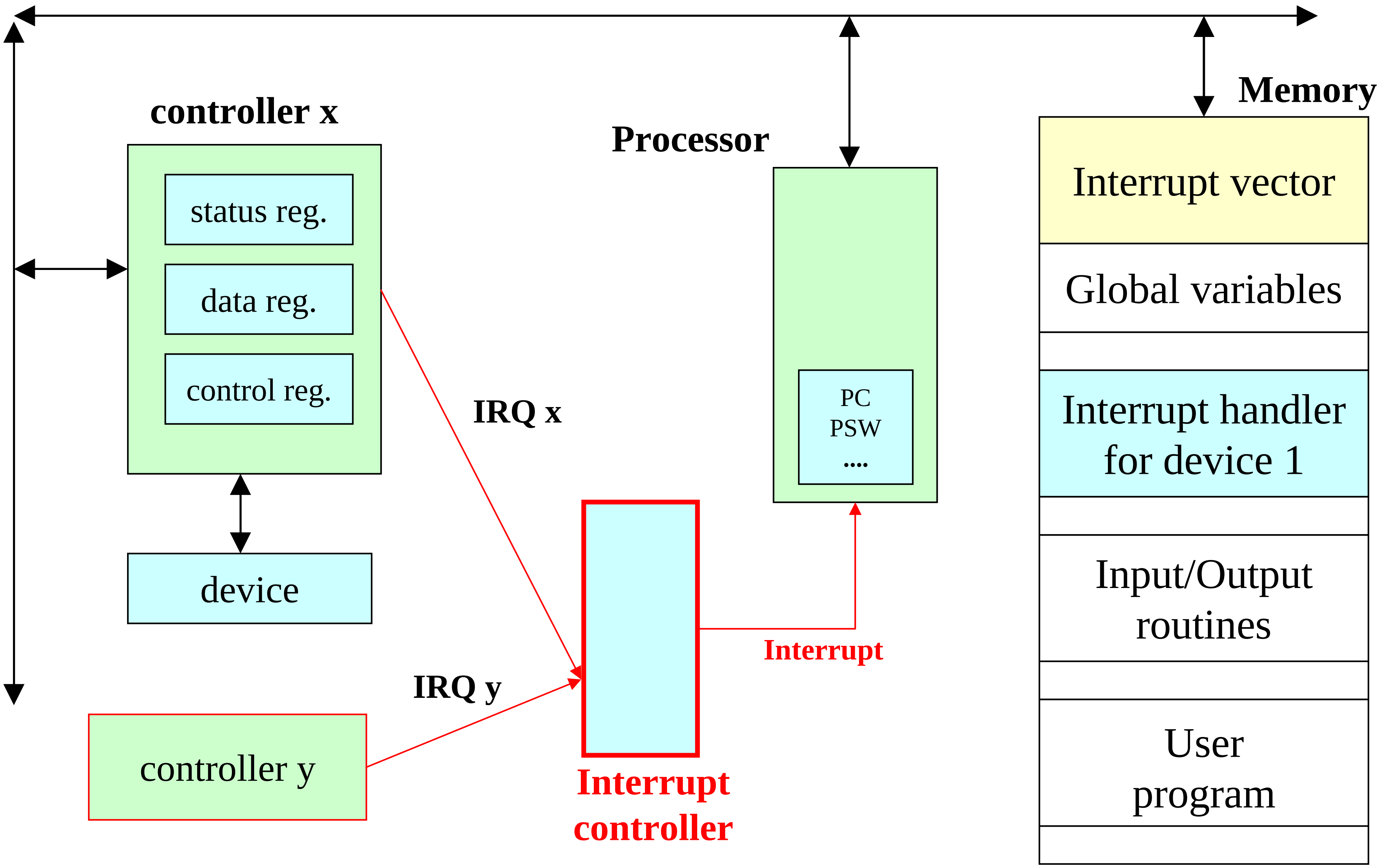




Interrupt handling



Interrupt driven Input/Output



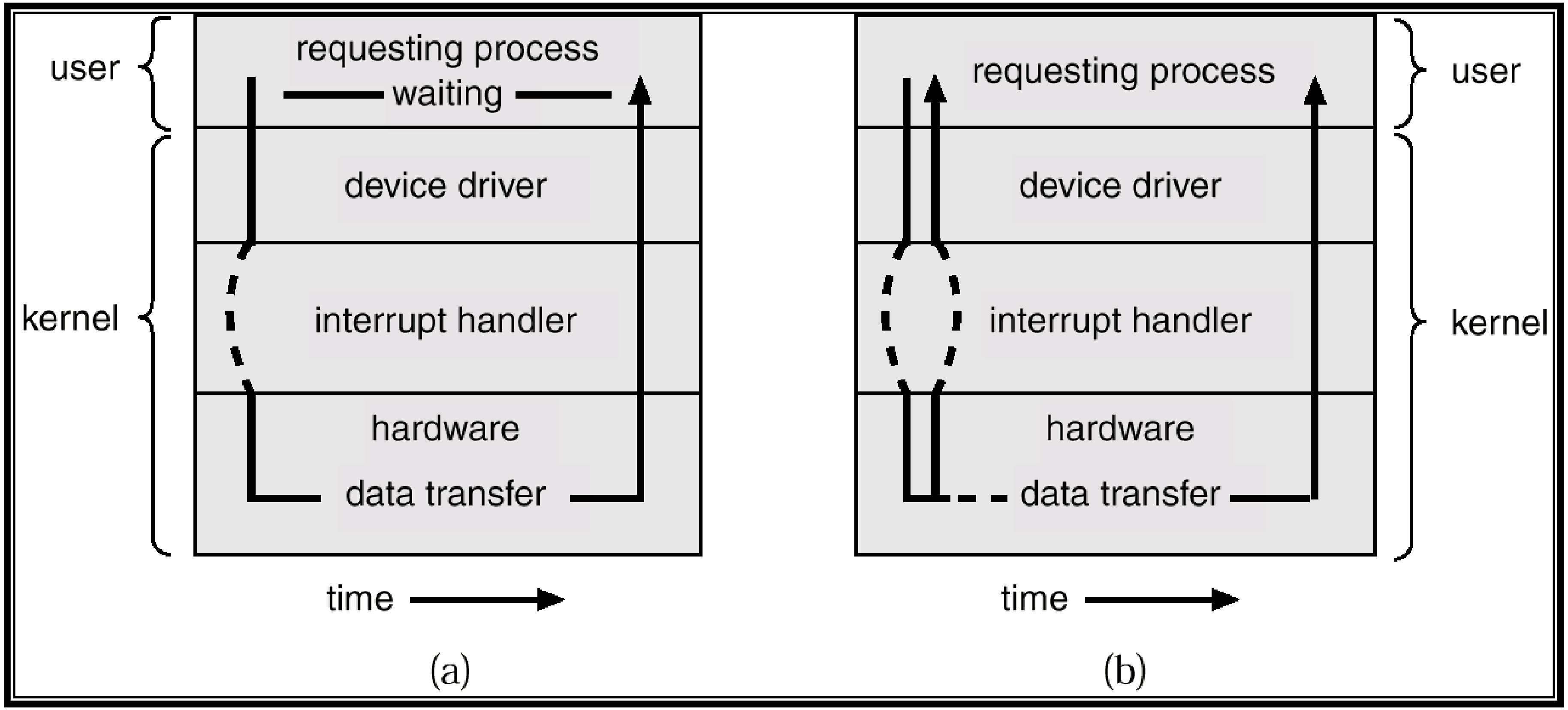
Interrupt driven Input/Output

- Provides parallelism: operations can be synchronous or asynchronous
- The device generates an interrupt when a character/block is read/written
- Besides the input and output routines, the programmer must code the interrupt-handler, which is executed when an interrupt occurs
- The interrupt vector must be updated with the address of the interrupt-handler routine

Types of I/O operations

Synchronous

Asynchronous



Interrupt driven Input/Output - Example

User program (synchronous)

```
main()
{
   char buff[80];

   change_interrupt_vector(DEV1, interrupt_handler_1);
   change_interrupt_vector(DEV2, interrupt_handler_2);
   while (TRUE)
   {
      in_routine_1(buff, 80, SYNC);
      out_routine_2(buff, 80, SYNC);
   }
}
```

Global variables

```
int end1 = TRUE, end2 = TRUE;
char *buff1, *buff2;
int count1 = 0, count2 = 0;
int index1, index2;
```

