

**PREPARED BY**  
**AHNAF TAHMID, CSEDU**

**OS FUNCTIONS**

Operating systems have two main functions.

- Providing abstractions to user programs
- Managing the computer's resources

But the interface between user programs and the operating system is primarily about dealing with the abstractions.

**OS STRUCTURES**

**MONOLITHIC VS MICROKERNEL**

- A microkernel is a kernel type that implements an operating system by providing methods, including low-level address space management, IPC, and thread management. On the other hand, a monolithic kernel is a type of kernel in which the complete OS runs in the kernel space.

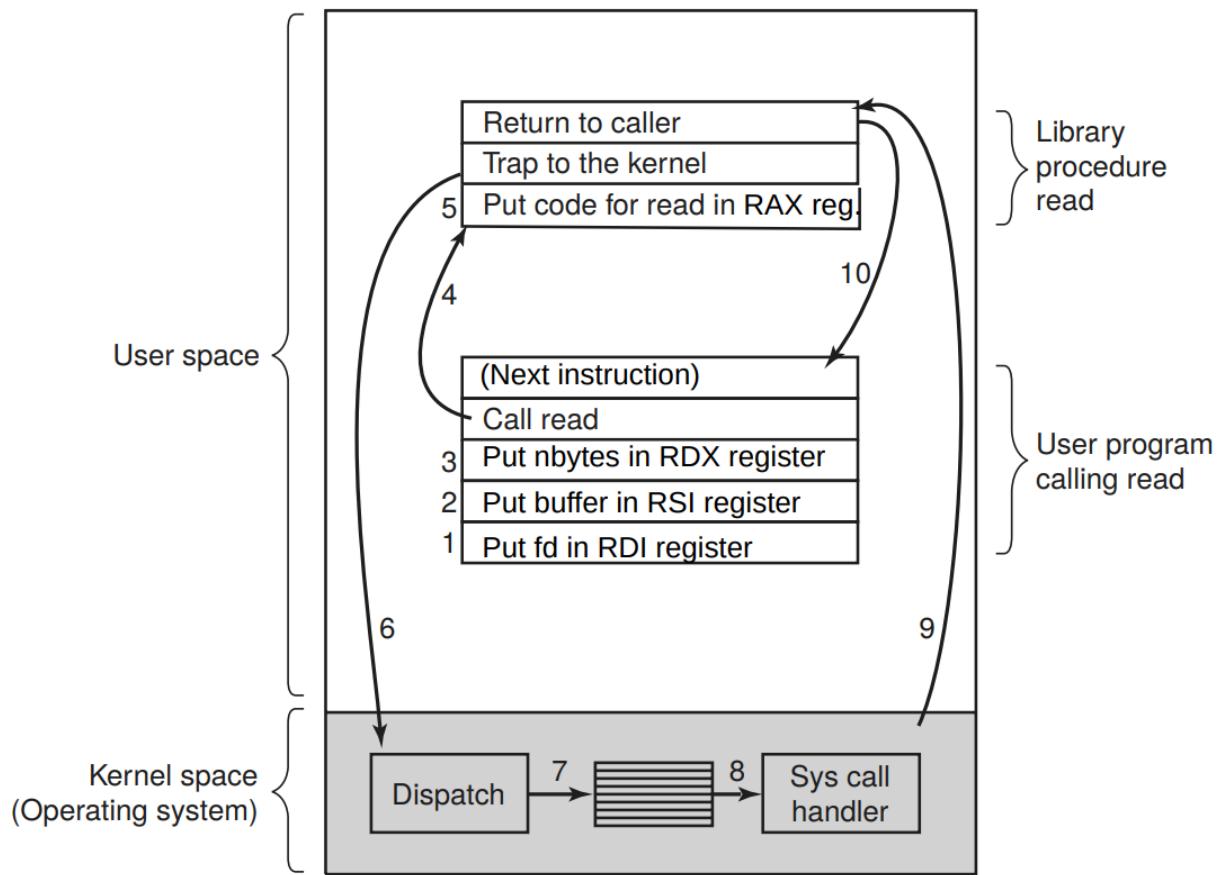
- The microkernel runs user and kernel services in different address spaces. On the other hand, the monolithic kernel runs both kernel and user services in the same address space. In microkernels, only essential processes like IPC, memory management, and scheduling take place in kernel space.
- The execution of the microkernel is slower because communication between the system's application and hardware is established by message passing. On the other hand, the execution of the monolithic kernel is faster because the system call establishes the communication of the system's application and hardware.
- Microkernels use the messaging queues to achieve IPC. On the other hand, monolithic kernels use sockets and signals to achieve IPC.

- The microkernel size is smaller than the monolithic kernel because only the kernel services run in the kernel address space. On the other hand, the monolithic kernel size is larger because both user and kernel services run in the same address space.
  - The microkernels are more secure than the monolithic kernels because the operating system is unchanged if a service fails in a microkernel. On the other hand, if a service fails in a monolithic kernel, the entire system fails.
- 
- The microkernel is simple to extend as new services are added in user address space, which is separate from kernel space, and thus the kernel doesn't need to be updated. On the other hand, the complete kernel must be updated if a new service is used in a monolithic kernel.
  - Microkernel designing needs less code that leads to fewer errors. In contrast, the monolithic kernel requires more code that leads to more errors.

---

## SYSTEM CALL

Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following system call. In a sense, making a system call is like making a special kind of procedure call - only system calls enter the kernel and procedure calls do not.



**Figure 1-17.** The 10 steps in making the system call `read(fd, buffer, nbytes)`.

The library procedure typically puts the system call number in a place where the operating system expects it, such as the RAX register. Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel.

### 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

### 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

**Figure 1-18.** Some of the major POSIX system calls. The return code *s* is *-1* if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

## COMPARISON BETWEEN THE TRAP INSTRUCTION AND THE PROCEDURE-CALL INSTRUCTION :

- The trap instruction is actually fairly similar to the procedure-call instruction in the sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.
- Nevertheless, the trap instruction also differs from the procedure-call instruction in two fundamental ways.

First as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode.

Second, rather than giving a relative or absolute address where the procedure is located, the trap instruction cannot jump to an arbitrary address.

Depending on the architecture, either it jumps to a single fixed location or there is an 8bit field in the instruction giving the index into a table in memory containing jump addresses or equivalent.

---

## SHADOWED STACK POINTER

There are two stack pointers in Cortex M4.

- Main Stack Pointer is the default stack pointer. It is used in the Thread mode when the CONTROL bit [1] i.e SPSEL is 0 and it is used in the Handler mode always.
- Process Stack Pointer is used in Thread mode when the CONTROL bit [1] i.e SPSEL is 1

In applications without embedded OS, we can just ignore the PSP and use the MSP for all operations.

In system with embedded OS, the MSP is used by the exception handlers and the PSP is used by the application tasks.

---

## WHAT ARE THE BENEFITS OF USING TWO STACK POINTERS?

- **Improves system reliability:** If an application task encounters a problem that leads to a stack corruption, the stack used by the OS kernel and other tasks is still likely to be intact.
- **Stack Space Allocation:** The stack space for each task only needs to cover the maximum stack usage + one level of stack frame. Stack space needed for the ISR and nested interrupt handling is allocated in the main stack only.

Stack Frame = Maximum 9 words including padding without Floating Point Unit / Maximum 27 words with Floating Point Unit

- **MPU utilization:** An OS can utilize the MPU to define the stack region which an application task can use. If an application task has a stack overflow problem, MPU can trigger a MemManage fault exception and prevent the task from overwriting memory regions outside the allocated stack space for this task.
-

## HOW DO WE INITIALIZE A TASK?

- When an OS first starts in Thread Mode, it can use the SVC exception to enter into the Handler mode. Then it can create a stack frame in the process stack, and trigger an exception return that uses the PSP. When the stack frame is loaded, the application task is started.
- 

### SVC

SVC (Supervisor Call) is exception type 11 and has a programmable priority level.

- SVC exception is triggered by the SVC instruction.

### WHY CAN'T WE TRIGGER SVC USING NVIC → STIR?

- Although it is possible to trigger an interrupt using software by writing to NVIC, the behaviour is different. Interrupts are imprecise. It means that a number of instructions could be executed after setting

the pending status but before the interrupt actually takes place. On the other hand, SVC is precise. The SVC handler must execute after the SVC instruction, except when another higher-priority exception arrives at the same time.

WRITE CODE TO GENERATE AN SVC EXCEPTION,  
TO FIND OUT WHICH STACK WAS USED BY THE  
PROGRAM THAT EXECUTED THE SVC AND TO  
EXTRACT THE STACKED PC FROM THE STACK  
FRAME AND THE SVC NUMBER.

- SVC exception generation code:

```
SVC #0x3 ; Call SVC function 3
```

- Finding out which stack was used by the program that executed the SVC, stacked PC from the stack frame and the SVC number:

```
SVC_Handler
TST LR, #4 ; Test bit 2 of EXC_RETURN
```

```
ITE EQ
MRSEQ R0, MSP ; if 0, stacking used MSP, copy
to R0
MRSNE R0, PSP ; if 1, stacking used PSP, copy
to R0
LDR R0, [R0, #24] ; Get stacked PC from the
stack frame ; (stacked PC = address of
instruction after SVC)
LDRB R0, [R0, #-2] ; Get first byte of the SVC
instruction ; now the SVC number is in R0
```

When your SVC service needs to return a value, you need to return the value using the stack frame. Otherwise, the return value stored in the register bank would be overwritten during the unstacking operation of an exception return.

Because of the exception priority model, you cannot use SVC inside an SVC handler because the priority is the same as the current priority. Doing so will result in a Usage Fault exception. For the same reason, you cannot use SVC in the NMI handler or the HardFault handler.

---

## PENDSV

PendSV (Pended Service Call) is exception type 14 and has a programmable priority level.

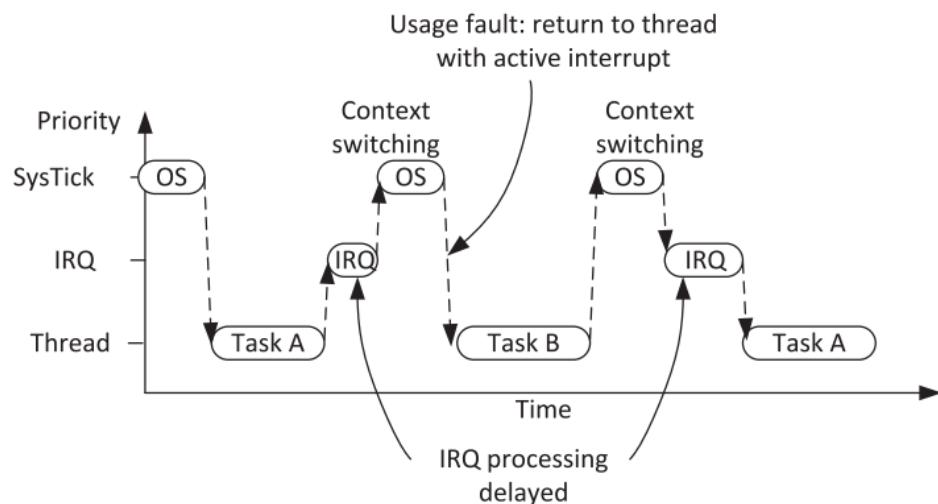
- The PendSV exception is triggered by setting its pending status by writing to the Interrupt Control and State Register (ICSR)
- Unlike the SVC exception, PendSV is not precise. So its pending status can be set inside a higher priority exception handler and executed when the higher priority handler finishes.
- We can schedule the PendSV exception handler to be executed after all other interrupt processing tasks are done, by making sure that the PendSV has the lowest exception priority level.

The execution of an OS kernel can be triggered by:

- Periodic Systick Exception
- Execution of SVC instruction from application tasks.

Problems if Systick Exception is used:

- If an interrupt request (IRQ) takes place before the SysTick exception, the SysTick exception might preempt the IRQ handler. In this case, the OS should not carry out the context switching. Otherwise, the IRQ handler process will be delayed.



**FIGURE 10.7**

Context switching during ISR execution can delay interrupt service

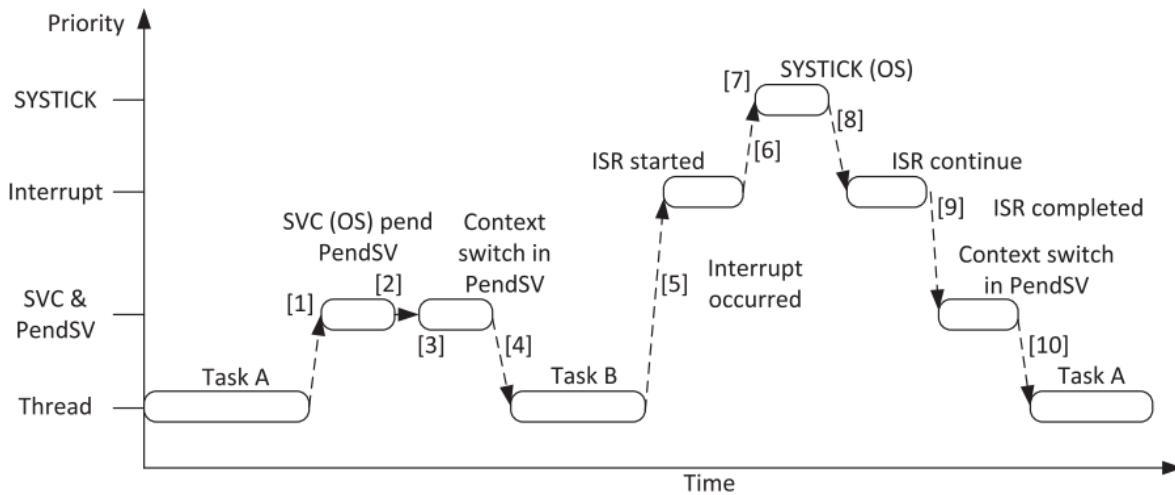
- If the OS attempts to return to the Thread mode with an active interrupt service running, it triggers a Usage fault exception.

- This problem is solved in some OS designs by not carrying out the context switching if an interrupt service is running. This can easily be done by checking the stacked xPSR from the stack frame or checking the interrupt active status registers in NVIC.

But this might affect the performance of the system, especially when an interrupt source keeps generating requests around the SysTick triggering time, which can prevent context switching from happening.

## Welcome boss PendSV:

- The PendSV exception solves the problem by delaying the context switching request until all other IRQ handlers have completed their processing.
- To do this, the PendSV is programmed as the lowest priority exception. If the OS decides that the context switching is needed, it sets the pending status of the PendSV and carries out the context switching within the PendSV exception.



**FIGURE 10.8**

Example context switching with PendSV

## CREATING TASK SCHEDULER THAT SWITCHES BETWEEN 3 TASKS IN A ROUND ROBIN ARRANGEMENT:

```
#include "stm32f4xx.h"
#define LED0 (1<<12)
#define LED1 (1<<13)
#define LED2 (1<<14)

/* Macros for word accesses */
#define HW32_REG(ADDRESS) (*((volatile
unsigned long *)(ADDRESS)))
```

```
/* Use Breakpoint to stop when error is
detected (KEIL MDK specific intrinsic) */ /*
it can be changed to while(1) XXif needed */
#define stop_cpu __breakpoint(0)
void LED_initialize(void); // Initialize LED
void task0(void); // Toggle LED0
void task1(void); // Toggle LED1
void task2(void); // Toggle LED2

// Event to tasks
volatile uint32_t systick_count=0;
// Stack for each task (8Kbytes each - 1024 x
8 bytes)
long long task0_stack[1024],task1_stack[1024],
task2_stack[1024];
// Data use by OS
uint32_t curr_task=0; // Current task
uint32_t next_task=1; // Next task
uint32_t PSP_array[3]; // Process Stack
Pointer for each task

int main(void)
{
SCB->CCR j= SCB_CCR_STKALIGN_Msk; // Enable
double word stack alignment
//(recommended in Cortex-M3 r1p1, default in
```

```
Cortex-M3 r2px and Cortex-M4)
LED_initialize();
// Starting the task scheduler

// Create stack frame for task0
PSP_array[0] = ((unsigned int) task0_stack)
+ (sizeof task0_stack) - 16*4;

HW32_REG((PSP_array[0] + (14<<2))) = (unsigned
long) task0; // initial Program Counter

HW32_REG((PSP_array[0] + (15<<2))) =
0x01000000; // initial xPSR

// Create stack frame for task1
PSP_array[1] = ((unsigned int) task1_stack)
+ (sizeof task1_stack) - 16*4;

HW32_REG((PSP_array[1] + (14<<2))) = (unsigned
long) task1; // initial Program Counter

HW32_REG((PSP_array[1] + (15<<2))) =
0x01000000; // initial xPSR

// Create stack frame for task2
```

```
PSP_array[2] = ((unsigned int) task2_stack)
+ (sizeof task2_stack) - 16*4;
HW32_REG((PSP_array[2] + (14<<2))) = (unsigned
long) task2; // initial Program Counter

HW32_REG((PSP_array[2] + (15<<2))) =
0x01000000; // initial xPSR

curr_task = 0; // Switch to task #0 (Current
task)

__set_PSP((PSP_array[curr_task] + 16*4)); // Set PSP to top of task 0 stack

NVIC_SetPriority(PendSV_IRQn, 0xFF); // Set PendSV to lowest possible priority

SysTick_Config(168000); // 1000 Hz SysTick
interrupt on 168MHz core clock

__set_CONTROL(0x3); // Switch to use Process
Stack, unprivileged state

__ISB(); // Execute ISB after changing CONTROL
```

```
(architectural
recommendation)
task0(); // Start task 0

while(1){
stop_cpu;// Should not be here
};

}

void task0(void) // Toggle LED #0
{
while (1) {
if (systick_count & 0x80)
{
    GPIOD->BSRRL = LED0;
} // Set LED 0
else {
    GPIOD->BSRRH = LED0;
} // Clear LED 0
};
}
// -----
-----
void task1(void) // Toggle LED #1
```

```
{  
while (1) {  
if (systick_count & 0x100)  
{  
    GPIOD->BSRRL = LED1;} // Set LED 1  
else {  
    GPIOD->BSRRH = LED1;  
} // Clear LED 1  
};  
}  
//  
-----  
-----  
void task2(void) // Toggle LED #2  
{  
while (1) {  
if (systick_count & 0x200)  
{  
    GPIOD->BSRRL = LED2;  
} // Set LED 2  
else  
{  
    GPIOD->BSRRH = LED2;  
} // Clear LED 2  
};  
}
```

```
_asm void PendSV_Handler(void)
{
    // Save current context
    MRS R0, PSP // Get current process stack
    pointer value

    STMDB R0!,{R4-R11} // Save R4 to R11 in task
    stack (8 regs)

    LDR R1,__cpp(&curr_task)

    LDR R2,[R1] // Get current task ID

    LDR R3,__cpp(&PSP_array)

    STR R0,[R3, R2, LSL #2] // Save PSP value into
    PSP_array

    // -----
    // Load next context
    LDR R4,__cpp(&next_task)
```

```
LDR R4,[R4] // Get next task ID

STR R4,[R1] // Set curr_task = next_task

LDR R0,[R3, R4, LSL #2] // Load PSP value from
PSP_array

LDMIA R0!,{R4-R11} // Load R4 to R11 from task
stack (8 regs)

MSR PSP, R0 // Set PSP to next task

BX LR // Return

ALIGN 4
}

void SysTick_Handler(void) // 1KHz
{
// Increment systick counter for LED blinking
Systick_count++;

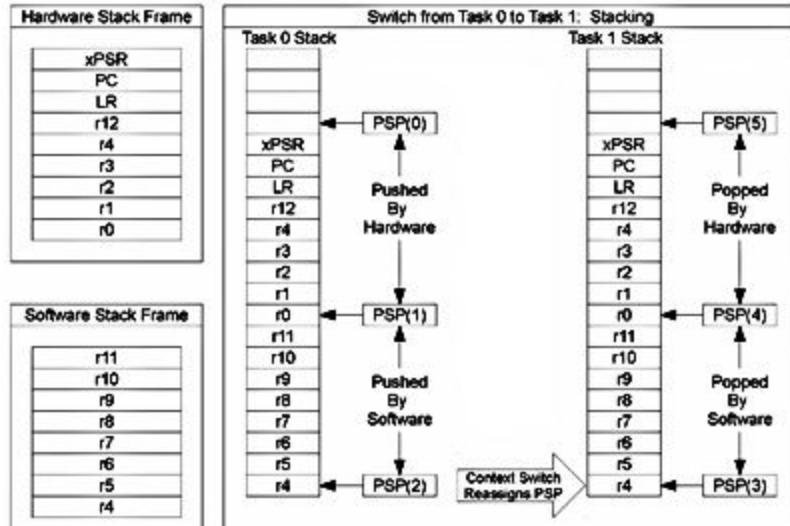
// Simple task round robin scheduler
switch(curr_task) {
case(0): next_task=1; break;
case(1): next_task=2; break;
```

```
case(2): next_task=0; break;
default: next_task=0;
stop_cpu;
break; // Should not be here
}

if (curr_task!=next_task)
{
// Context switching needed
SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk; // Set
PendSV to pending
}
return;
}
```

## CONTEXT SWITCH ER HATEKHORI

When an interrupt request is serviced on the CM4, some registers (see “Hardware Stack Frame” ) are automatically pushed by hardware onto the current stack - in this case, the process stack.