Synchronization routine

```
void synchronize(int *end)
{
   while ((*end) == FALSE) NOP;
}
```

Input routine for device 1

```
in_routine_1(char *vector, int count, int async_sync)
{
   end1 = FALSE; buff1 = vector; index1 = 0;
   count1 = count;
   if (async_sync == SYNC)
      synchronize(&end1);
}
```

Output routine for device 2

```
out_routine_2(char *vector, int count, int async_sync)
{
   int status;
   end2 = FALSE;
   /* first write by busy waiting */
   do {
      status = read_status_register(DEV2);
   } while (status == BUSY);
   error(DEV2, status);
   buff2 = vector;
   index2 = 1;
   count2 = count;
   write_data_register(DEV2, buff2[0]);
   write_control_register(DEV2, WRITTEN);
   if (async_sync == SYNC)
      synchronize(&end2);
}
```

Interrupt driven Input/Output - Example

Interrupt handler for device 1

```
interrupt_handler_1()
{
  int status;

  if (count1 != 0)
  {
    status = read_status_register(DEV1);
    error(DEV1, status);
    buff1[index++] =
             read_data_register(DEV1);
    count1--;
    if (count1 == 0) end1 = TRUE;
  }
  write_control_register(DEV1, READ);
  end_interrupt_handler();   /* EOI, IRET */
}
```

Interrupt handler for device 2

```
interrupt_handler_2()
{
  int status;

  status = read_status_register(DEV2);
  error(DEV2, status);
  count2--;
  if (count2 > 0)
  {
    write_data_register(DEV2,
                      buff2[index2++]);
    write_control_register(DEV2, WRITTEN);
  }
  else end2 = TRUE;
  end_interrupt_handler();   /* EOI, IRET */
}
```

Interrupt driven Input/Output - Example

User program (synchronous I/O)

```
main()
{
  char buff[80];

  change_interrupt_vector(DEV1,
                          interrupt_handler_1);
  change_interrupt_vector(DEV2,
                          interrupt_handler_2);
  while (TRUE)
  {
    in_routine_1(buff, 80, SYNC);
    out_routine_2(buff, 80, SYNC);
  }
}
```

User program (sync. I, async. O)

```
main()
{
  char v1[80], v2[80];

  change_interrupt_vector(DEV1,
                          interrupt_handler_1);
  change_interrupt_vector(DEV2,
                          interrupt_handler_2);
  while (TRUE)
  {
    in_routine_1(v1, 80, SYNC);
    synchronize(&end2);
    out_routine_2(v1, 80, ASYNC);
    in_routine_1(v2, 80, SYNC);
    synchronize(&end2);
    out_routine_2(v2, 80, ASYNC);
  }
}
```

Interrupt driven Input/Output - Example

User program (sync. I, async. O)

```
main()
{
   char v1[80], v2[80];

   change_interrupt_vector(DEV1,
                                       interrupt_handler_1);
   change_interrupt_vector(DEV2,
                                       interrupt_handler_2);
   while (TRUE) {
      in_routine_1(v1, 80, SYNC);
      synchronize(&end2);
      out_routine_2(v1, 80, ASYNC);
      in_routine_1(v2, 80, SYNC);
      synchronize(&end2);
      out_routine_2(v2, 80, ASYNC);
   }
}
```

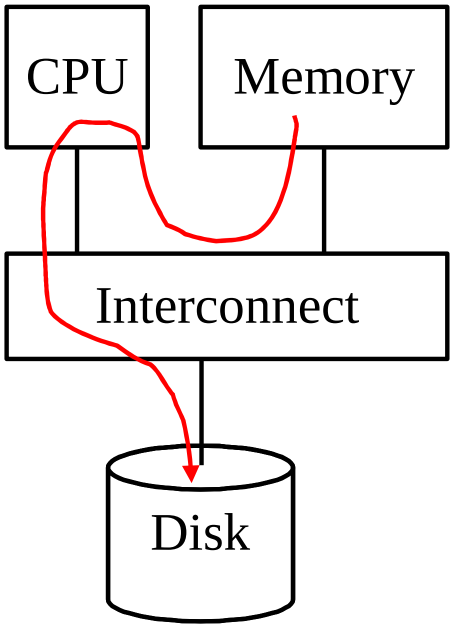
User program (asynchronous I/O)

```
main()
{
   char v1[80], v2[80];

   change_interrupt_vector(DEV1,
                                       interrupt_handler_1);
   change_interrupt_vector(DEV2,
                                       interrupt_handler_2);
   while (TRUE) {
      in_routine_1(v1, 80, ASYNC);
      synchronize(&end1);
      synchronize(&end2);
      out_routine_2(v1, 80, ASYNC);
      in_routine_1(v2, 80, ASYNC);
      synchronize(&end1);
      synchronize(&end2);
      out_routine_2(v2, 80, ASYNC);
   }
}
```

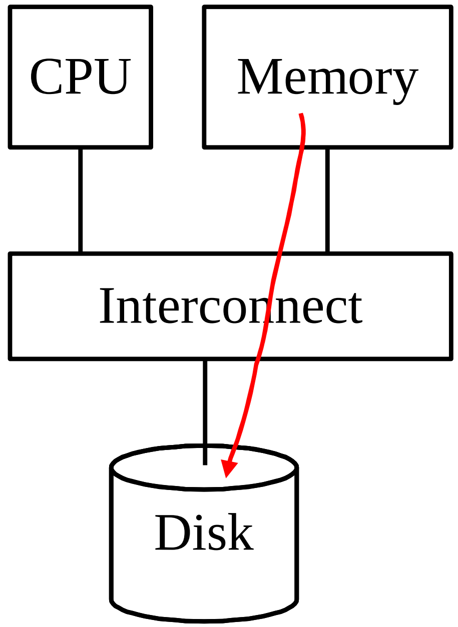
Direct Memory Access (DMA)

- Used to avoid programmed Input/Output for large data movement
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory

Programmed I/O vs DMA

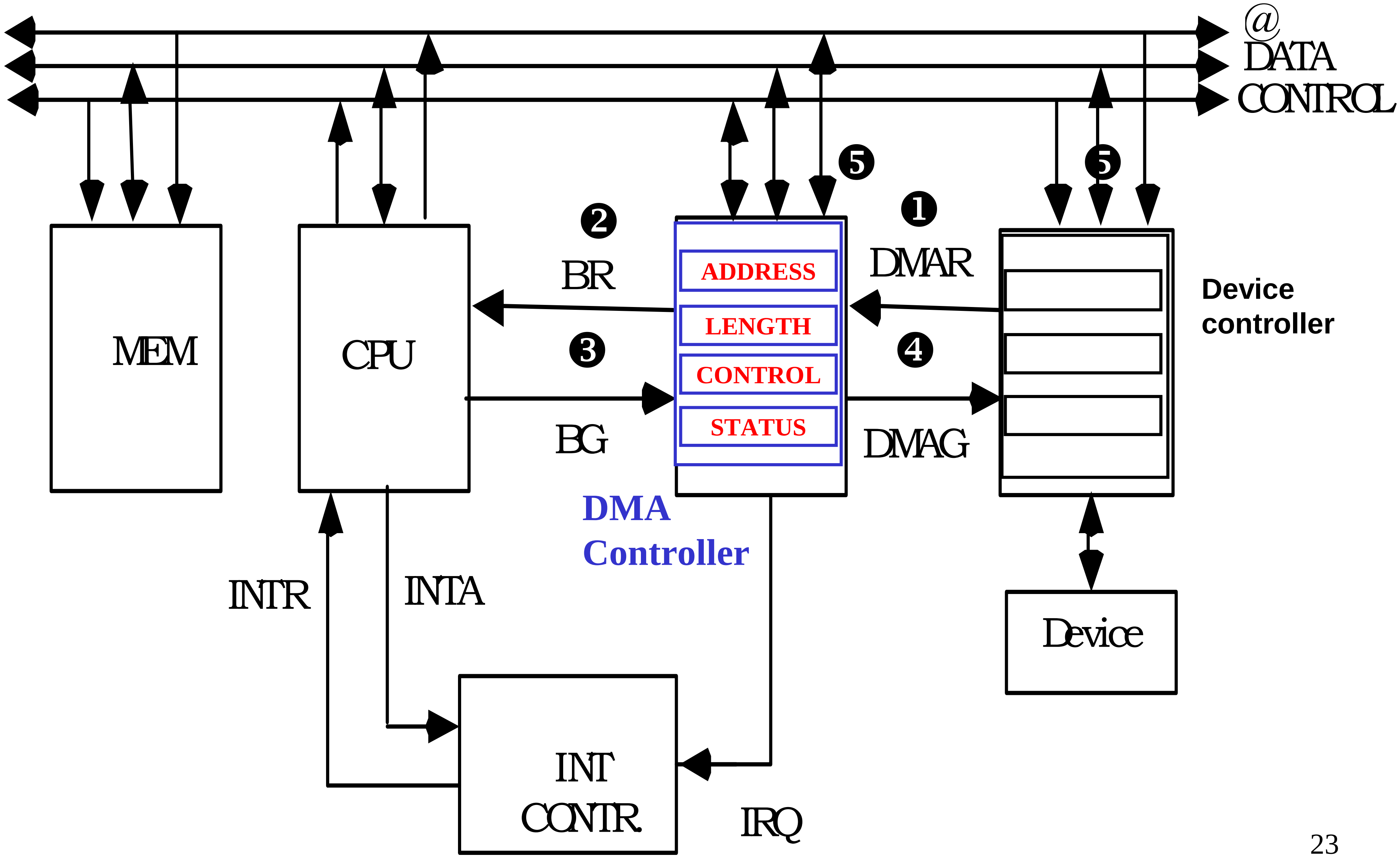


Programmed I/O



DMA

Direct memory Access (DMA) Architecture



DMA based Input/Output

- Provides a higher level of parallelism: the device and the DMA hardware take care of the data transfer in a parallel way

- When a data block must be transferred, the DMA controller establishes a protocol between the device an the memory

- The DMA controller generates an interrupt when the whole transfer has been accomplished. The programmer must code the interrupt handler routine.

DMA based Input/Output - Example

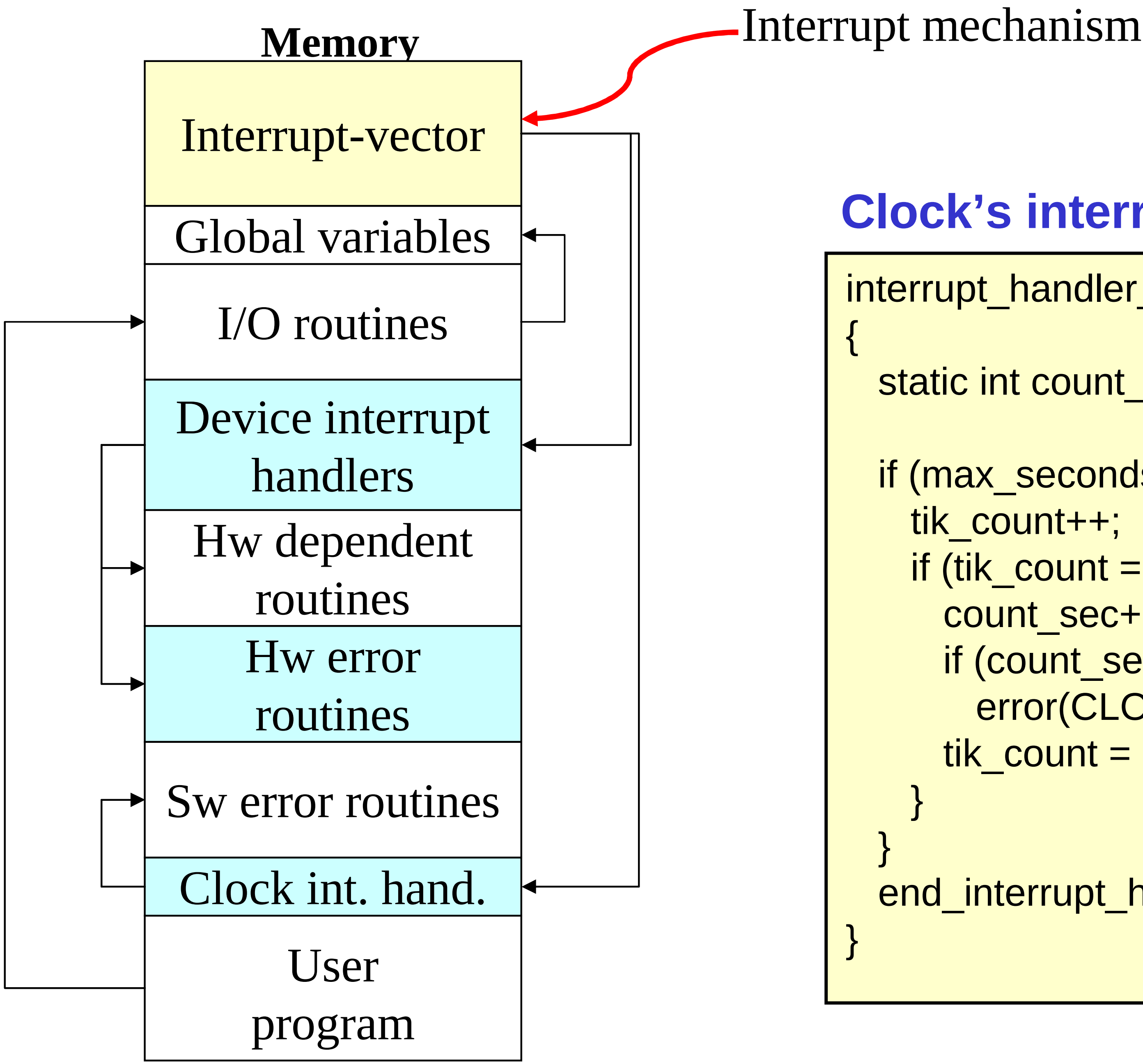
Interrupt handler for device 2

```
interrupt_handler_2()
{
   int status;

   status = read_status_register(CDMA);
   error(CDMA, status);
   end2 = TRUE;
   end_interrupt_handler();
}
```

Output routine for device 2

```
out_routine_2(char *vector, int count, int async_sync)
{
   int status;

   end2 = FALSE;
   do {
      status = read_status_register(CDMA);
   } while (status == BUSY);
   error(CDMA, status);
   write_address_register(CDMA, &(vector[1]));
   write_length_register(CDMA, count-1);
   write_control_register(CDMA, START_WRITE);
   program_controller(DEV2, vector[0]);
   if (async_sync == SYNC)
      synchronize(&end2);
}
```