Software must save the remaining general purpose registers (see “Software Stack Frame”).

The following describes the chronological values assigned to the PSP when performing a context switch.

- PSP(0) : Just before an interrupt request is serviced
- PSP(1) : Just after an interrupt request is serviced
- PSP(2) : After the context switcher saves the necessary registers on the stack
- PSP(3) : After the context switcher reassigns the PSP to a new execution thread
- PSP(4) : After the context switcher loads the last known state of the new thread
- PSP(5) : After the interrupt request returns and execution of the new thread begins/resumes

Using the above details about the CM4 stacking and registers, the systems designer needs to create just three routines

- Context Switcher
- System Initializer
- Task Creator

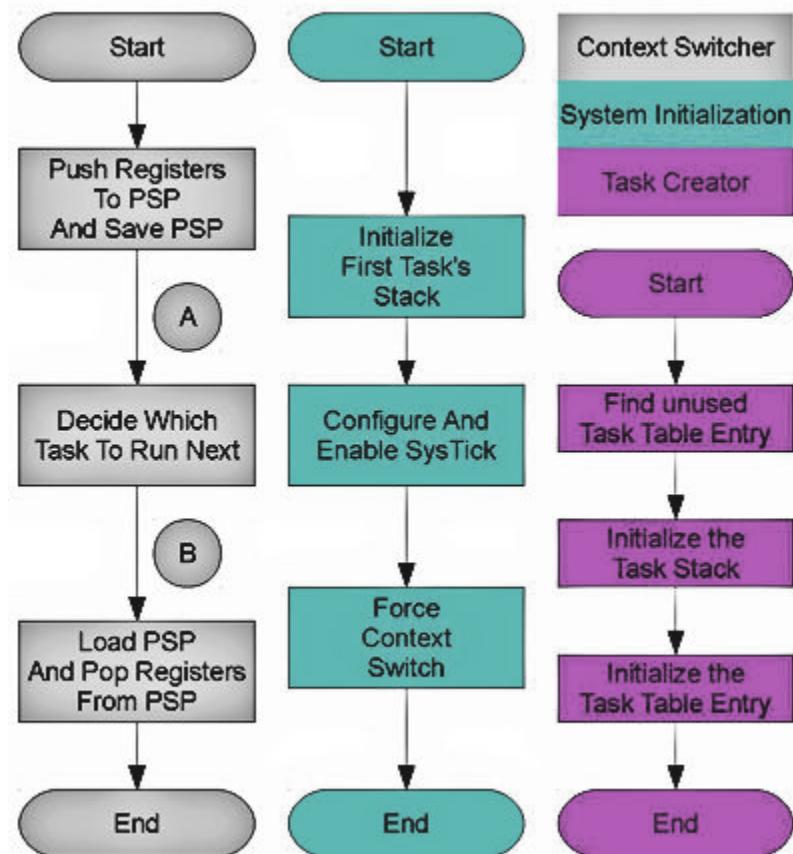
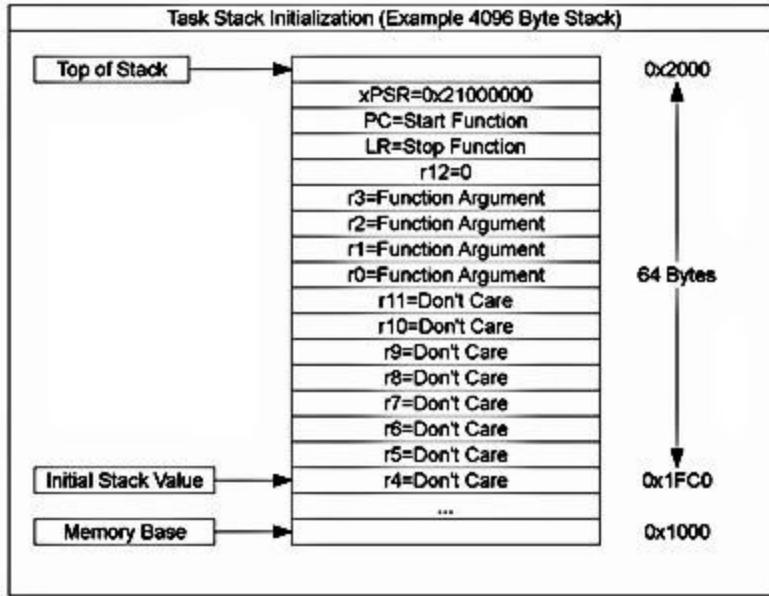


Fig: Software flow of system routines (Violet-Task Creator, Grey- Context switcher, Colorernamjanina-System Initializer)

The context switcher is invoked only through the SysTick and PendSV interrupt requests. It immediately pushes the software stack frame on to the process stack. It then saves the current value of PSP in the task table of the previously executing task. Next, it decides which task to execute. Once the next task is determined, the PSP is assigned the value of the new task's stack pointer retrieved from the task table. Lastly, immediately before returning from the interrupt, the software pops the software stack frame from the process stack. When the interrupt returns, the CM4 interrupt handling hardware pops the hardware stack frame and execution of the new task begins/resumes.

The system initializer initializes the first task's stack as well as the switching related hardware, namely SysTick and PendSV interrupts. The figure below shows the values of an initialized task stack - allocated dynamically or statically.



The hardware stack frame must be populated correctly in order for the task to start and stop properly.

The values of the software stack frame are ignored.

Nonetheless, the initial value of the stack pointer must point to the bottom of the software stack frame in order for the context switcher to load the software stack frame when switching to the task.

Once the task's stack is ready, the SysTick and PendSV interrupts are initialized.

The SysTick reload register is loaded with the desired value to set the round robin interrupt time. The interrupt interval

is calculated by multiplying the CPU frequency by the reload value. Once the SysTick timing is configured, the interrupt is enabled to start switching.

The PendSV interrupt is enabled by default, and no initialization is required. The PendSV interrupt is used to force a context switch after all initialization is complete.

After which, execution never returns to the task initialization function.

The initial task can create additional tasks using the task creator routine. Creating an additional task involves preparing the task's stack and configuring an entry in the task table.

The task creator routine finds an unused entry in the task table, populates the entry, and initializes the stack.

If there are no unused entries in the task table, no more tasks can be created unless the systems designer integrates a mechanism to dynamically resize the task table.

If an available entry is found, the task creator routine initializes the stack for the new task.

The calling function provides the memory location and size of the stack. This allows the caller to dynamically or statically allocate the stack.

The task switching routines can then be implemented in systems with or without a dynamic memory allocator. The stack initialization of a new task uses the same approach as initializing the initial task.

The hardware stack frame must be properly initialized while the software stack frame is allocated but can be left uninitialized.

---

## PROCESS

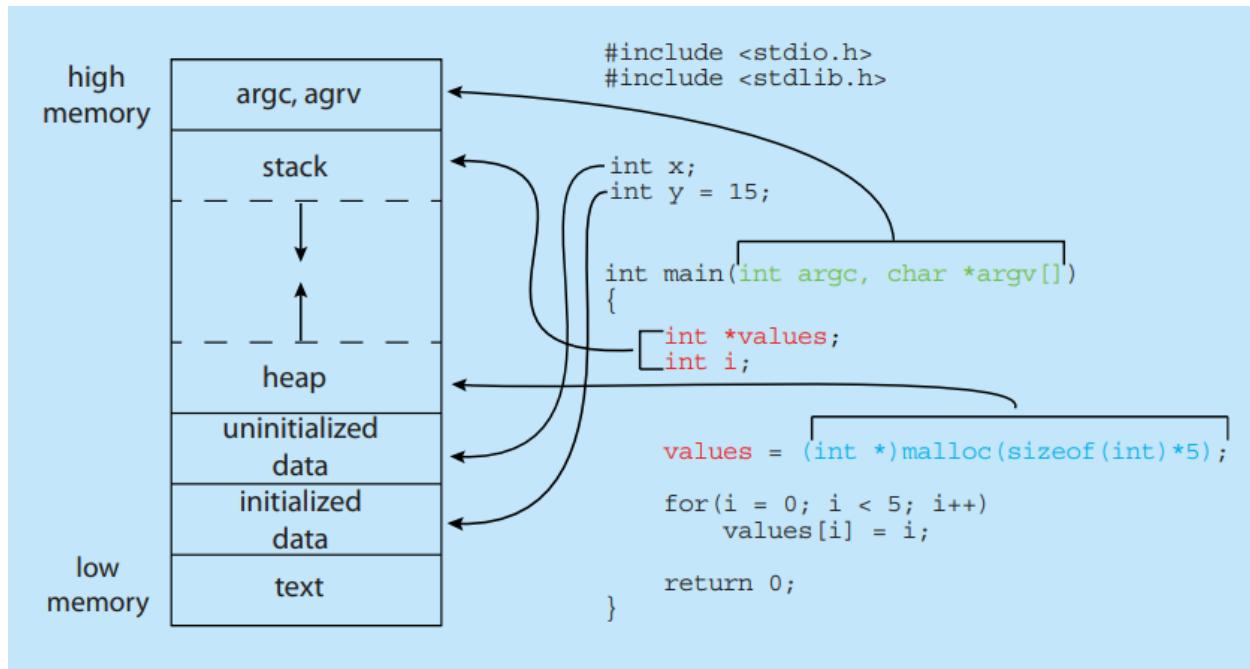
A process is just an instance of an executing program, including the current values of the program counter, registers and variables.

- Conceptually, each process has its own virtual CPU. In reality, of course, each real CPU switches back and

forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in pseudo parallel than to try to keep track of how each CPU switches from program to program. Switching rapidly back and forth like this is known as Multiprogramming.

Key idea is that a process is an activity of some kind. It has a program, input, output and a state. A single processor maybe shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that maybe stored on disk, not doing anything.

- It is worth noting that if a program is running twice, it counts as two processes.
- The fact that two process happen to be running the same program does not matter; they are distinct processes. The operating system maybe able to share the code between them so only one copy is in memory, but that is a technical detail that does not change the conceptual situation of two processes running.



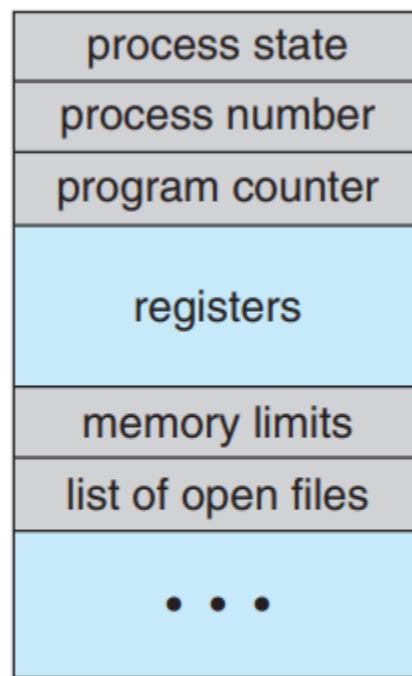
## PROCESS CONTROL BLOCK

Each process is represented in the operating system by a Process Control Block/ Task Control Block.

It contains many pieces of information associated with a specific process, including these

- Process State
- Program Counter
- CPU Registers
- CPU Scheduling Information
- Memory Management Information

- Accounting Information ( This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on)
- I/O status information



**Figure 3.3** Process control block (PCB).

---

# PROCESS CREATION

## When processes are created?

- **Boot Time :** When an operating system is booted, numerous foreground and background processes are created.
  - Foreground processes interact with users and perform work for them.
  - Background processes run in the background and are not associated with particular users, but instead have some specific function. For example, a background process maybe designed to accept incoming requests for web pages hosted on the machine, waking up when a request arrives to service the request.
  - Processes that stay in the background to handle some activity such as email, web pages, news, printing, and so on are called daemons.
- **Running process system call:** Often a running process will issue system calls to create one or more new processes to help it do its job.

- **User request:** In interactive systems users can start a program by typing a command or double clicking an icon. Taking either of these actions starts a new process and runs selected program in it.
- **Batch job initialization:** The last situation where processes are created applies only to batch systems found on large mainframes. Think of inventory management at the end of a day at a chain of stores - calculating what to order, analyzing per-store product popularity etc. Here users can submit batch jobs to the system (possibly remotely). When an operating system decides it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

### **How processes are created?**

In all the cases mentioned above, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch manager process. What that process does is execute a

system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly which program to run in it. To get the ball rolling, the very first process is hard crafted when the system is booted.

## Which system calls to use?

- In Unix there is only one system call to create a new process: fork. This call creates an exact clone of the calling process. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. Usually, the child process then executes execve or a similar system call to change its memory image and run a new program.
- In windows, a single Win32 function call, CreateProcess, handles both process creation and

loading the correct program into the new process. This call has 10 parameters, which include

- the program to be executed,
- the command line parameters to feed that program,
- various security attributes,
- bits that control whether open files are inherited,
- priority information,
- a specification of the window to be created for the process if any and
- a pointer to a structure in which information about the newly created process is returned to the caller.

In both unix and windows, after a process is created, the parent and child have their own distinct address space. If either process changes a word in its address space, the change is not visible to other process.

- In traditional unix, the child's initial address space is a copy of the parent's, but there are definitely two distinct address spaces involved; no writable memory is shared. Some unix implementations share the program text between the two since that cannot be modified. Alternatively the child may share all of the parent's memory but in that case the memory is shared copy-on-write, which means that whenever either of

the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area. Again no writable memory is shared. It is however possible for a newly created a process to share some of its creator's other resources, such as open files.

- In windows, the parent's and child's address spaces are different from the start.
- 

## PROCESS TERMINATION

### How a process gets terminated?

- **Normal Exit (Voluntary)** : Most process terminate because they have done their work.

For example, when a compiler has compiled the program given to it, the compiler executes system call to tell the operating system that it is finished. This call is exit in Unix and ExitProcess in Windows.

- **Fatal Error (Involuntary)** : Another reason for termination is that the process discovers a fatal error.

For example, if a user types the command cc foo.c to compile the program foo.c and no such file exists, the compiler simply announces this fact and exits.

- **Error exit (Voluntary)** : Another reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing non-existent memory, or dividing by zero.
  - **Killed by another process (Involuntary)** : Another reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In Unix, this call is kill. The corresponding Win32 function is TerminateProcess.
- 

## PROCESS STATES

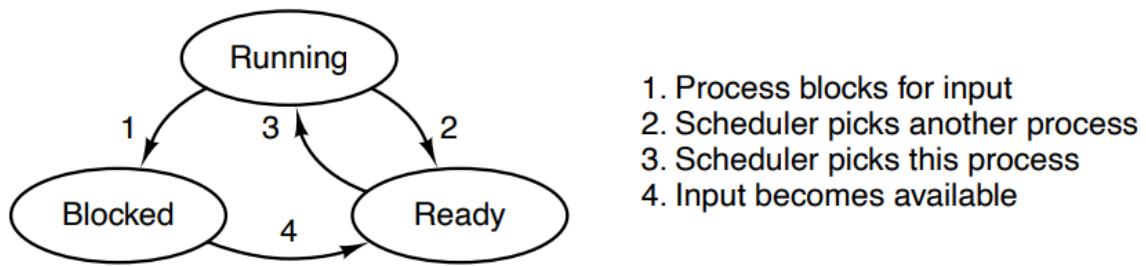
When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not

yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem. In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor).

Three states a process may be in:

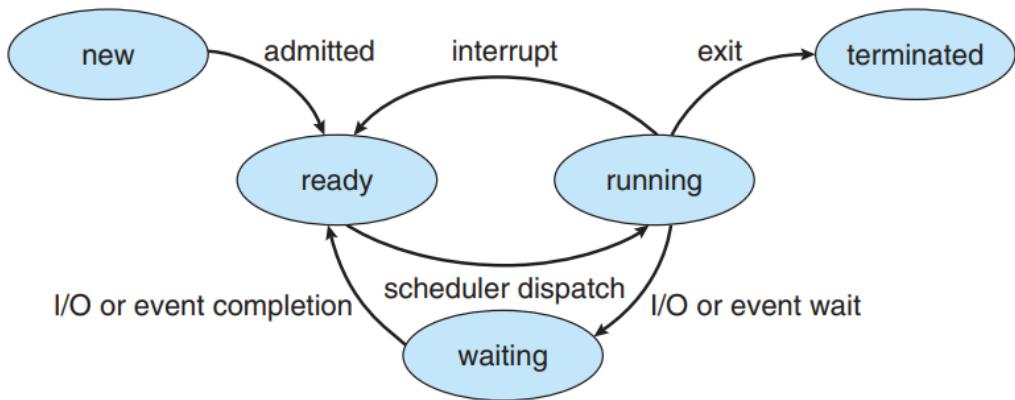
- **Running** : Actually using the CPU at that instant
- **Ready** : Runnable; temporarily stopped to let another process run
- **Blocked** : Unable to run until some external event happens

Logically the first two states are similar. In both cases, the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is fundamentally different from the first two in that the process cannot run, even if the CPU is idle and has nothing else to do.



**Figure 2-2.** A process can be in running, blocked, or ready state. Transitions between these states are as shown.

- Transition 1 occurs when the operating system discovers that a process cannot continue right now.
- Transition 2 occurs when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time.
- Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.
- Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens.



**Figure 3.2** Diagram of process state.

- New : The process is being created.
- Running: Instructions are being executed.
- Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready: The process is waiting to be assigned to a processor.
- Terminated: The process has finished execution.

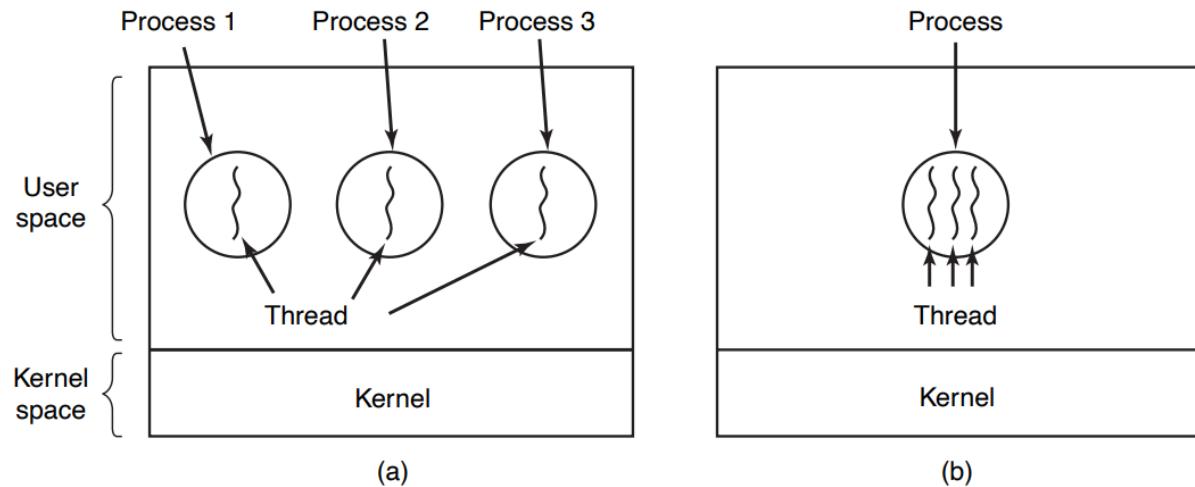
[Taken From Galvin's Book]

# THREADS

## Process vs Thread

A process has an address space that contains program next and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more.