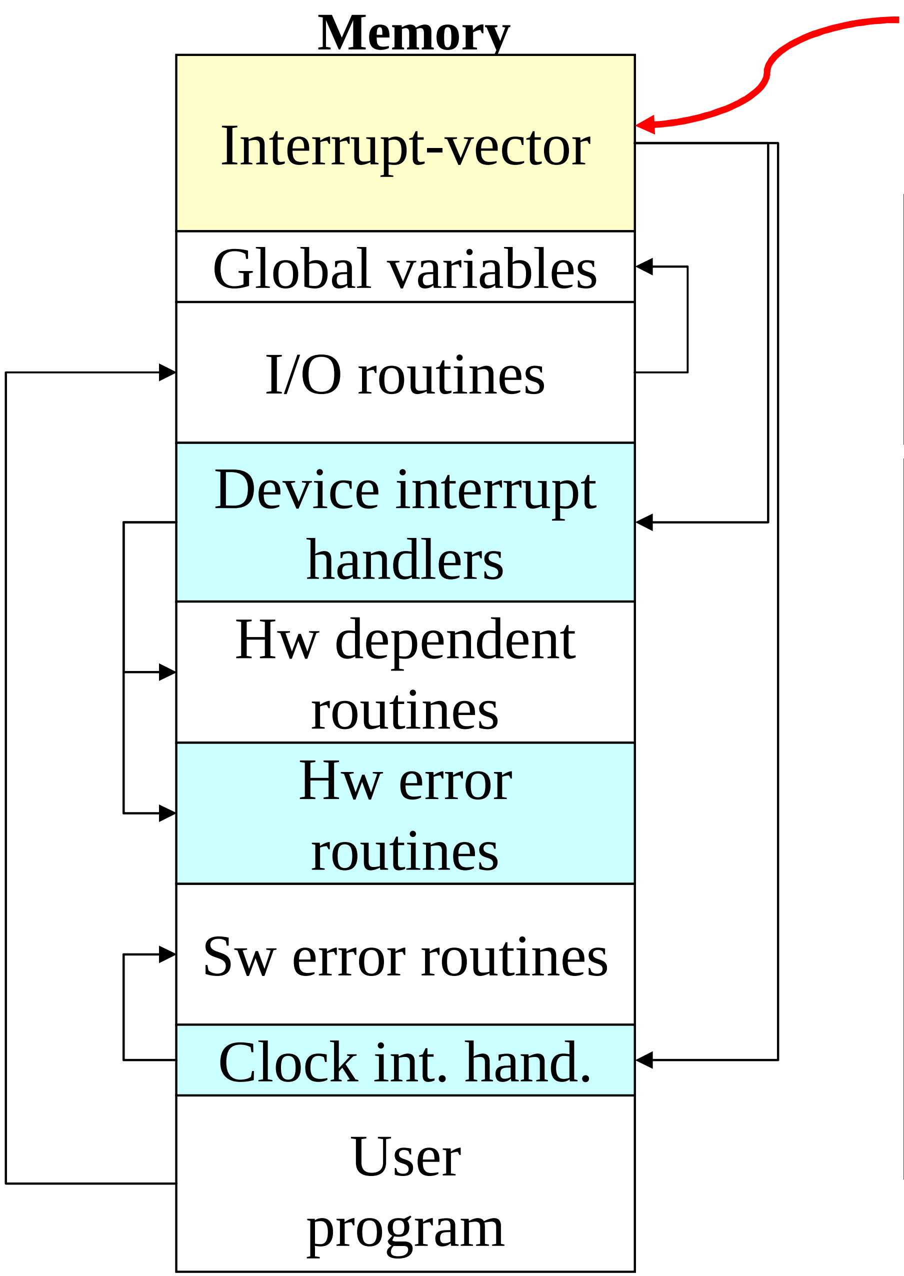
Time and error control



Clock's interrupt handler

```
interrupt_handler_clock()
{
  static int count_sec = 0, tik_count = 0;

  if (max_seconds != 0) {
    tik_count++;
    if (tik_count == ONE_SECOND) {
      count_sec++;
      if (count_sec == max_seconds)
        error(CLOCK, OVERTIME);
      tik_count = 0;
    }
  }
  end_interrupt_handler();
}
```

Time and error control



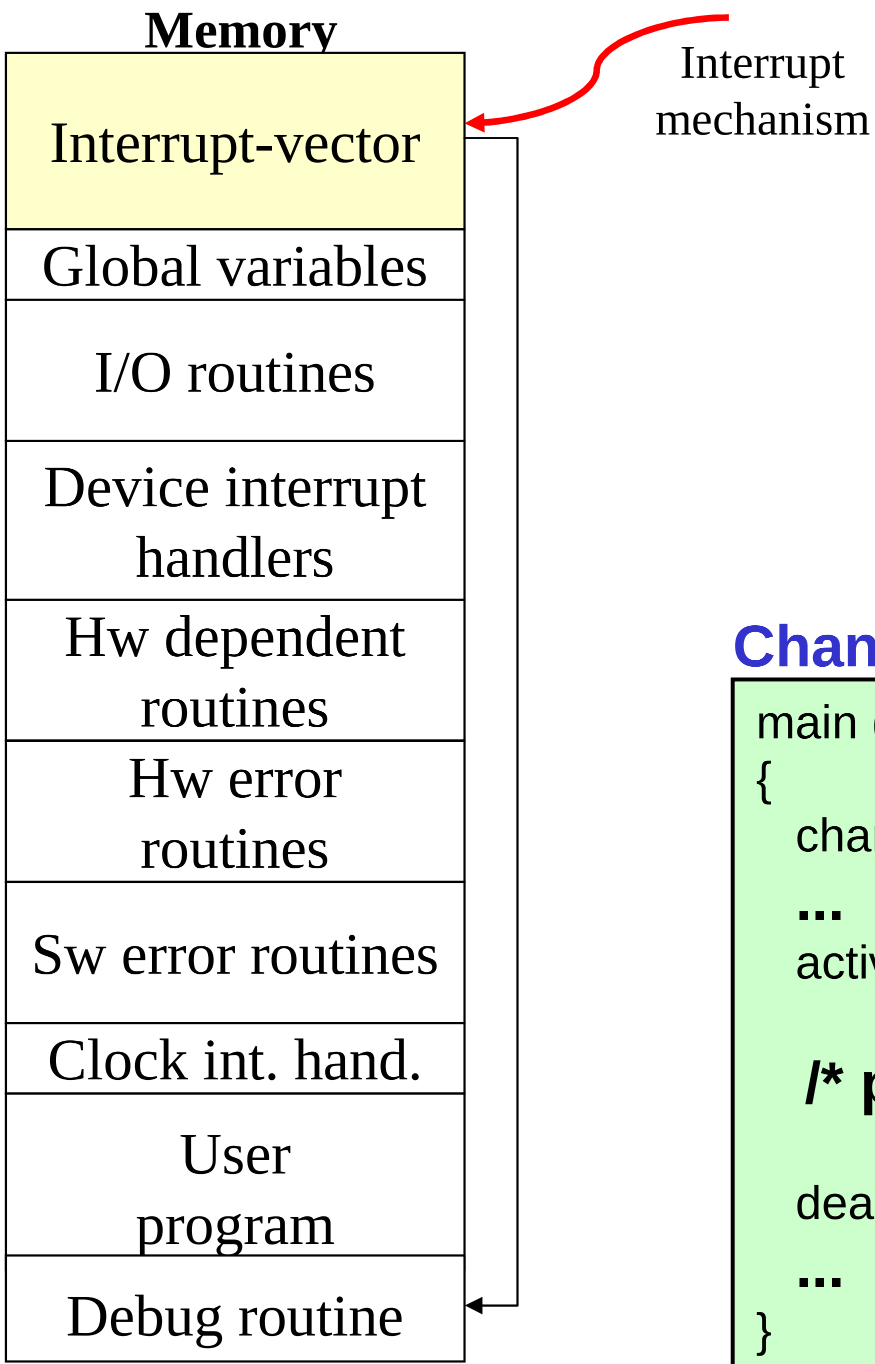
Global variables

```
…
int max_seconds = 0;
```

User program

```
main ()
{
   char buff[80];

   change_interrupt_vector(CLOCK,
                               interrupt_handler_clock);
   max_seconds = TIME_MAX;
   /* Rest of program code */
   …
}
```