The thread has a program counter associated with it that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It also has a stack, which contains the thread's execution history, one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together, threads are the entities scheduled for execution on the CPU.



**Figure 2-10.** (a) Three processes each with one thread. (b) One process with three threads.

The different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack.

<b>Per-process items</b>	<b>Per-thread items</b>
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

**Figure 2-11.** The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

It is important to realize that each thread has its own stack. Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished.

- For example, if procedure X calls procedure Y and Y calls procedure Z, then while Z is executing, the frames for X, Y and Z will all be on the stack. Each thread will generally call different procedures and thus have a different execution history. This is why each thread needs its own stack.

## THREAD CALLS

**Thread\_create :** A thread can create new threads by calling this library procedure. A parameter to `thread_create` specifies the name of a procedure for the new thread to run. It's not necessary to specify anything about the new thread's address space, since it automatically runs in the address space of the creating thread.

**Thread\_exit:** When a thread has finished its work, it can exit by calling this library procedure. It then vanishes and is no longer schedulable.

**Thread\_join :** One thread can wait for a specific thread to exit by calling this procedure. It blocks the calling thread until a specific thread has exited.

**Thread\_yield:** Allows a thread to voluntarily give up the CPU to let another thread run.

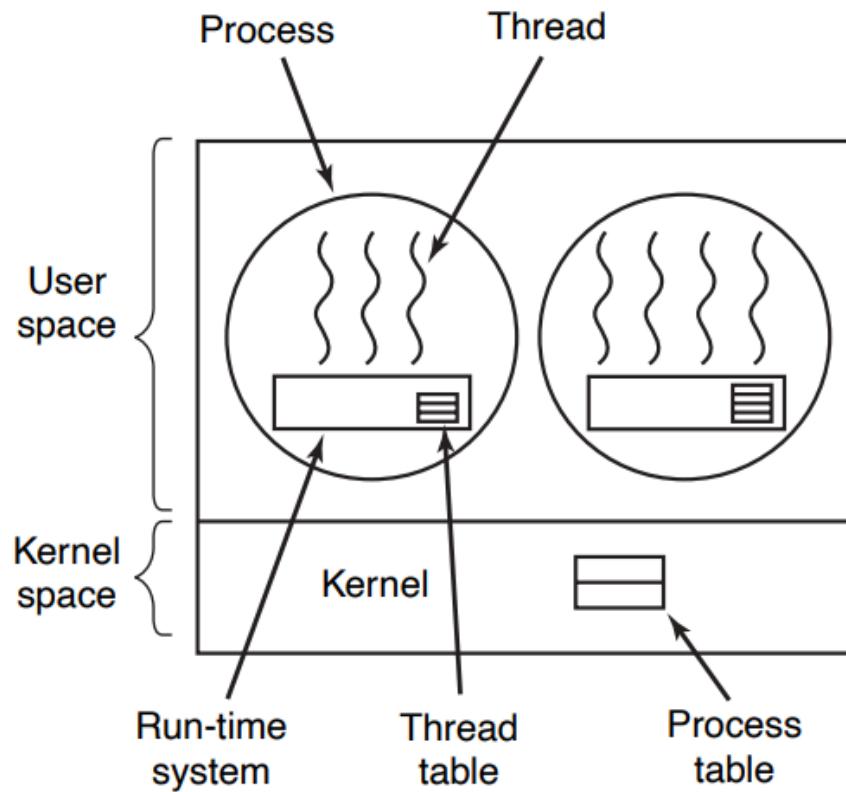
## **Why Threads?**

- Adds a new element : the ability for the parallel entities to share an address space and all of its data among themselves.
- Since they are lighter weight than processes, they are easier i.e faster to create and destroy.
- Yield no performance gain when all of them are CPU bound but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.
- Useful on systems with multiple CPUs, where real parallelism is possible.

- 1. Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
- 2. Resource sharing.** Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- 3. Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
- 4. Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

## IMPLEMENTING THREADS IN USER SPACE

When threads are managed in user space, each process needs its own private THREAD TABLE to keep track of the threads in that process. This table is analogous to the kernel's process table except that it keeps track only of the per thread properties such as each thread's program counter, stack pointer, registers, state and so forth. The thread table is managed by the runtime system.



## ADVANTAGES OF USER LEVEL THREADS

- A user level threads package can be implemented on an operating system that does not support threads.
- When a thread is finished running for the moment, for instance, when it calls `thread_yield`, the code of `thread_yield` saves the thread's information in the thread table and then calls the thread scheduler to pick another thread to run. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. There is no need for a trap, a context switch, flushing of caches and so on. This makes thread scheduling very fast.
- They allow each process to have its own customized scheduling algorithm.
- They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there is a very large number of threads.

## DISADVANTAGE OF USER LEVEL THREADS

**Problem of how blocking system calls are implemented:** If a thread makes a blocking system call, the process along with all internal threads blocks.

- Can use WRAPPERS as a work around. For example, the library procedure read() can be replaced with a new one that first does a select call and then does the read call only if it is safe i.e will not block. If the read call will block, the call is not made. Instead, another thread is run. This approach requires rewriting parts of the system call library and is inefficient and inelegant.
- Another workaround would be that the system calls could all be changed to non blocking system calls but requiring changes to the operating system is unattractive.

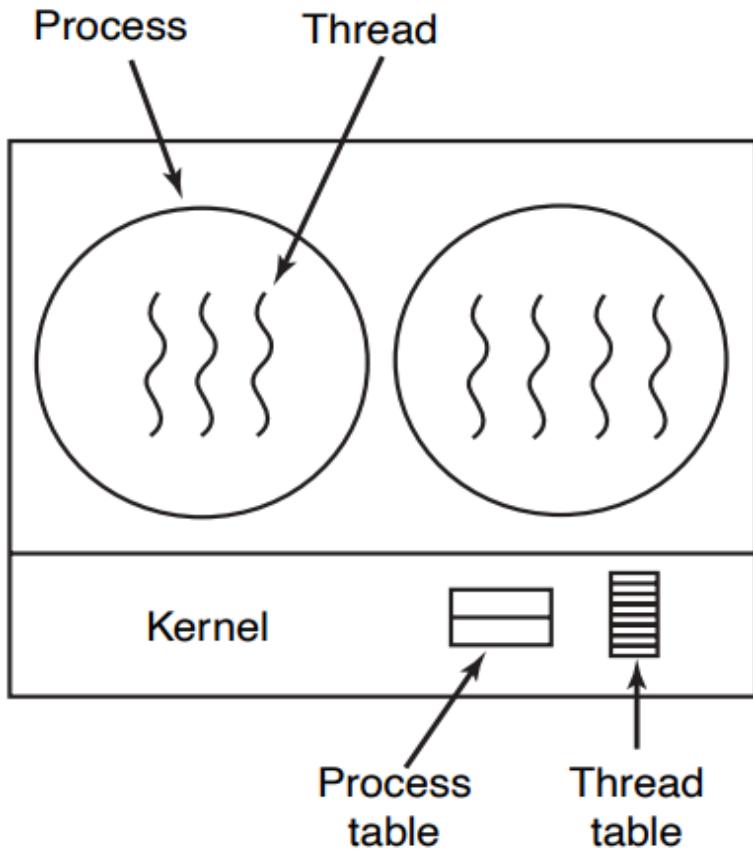
**Problem of page faults:** If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process

until the disk I/O is complete, even though other threads might be runnable.

**Problem of threads running forever:** If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule processes in round robin fashion. Unless a thread exits the runtime system of its own free will, the scheduler will never run.

## IMPLEMENTING THREADS IN THE KERNEL

There is now no need for a run time system or thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.



The kernel thread table holds each thread's registers, state, and other information. The kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a runtime system procedure. When a thread blocks, the kernel may choose to run either another thread from the same process or a thread from a different process.

Kernel threads do not require any new, non blocking system calls. If one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads and if so, run one of them while waiting for the required page to be brought in from the disk.

Main disadvantage is that the cost of a system call is substantial, so if thread operations like creation, terminations etc) are common, much more overhead will be incurred.

## PRACTICE PROBLEMS ON THREADS

- ❖ Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.
- (1) Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return.

- (2) Another example is a "shell" program such as the C-shell or Korn shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

❖ Which of the following components of program state are shared across threads in a multithreaded process?

- a. Register values
- b. Heap memory
- c. Global variables
- d. Stack memory

- The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

❖ Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

- A multithreaded system comprising of multiple user-level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and

will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

❖ Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.

- a. The number of kernel threads allocated to the program is less than the number of processors.
- b. The number of kernel threads allocated to the program is equal to the number of processors.
- c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of userlevel threads.

When the number of kernel level threads is less than the number of processors, then some of the processors would remain idle since the scheduler maps only kernel threads to processors and not user level threads to processors.

When the number of kernel threads is exactly equal to the number of processors, then it is possible that all of the processors might be utilized simultaneously. However, when a kernel thread blocks inside the kernel (due to page fault or while invoking system calls) the corresponding processor would remain idle.

When there are more kernel threads than processors, a blocked kernel thread could be swapped out in favor of another kernel thread that is ready to execute, thereby increasing the utilization of the multiprocessor system.