Debugging



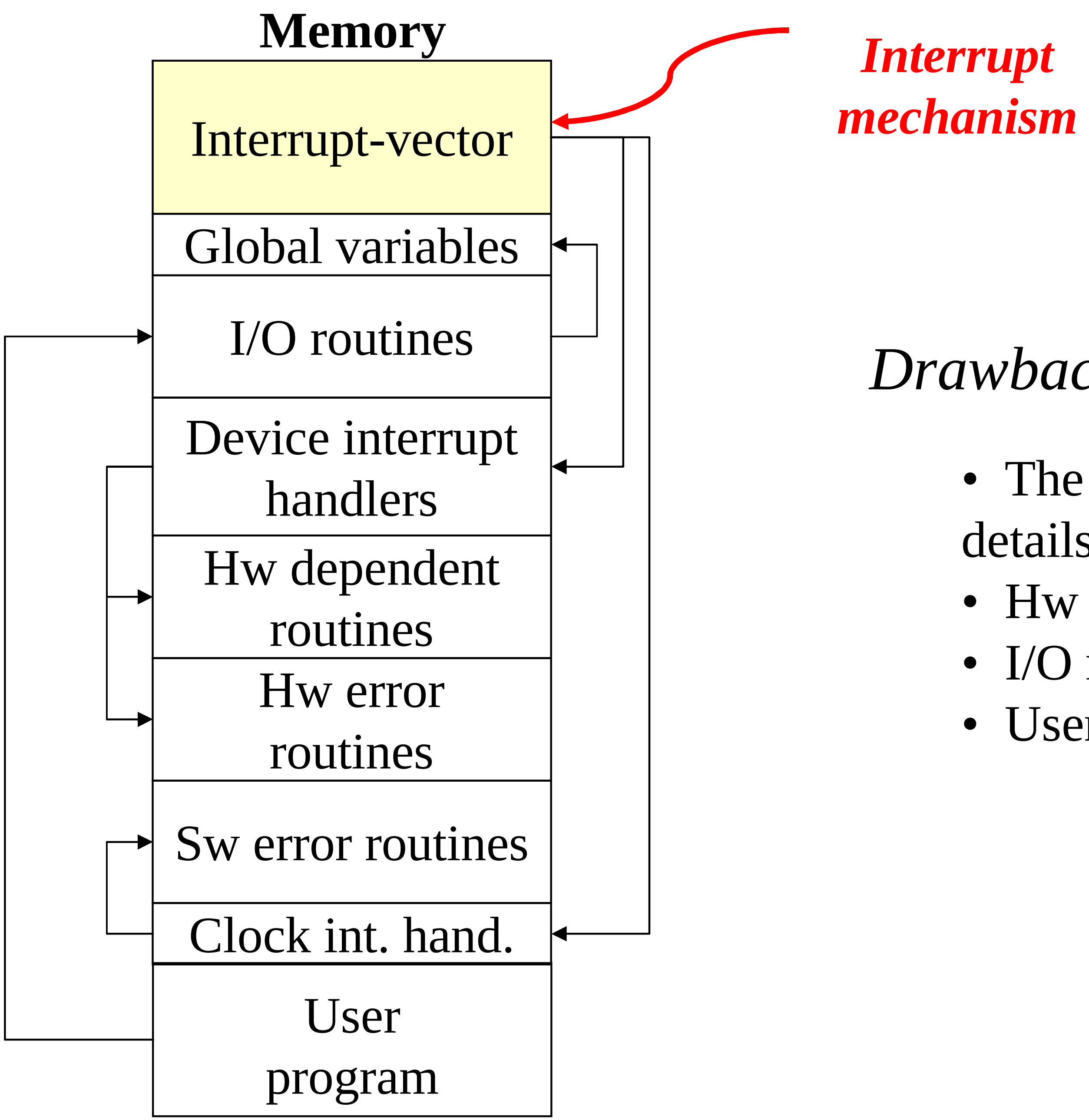
Debugging routine

```
debug_routine()
{
   deactivate_debug_bit();

   /* debugging code */

   activate_debug_bit();
}
```

Changes in the user program

```
main ()
{
   change_interrupt_vector(DEBUG, debug_routine);
   ...
   activate_debug_bit();

   /* program code */

   deactivate_debug_bit();
   ...
}
```

Interrupt driven I/O general scheme



Drawbacks of the mechanism:

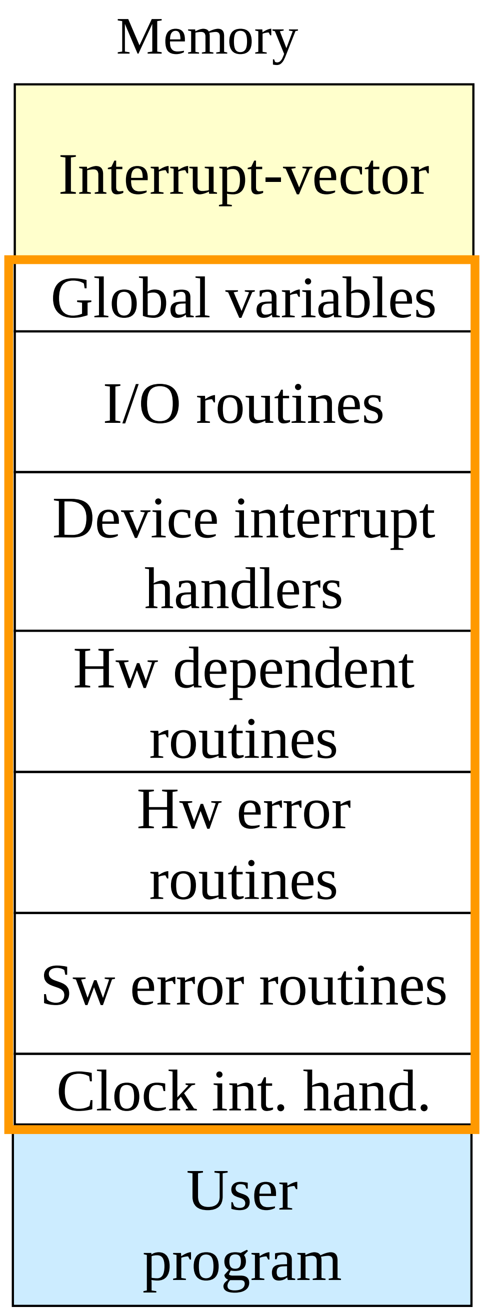
- The programmer must know the details of the machine/processor
- Hw dependency (changes?)
- I/O routine update
- User program update, life-cycle

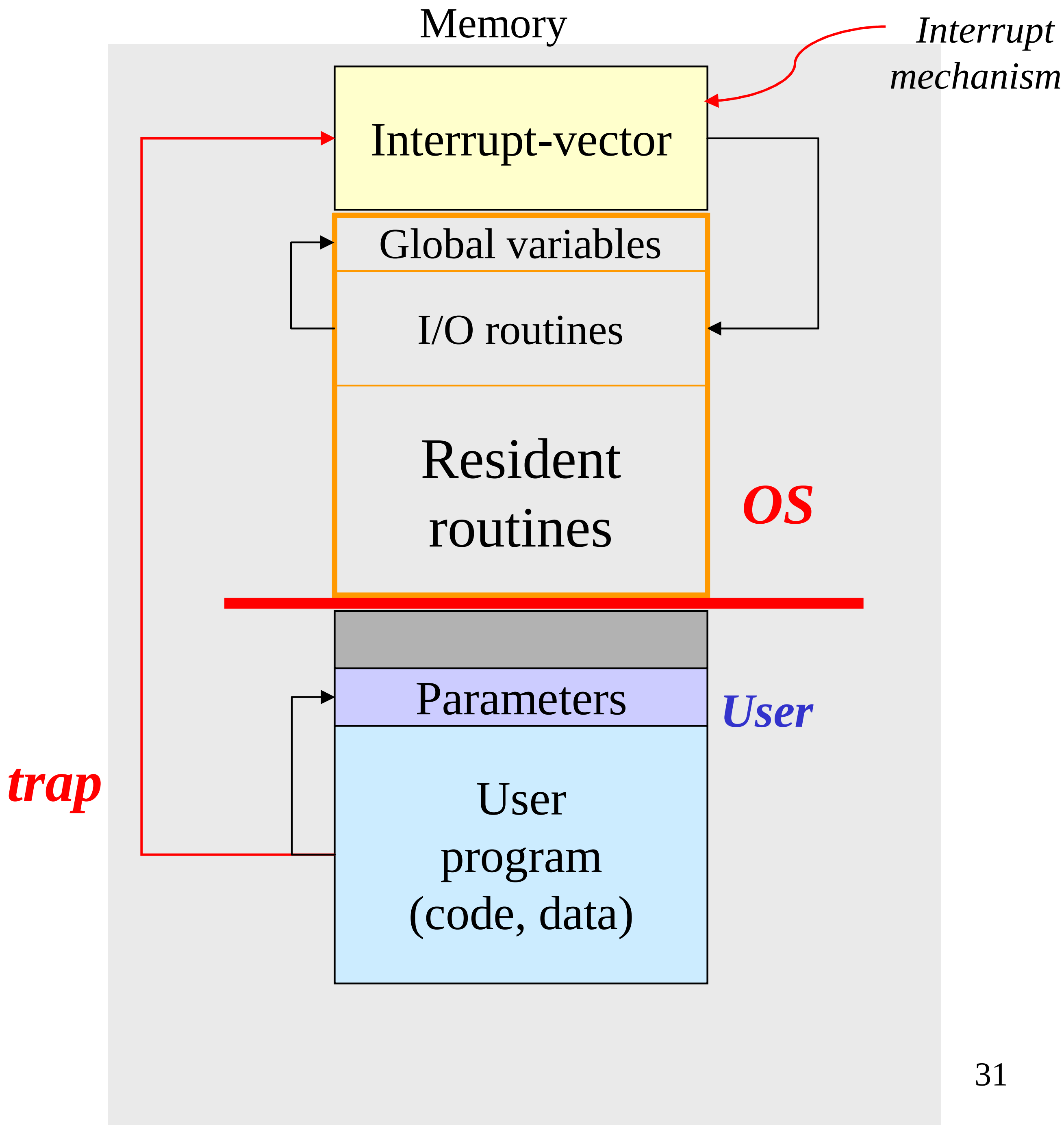
System calls

- Interface between the user program and the operating system
- Each system call has a concrete number
- System calls are usually implemented using a special instruction (software interrupt or *TRAP*) of the processor:

INT xx, int86()	(Intel x86)
SVC	(IBM 360/370) – supervisory call
trap	(PDP 11)
tw	(PowerPC) – trap word
tcc	(Sparc)
break	(MIPS)

Interrupt-vector based system call





System calls

- Nowadays, system calls can be used through high level programming languages (e.g., C)

- Programming is easier:

  SVC 15   vs   read(file-d, buffer, n-bytes)

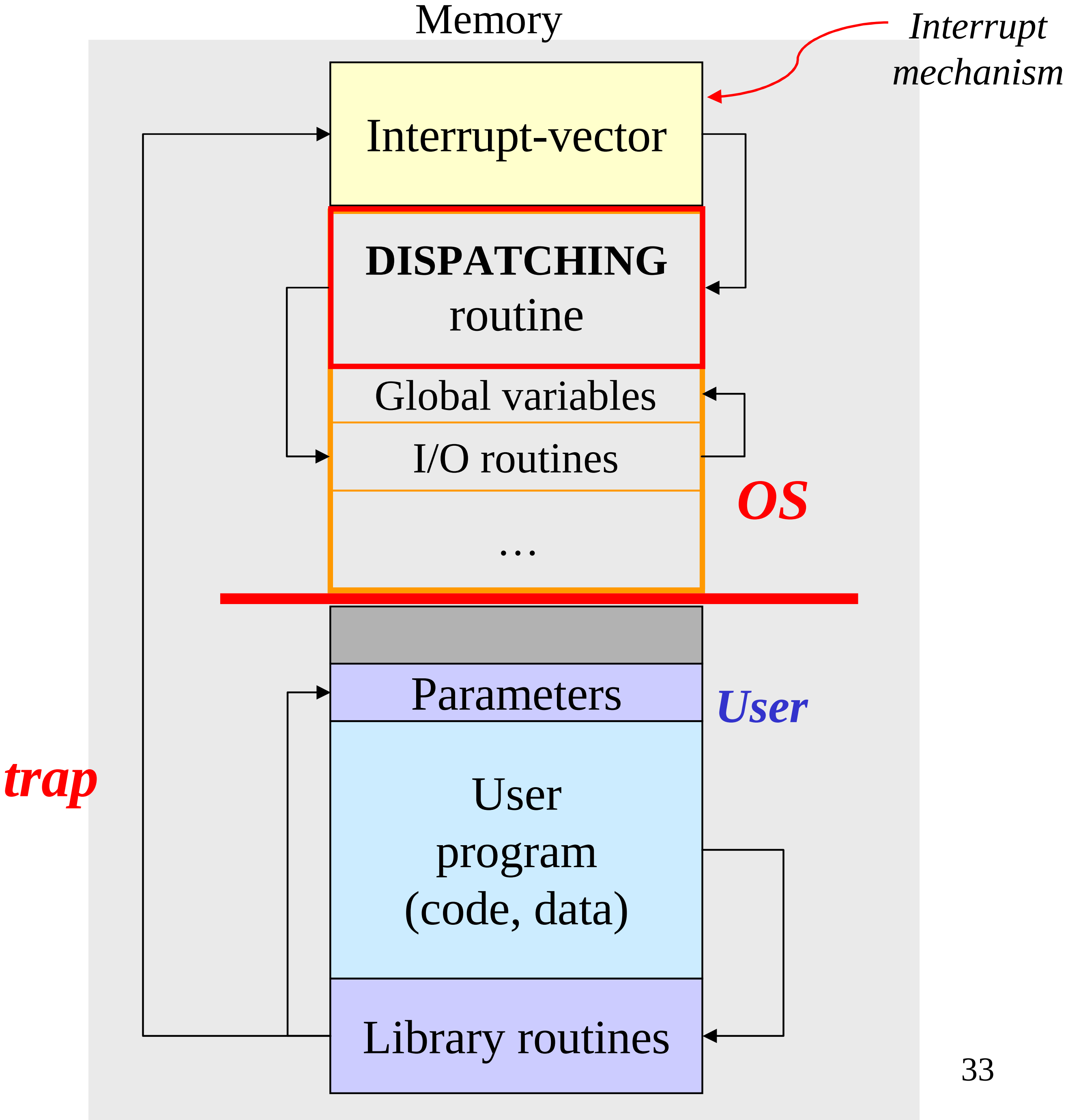
- System calls execute instructions that access the resources of the machine, e.g., I/O instructions that access devices

- These instructions must be executed in a controlled way, under the control of the OS!

Interrupt-vector based system cal

Library + technique for limiting the size of the interrupt-vector

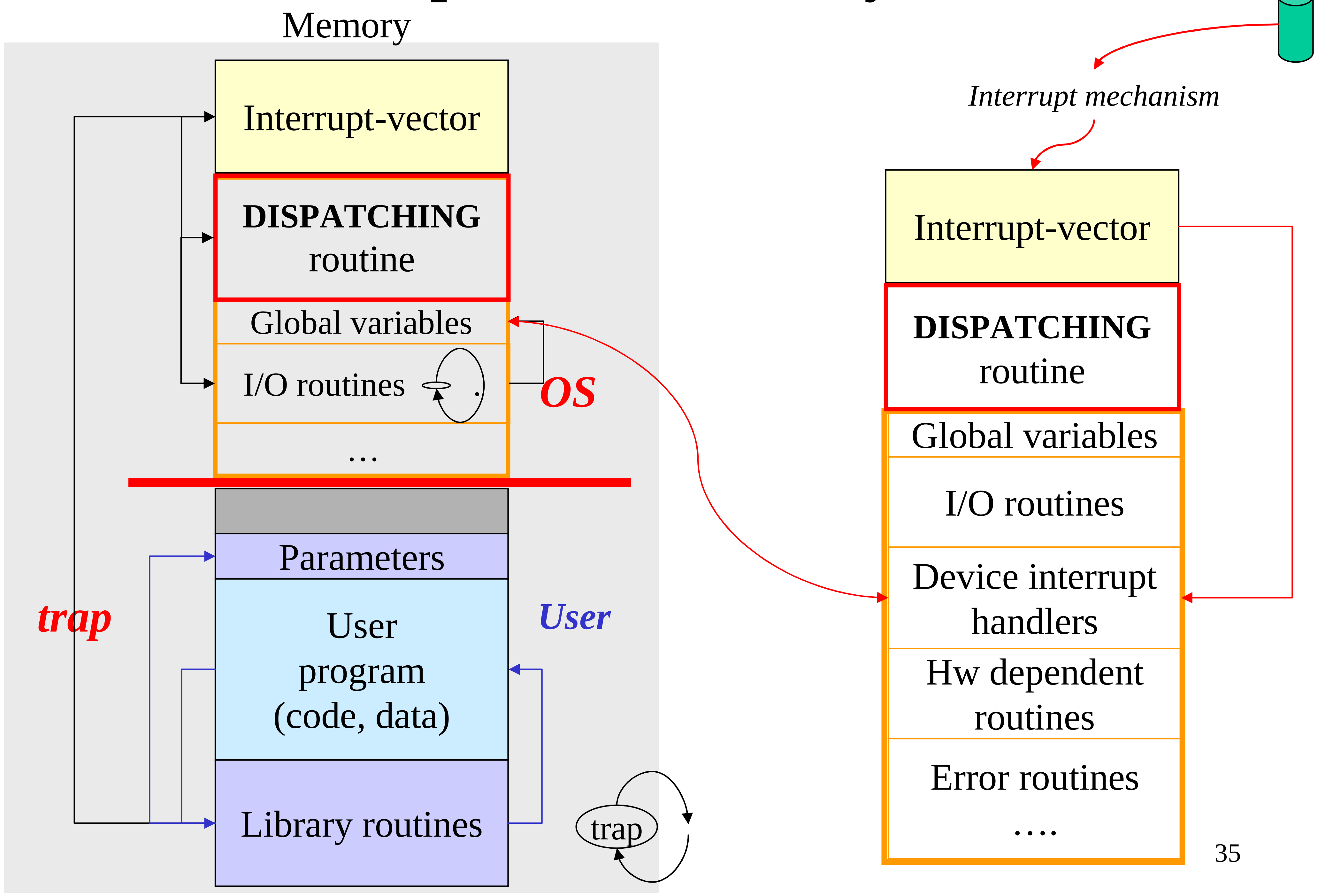
- Global variables
- I/O routines
- Device interrupt handlers
- Hw dependent routines
- Hw error routines
- Sw error routines
- Clock int. hand.