- ❖ Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

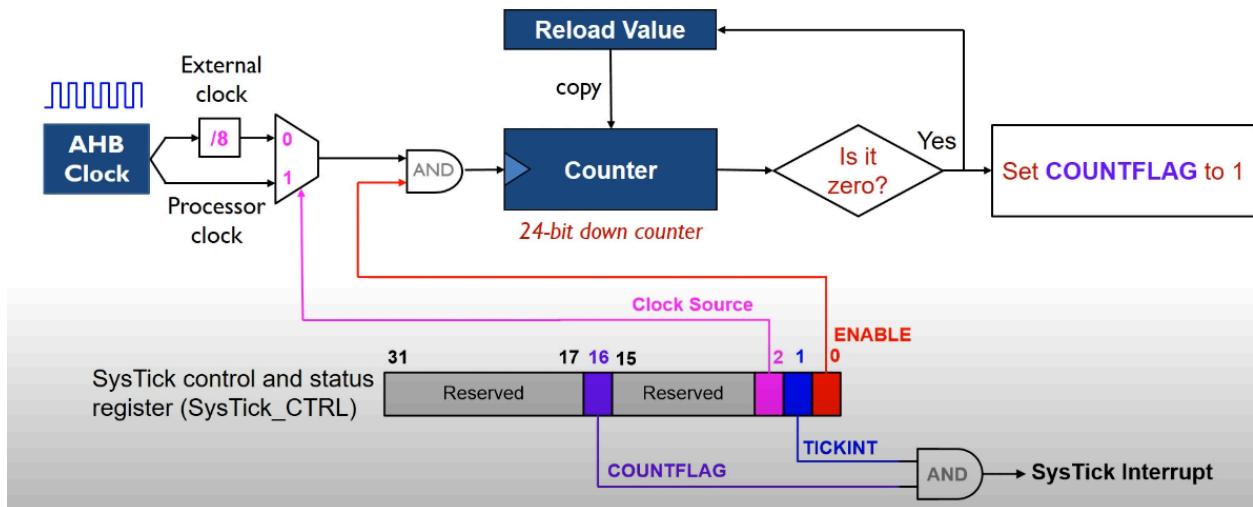
When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner.

A single threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system

events, a multi threaded solution would perform better even on a single processor system.

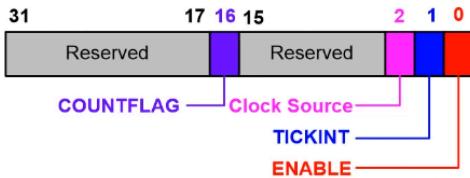
## SYSTICK TIMER IMPLEMENTATION

Diagram of System Timer (SysTick)



# Registers of System Timer

SysTick control and status register (SysTick\_CTRL)



SysTick reload value register (SysTick\_LOAD)



SysTick current value register (SysTick\_VAL)



SysTick calibration register (SysTick\_CALIB)



```
void SysTick_Initialize (uint32_t ticks) {
    SysTick->CTRL = 0;           // Disable SysTick
    SysTick->LOAD = ticks - 1;   // Set reload register
    // Set interrupt priority of SysTick to least urgency (i.e., largest priority value)
    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
    SysTick->VAL = 0;           // Reset the SysTick counter value
    // Select processor clock: 1 = processor clock; 0 = external clock
    SysTick->CTRL |= SysTick_CTRL_CLKSOURCE;
    // Enables SysTick interrupt, 1 = Enable, 0 = Disable
    SysTick->CTRL |= SysTick_CTRL_TICKINT;
    // Enable SysTick
    SysTick->CTRL |= SysTick_CTRL_ENABLE;
}
```

---

```

volatile int32_t TimeDelay;

int main (void {
    SysTick_Initialize(1000); // Interrupt period = 1000 cycles
    Delay(100);             // Delay 100 ticks
    ...
}

void SysTick_Handler (void) { // SysTick interrupt service routine
    if (TimeDelay > 0)      // Prevent it from being negative
        TimeDelay--;         // TimeDelay is a global volatile variable
}

void Delay (uint32_t nTime) {
    // nTime: specifies the delay time length
    TimeDelay = nTime;       // TimeDelay must be declared as volatile
    while(TimeDelay != 0);   // Busy wait
}

```

---

$$\text{Time interval between two SysTick interrupts} = \\ (\text{RELOAD} + 1) \times \text{Source\_Clock\_Period}$$

If 100 clock periods between two SysTick interrupts needed, then RELOAD = 99

Suppose clock source = 80MHz

Goal: Systick Interval = 10ms

What is RELOAD value?

$$\text{RELOAD} = (10 \times 10^{-3}) \times (80 \times 10^6) - 1$$

## CORTEX M4 CORE REGISTERS

### **PRIMASK**

When set it blocks all exceptions and interrupts apart from NMI and HardFault. It raises the current exception level to 0 which is the highest level for a programmable exception/interrupt.

To disable all interrupts

- MOVS R0, #1
- MSR PRIMASK, R0

To allow all interrupts

- MOVS R0, #0
- MSR PRIMASK, R0

### **FAULTMASK**

It also blocks HardFault exception. It raises the current exception level to -1.

To disable all interrupts

- MOVS R0, #1
- MSR FAULTMASK, R0

To allow all interrupts

- MOVS R0, #0
- MSR FAULTMASK, R0

## **BASEPRI**

It masks exceptions or interrupts based on priority level.  
Most M3/M4 have 8 or 16 programmable exception priority level.

When BASEPRI is set to 0, it is disabled. When it is set to a non-zero value, it blocks exception that have the same or lower priority level, while still allowing exception with a higher priority level to be accepted by the processor.

To disable all interrupts with priority 0x60-0xFF

- MOVS R0, #0x60
- MSR BASEPRI, R0

To allow all interrupts

- MOVS R0, #0x0
- MSR BASEPRI, R0

## **CONTROL REGISTER**

Control Register defines

- Selection of stack pointer ( Main Stack Pointer or Process Stack Pointer)
- Access level in Thread Mode ( Privileged or unprivileged)

After reset, the control register is 0. This means the Thread mode uses the Main stack pointer as stack pointer and Thread mode has privileged access level. Programs in privileged Thread mode can switch the stack pointer selection or switch unprivileged access level by writing to CONTROL register.

- nPRIV (CONTROL[0]) : 0 if privileged access level & Thread mode , 1 if unprivileged access level & Thread mode [ Handler mode always in Privileged Access level]
- SPSEL (CONTROL[1]) : 0 if Thread mode MSP and 1 if Thread mode PSP (MSP is used in Handler mode always)
- FPCA (CONTROL[2]) : 0 if FPU has not been used in the current context and no need to save floating point registers. 1 if FPU is used in the current context and need to save the floating point registers.

To access the CONTROL register in assembly, the MRS & MSR instructions are used:

```
MRS r0, CONTROL; Read CONTROL register into R0  
MSR CONTROL, r0; Write R0 into CONTROL  
register
```

To detect if the current execution level is privileged by checking the value of IPSR and CONTROL:

```
int in_privileged(void)
{
    if(__get_IPSR()!=0) return 1;
    else if ((__get_CONTROL() & 0x1)==0)
return 1;
    else return 0;
}
```

## INTERRUPT MANAGEMENT

```
void __NVIC_SetPriority(IRQn_TypeDef IRQn, uint32_t priority)
{
    if (IRQn >= 0)
    {
        NVIC->IP[IRQn] = (uint8_t)((priority << 4));
    }
    else
    {
        SCB->SHPR[(IRQn & 15) - 4] = (uint8_t)(priority << 4);
    }
}

uint32_t __NVIC_GetPriority(IRQn_TypeDef IRQn)
{
```

```

    if (IRQn >= 0)
    {
        return (NVIC->IP[IRQn] >> 4);
    }
    else
    {
        return (SCB->SHPR[(IRQn & 15) - 4] >> 4);
    }
}

void __NVIC_EnableIRQ(IRQn_TypeDef IRQn)
{
    if (IRQn >= 0)
    {
        NVIC->ISER[IRQn / 32] |= (1 << (IRQn % 32));
    }
}

void __NVIC_DisableIRQ(IRQn_TypeDef IRQn)
{
    if (IRQn >= 0)
    {
        NVIC->ICER[IRQn / 32] |= (1 << (IRQn % 32));
    }
}

void __set_BASEPRI(uint32_t value)
{
    value = (value << 4);
    asm volatile("MSR BASEPRI, %0"
               :
               : "r"(value)
               : "memory");
    kprintf("BASEPRI disabled interrupt with priority lower than %d\n", (value
>> 4));
}

```

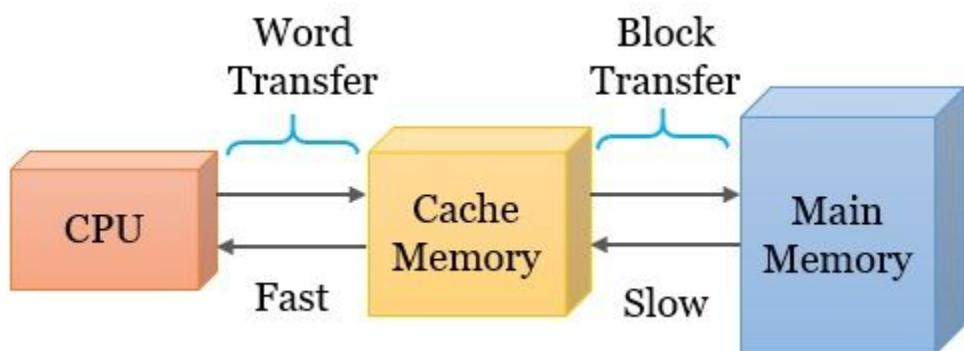
## CACHE MEMORY

Cache memory is an extremely fast memory type that acts as a buffer between RAM and CPU.

Cache memory holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is costlier than main memory or disk memory but more economical than CPU registers.

Cache memory is used to speed up and synchronize with a high speed CPU.



BASIS OF COMPARISON	CACHE MEMORY	MAIN MEMORY
Purpose	It is used to store frequently used data.	It holds the data that is currently being processed.
Access	Comparatively faster than main memory.	It is also the faster accessing memory.
Cost	More expensive than main memory.	Expensive memory.
Size	Comparatively smaller than main memory.	Larger than cache memory.
Types	L1, L2 and L3	SRAM and DRAM

## EFFECTIVE ACCESS TIME

$$T_{\text{eff}} = H \times T_1 + (1-H) \times (T_1+T_2)$$

$T_1$  = Access time of memory 1

$T_2$  = Access time of memory 2

H = Hit ratio of memory 1

Cache memory access time 1ns

Main memory access time 10ns

Hit rate of 95%

$$T_{\text{eff}} = 0.95 \times 1 \times 10^{-9} + 0.05 \times (1 \times 10^{-9} + 10 \times 10^{-9})$$

$$= 1.5 \times 10^{-9}$$

The access time of cache memory is 100 ns and that of the main memory is 1  $\mu$ sec. 80% of the memory requests are for reading and others are for write. The hit ratio for reading only accesses is 0.9. A write of the procedure is used. The average access time of the system for both read and write requests is ?

$$\begin{aligned} T_{\text{eff}} \text{ for reading} &= 0.9 \times 100 + (1-0.9) \times (1000+100) \\ &= 200 \text{ ns} \end{aligned}$$

$$\text{Average access time} = 200 \times 0.8 + 0.2 \times 1000 = 360 \text{ ns}$$

## PROCESS SYNCHRONIZATION (Galvin - Chapter 06)

A cooperating process is one that can affect or be affected by other processes executing in the system. They can either directly share a logical address space (both code and data) or be allowed to share data only through shared memory or message passing.

## Race Condition

The situation where several process access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a Race Condition.

## CRITICAL SECTION PROBLEM

Consider a system consisting of  $n$  processes. Each process has a segment of code, called a Critical Section, in which the process may be accessing and updating data that is shared with at least one other process.

- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section.

The Critical Section Problem is to design a protocol that the processes

- can use to synchronize their activity so as to cooperatively share data.
- Must request permission to enter its critical section

The section of code implementing this request is the entry section.

The critical section maybe followed by an exit section.

The remaining code is the remainder section.

```
while (true) {
```

*entry section*

critical section

*exit section*

remainder section

```
}
```

A solution to the critical section problem must satisfy the following three requirements:

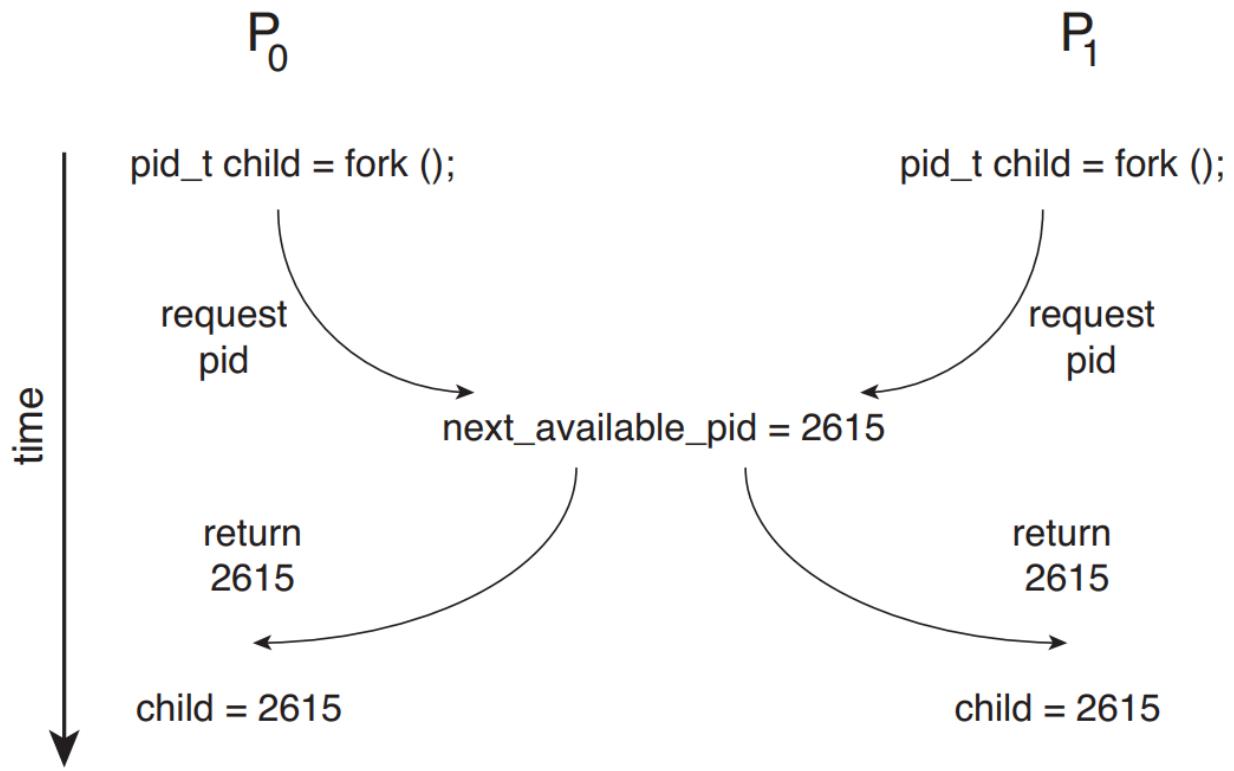
Mutual Exclusion: If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.

Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded Waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

## EXAMPLES OF RACE CONDITIONS

Two processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call. `fork()` returns the process identifier of the newly created process to the parent process.



**Figure 6.2** Race condition when assigning a pid.

In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

## SOLUTIONS TO CRITICAL SECTION PROBLEM

## Solution 01: Interrupt Prevention

Critical section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Problems of Solution 01: Unfortunately, solution 01 is not as feasible in multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

**WHY WOULD ANYONE FAVOUR A PREEMPTIVE KERNEL OVER A NON-PREEMPTIVE ONE?**

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrary long period before relinquishing the processor to waiting processes.

A preemptive kernel is more suitable for real time programming, as it will allow a real time process to preempt a process currently running in the kernel.

## SOLUTION 02: PETERSON'S SOLUTION

- Classic software-based solution

- Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures.

Peterson's solution requires two processes to share two data items:

- int turn;
- boolean flag[2];

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;
    /* critical section */