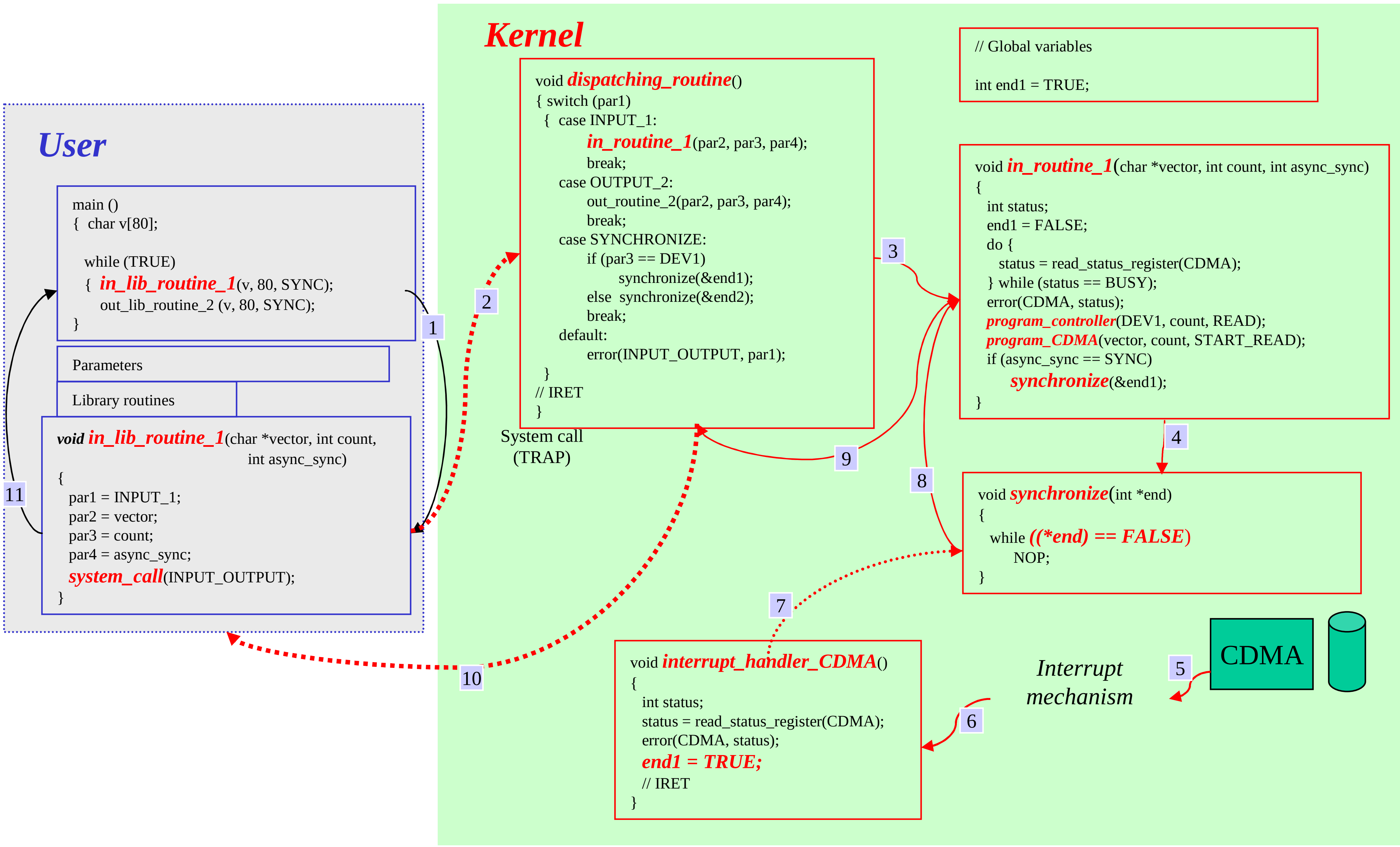
Abstraction techniques

- System calls are sometimes (if not always) wrapped by library functions
- Example: fopen() / fclose() – C abstractions
  - In Windows:
    - fopen() ⇨ CreateFile()
    - fclose() ⇨ CloseHandle()
  - In Unix:
    - fopen() ⇨ open()
    - fclose() ⇨ close()

Interrupt-vector based system call



Interrupt-vector based system call



Library routines

```
in_lib_routine_1(char *vector, int count,
                                        int async_sync)
{
   par1 = INPUT_1;   par2 = vector;
   par3 = count;   par4 = async_sync;
   system_call(INPUT_OUTPUT);
}
```

```
out_lib_routine_2(char *vector, int count,
                                        int async_sync)
{
   par1 = OUTPUT_2;   par2 = vector;
   par3 = count;   par4 = async_sync;
   system_call(INPUT_OUTPUT);
}
```

```
synchronize_lib_routine(int device)
{  par1 = SYNCHRONIZE;   par3 = device;
   system_call(INPUT_OUTPUT);
}
```

Variables for passing parameters

```
int par1, par3, par4; char *par2;
```

Dispatching routine

```
dispatching_routine()
{  switch (par1) {
      case INPUT_1:
         in_routine_1(par2, par3, par4); break;
      case OUTPUT_2:
         out_routine_2(par2, par3, par4); break;
      case SYNCHRONIZE:
         if (par3 == DEV1)
            synchronize(&end1);
         else synchronize (&end2);
         break;
      default:
         error(INPUT_OUTPUT, par1);
   }
}
```

User program (synchronous)

```
main()
{  char v[80];
   while (TRUE) {
      in_lib_routine_1(v, 80, SYNC);
      out_lib_routine_2(v, 80, SYNC);
   }
}
```

User program (synchronous)

```
main()
{
  char v[80];

  while (TRUE)
  {
    in_lib_routine_1(v, 80, SYNC);
    out_lib_routine_2(v, 80, SYNC);
  }
}
```

User program (asynchronous)

```
main()
{
  char v1[80], v2[80];

  while (TRUE)
  {
    in_lib_routine_1(v1, 80, SYNC);
    synchronize_lib_routine(DEV2);
    out_lib_routine_2(v1, 80, ASYNC);
    in_lib_routine_1(v2, 80, SYNC);
    synchronize_lib_routine(DEV2);
    out_lib_routine_2(v2, 80, ASYNC);
  }
}
```

System calls (1)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

System calls (2)

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

System calls (3)

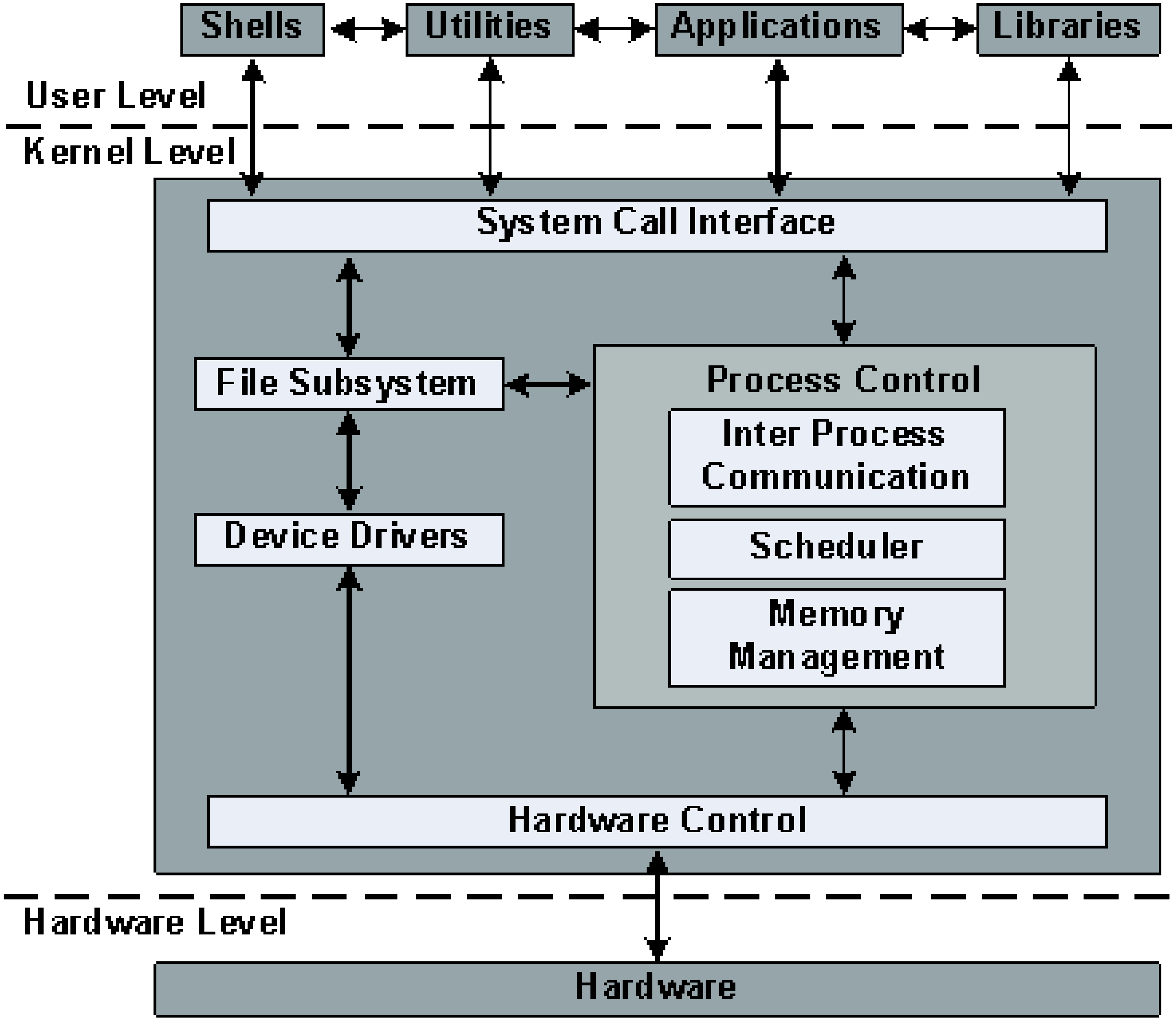
Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Unix - Architecture



Windows NT

- What is a system call in Windows?
  - A call to the API offered by a subsystem?

  or

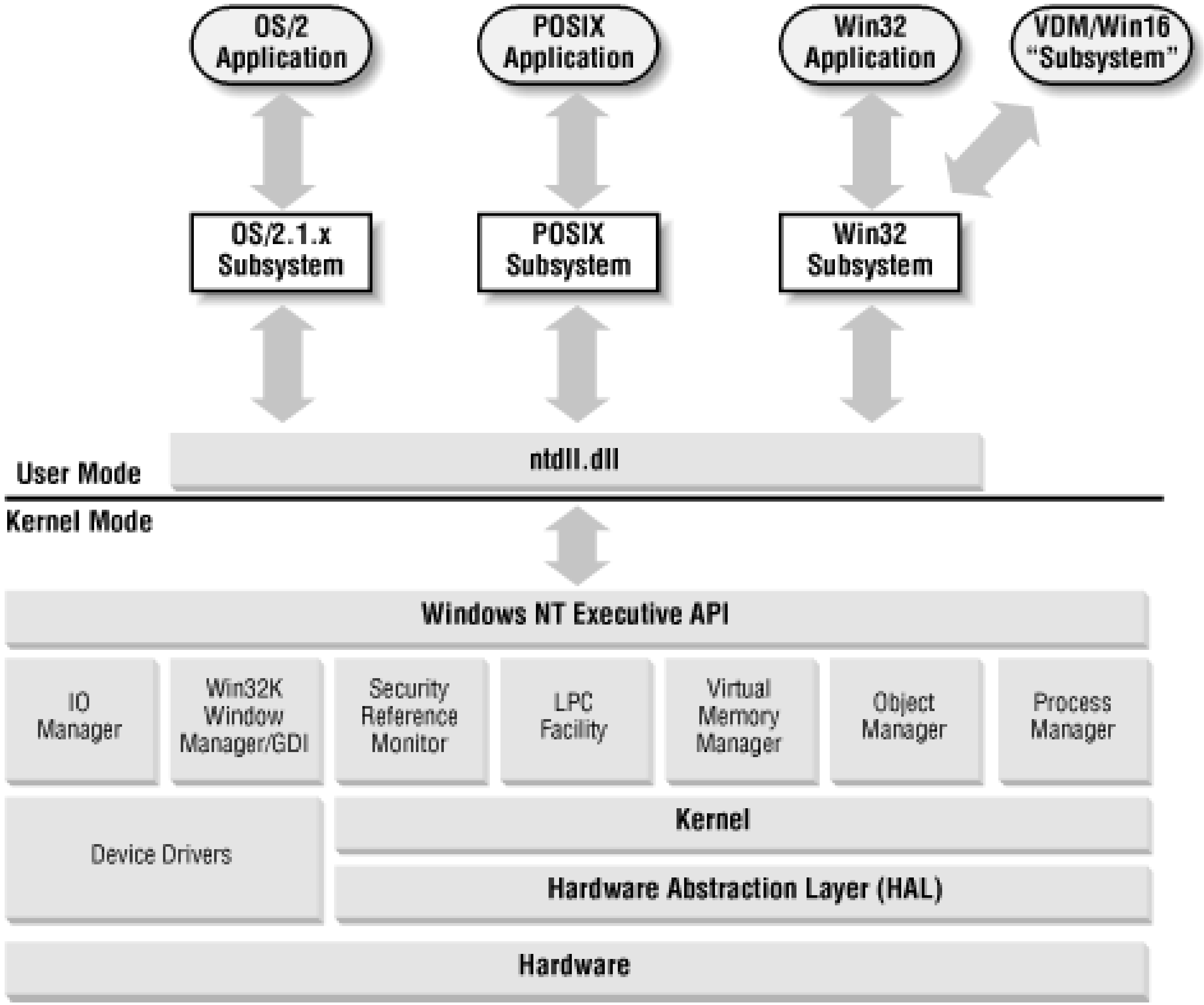
  - a call to the *Native* API?

Unix vs Windows NT

"Unix applications can call kernel functions, or *system calls*, directly. In Windows NT, applications call APIs that the OS environment to which they are coded (Win32, DOS, Windows 3.x, OS/2, POSIX) exports. The NT system-call interface, called the Native API, is hidden from programmers and largely undocumented (>1000 system calls)."

"The number of Unix system calls is around 200 to 300. The API that Unix applications write to is the Unix system-call interface, whereas the API that the majority of NT applications write to is the Win32 API, which translates Win32 APIs to Native APIs."

Windows NT - Architecture



APIs

- Some OS (e.g., Windows) do not publish the complete set of system calls
  - They publish a partial API (library)

- Others (e.g., Unix) publish the complete set of library and kernel functions
  - Both define a virtual machine

- Discuss pros and cons of these two approaches

- System call types:
  - Process management
    - Create/destroy
    - Suspend/activate
    - Inter-process communication
  - Memory management
    - Allocate/free memory
    - Increase/decrease memory
  - File management
    - create/remove/rename
    - open/close/read/write/move
    - change attributes
  - I/O routines
  - Other functions
    - get/set date and time, alarms
    - generate diagnostics
    - GUI

    …

Example

```
#include <stdio.h>
#include <time.h>

int main() {
   unsigned long t;

   while (1) {
      t = time(0);
      printf("time(0) = %ld -> %s\n", t, ctime(&t));
      sleep(1);
   }
}
```

Example

```
[acaf0266@acpt00 acaf0266]$ ls -l
total 16
-rwxr-xr-x    1 acaf0266 acaf        11908 feb 27 17:25 clock
-rw-r--r--    1 acaf0266 acaf          183 feb 27 15:17 clock.c

[acaf0266@acpt00 acaf0266]$ clock
time(0) = 1172596555 -> Tue Feb 27 18:15:55 2007
time(0) = 1172596556 -> Tue Feb 27 18:15:56 2007
time(0) = 1172596557 -> Tue Feb 27 18:15:57 2007
time(0) = 1172596558 -> Tue Feb 27 18:15:58 2007
time(0) = 1172596559 -> Tue Feb 27 18:15:59 2007
time(0) = 1172596560 -> Tue Feb 27 18:16:00 2007
time(0) = 1172596561 -> Tue Feb 27 18:16:01 2007
```