    flag[i] = false;
    /*remainder section */
}
```

---

**Figure 6.3** The structure of process  $P_i$  in Peterson's solution.

The variable turn indicates whose turn it is to enter its critical section. That is, if  $\text{turn}==i$ , then process  $P_i$  is allowed to execute in its critical section.

The flag array is used to indicate if a process is ready to enter its critical section. That is, if  $\text{flag}[i]$  is true,  $P_i$  is ready to enter its critical section.

Algorithm:

- To enter the critical section, process  $P_i$  first sets the  $\text{flag}[i]=\text{true}$  then sets  $\text{turn}=j$ , thereby asserting that if the other process wishes to enter the critical section it can do so.
- If both processes try to enter at the same time, turn will be set to both  $i$  and  $j$  at roughly the same time. Only one of the assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

## CORRECTNESS OF PETERSON'S SOLUTION

To prove that Peterson's solution is correct, we need to show that

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met.

## SEMAPHORES (Stallings, Chapter 5.4)

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specific place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals.

For signaling, special variables called SEMAPHOREs are used.

To transmit a signal via semaphore s, a process executes the primitive semSignal(s).

To receive a signal via semaphore  $s$ , a process executes the primitive `semWait(s)`; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only 3 operations are defined:

- A semaphore may be initialized to a non-negative integer value.
- The `semWait` operation decrements the semaphore value. If the value becomes negative, then the process executing the `semWait` is blocked. Otherwise, the process continues execution.
- The `semSignal` operation increments the semaphore value. If the resulting value is less than or equal to 0, then a process blocked by a `semWait` operation, if any, is unblocked.
- The `semWait` and `semSignal` primitives are assumed to ATOMIC.

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Fig: A definition of semaphore primitives

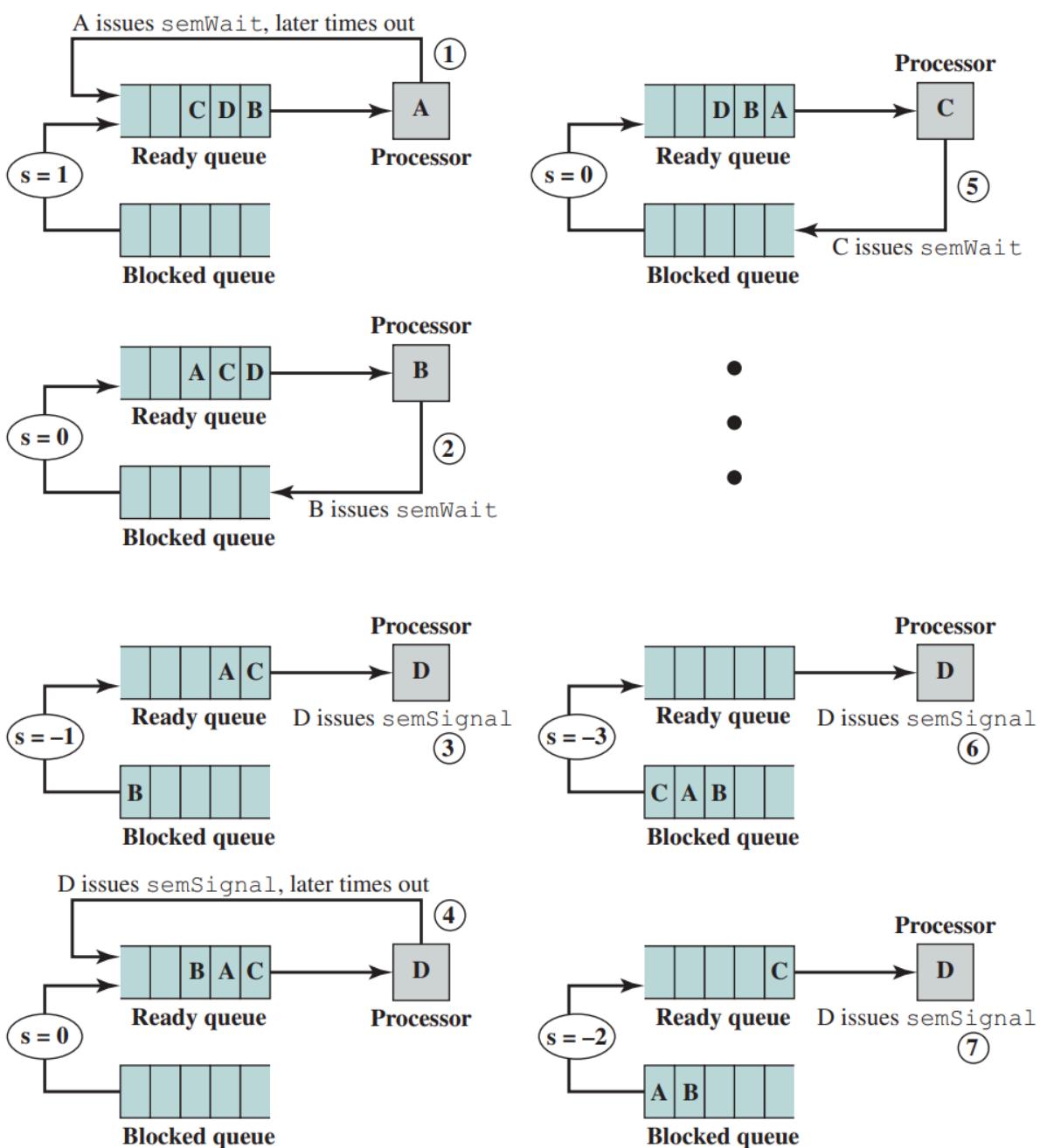


Fig: Example of Semaphore Mechanism

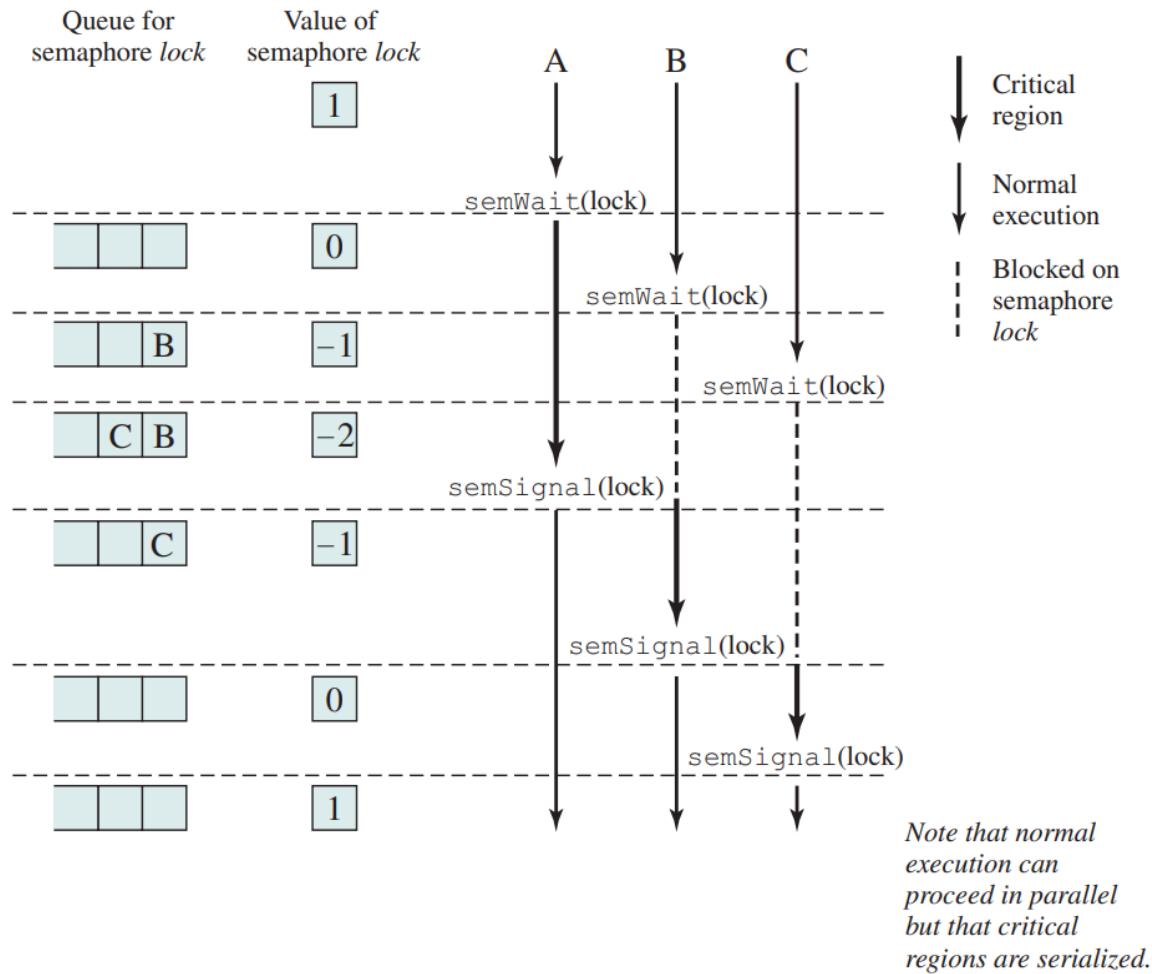
## Mutual Exclusion Using Semaphores

Consider  $n$  processes, identified in the array  $P(i)$ , all of which need access to the same resource. Each process has a critical section used to access the resource.

In each process, a `semWait(s)` is executed just before its critical section. If the value of  $s$  becomes negative, the process is blocked. If the value is 1, then it is decremented to 0 and the process immediately enters its critical section; because  $s$  is no longer positive, no other process will be able to enter its critical section.

The semaphore is initialized to 1. Thus, the first process that executes a `semWait` will be able to enter the critical section immediately, setting the value of  $s$  to 0. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of  $s$  to -1. Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of  $s$ . When the process that initially entered its critical section departs,  $s$  is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready

state. When it is next scheduled by the OS, it may enter the critical section.



```

/* program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}

```

Fig: Mutual Exclusion Using Semaphores

## Binary Semaphores

A binary semaphore may only take on the values 0 and 1, and can be defined by the following 3 operations:

- A binary semaphore may be initialized to 0 and 1.
- The semWaitB operation checks the semaphore value. If the value is 0, then the process executing the semWaitB is blocked. If the value is 1, then the value is changed to 0, and the process continues execution.
- The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a

semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to 1.

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

## Mutex vs Binary Semaphore

A mutex is a programming flag used to grab and release an object. When data are acquired that cannot be shared, or processing is started that cannot be performed simultaneously elsewhere in the system, the mutex is set to lock (typically zero), which blocks other attempts to use it. The mutex is set to unlock when the data are no longer needed or the routine is finished.

- A key difference between a mutex and a binary semaphore is that the process that locks the mutex(sets the value to zero) must be the one to unlock it(sets the value to one).
- In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

## PRODUCER-CONSUMER PROBLEM

### Requirements

- There are one or more producers generating some type of data and placing these in a buffer.
- There is a single consumer that is taking items out of the buffer one at a time.
- Only one agent (producer or consumer) may access the buffer at any one time.
- The producer must not try to add data into the buffer if it's full (we'll ignore this requirement while solving infinite buffer problem)
- The consumer won't try to remove data from an empty buffer.

Before going into the solutions, we need to understand, in general, what we do to protect the critical section.

Suppose there are two processes  $P_1$  &  $P_2$  and we want to make sure that both of them does not enter the critical section at the same time.

In order to do that, we initialize semaphore  $s=1$ .

For process  $P_1$  we run the code:

```
void P1()
{
    while (true) {

        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
```

For Process  $P_2$  we run the code:

```
void P2()
{
    while (true) {

        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
```

Let's test the mutual exclusion now:

Initially semaphore s=1

Process P<sub>1</sub> starts executing, performs semWait(s), decrements the s value to 0 and enters into the critical section.

Suppose process P<sub>2</sub> wants to enter into the critical section now. Process P<sub>2</sub> calls semWait(s) and decrements the value of s to -1 and the process P<sub>2</sub> will be blocked.

When P<sub>1</sub> comes out of the critical section, it makes call to semSignal(s) which increments the s value to 0 and then

change the state of  $P_2$  to be ready. And then  $P_2$  can easily enter into the critical section.

Solution to Infinite Buffer Producer-consumer problem using semaphore:

In the producer code, we have a critical section `append()` which we want to protect.

In the consumer code, we have a critical section `take()` which we want to protect.

So what must we do? Yeah, you are right. We use the semaphore `s` initialized to 1.

In the producer code, we do this:

```
semWait(s);  
append();  
semSignal(s);
```

In the consumer code, we do this:

```
semWait(s);  
take();  
semSignal(s);
```

But this isn't it, right? Yeah, we need to meet other requirements too.

First take the underflow condition, that is, consumer can't try to remove data from an empty buffer.

To meet this requirement, we take another semaphore  $n$  initialized to 0 (here,  $n$  stands for the number of full cells, and initially no cells are full).

So, before even the consumer tries to enter the critical section, we restrict it by calling `semWait(n)`. [ because  $n$  is initially 0, `semWait` will make it -1, then it will block the process ].

And very obviously, after producer appends the item, it has to call `semSignal(n)` to increment the number of cells which are full.

Since we are considering now the infinite buffer problem, we need not to worry about the overflow condition.

The full solution to the infinite buffer producer-consumer problem is given below:

```
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
```

Code: Solution to the infinite buffer producer-consumer problem

While solving the problem for bounded buffer, we need to check the overflow condition also, that is, the producer must not try to add data into the buffer if it's full.

In order to meet that requirement, we take another semaphore e initialized to sizeofbuffer [ here e stands for the number of empty cells in the buffer ; initially all the cells are empty and thus e equals to sizeofbuffer]

Suppose e at any moment equals to 0, that means the buffer is full or has no empty cells.

So, what we do is we restrict the producer by calling semWait(e) before it tries to enter into its critical section.

And very obviously, after consumer takes an item, it has to call semSignal(e) to increment the number of cells which are empty.

The full solution to the bounded buffer producer-consumer problem is given below:

```
semaphore n = 0, s = 1, e = sizeofbuffer;
void producer()
{
```

```
while (true) {
    produce();
    semWait(e)
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
```

Code: Solution to the bounded buffer producer-consumer problem

## **READERS/WRITERS PROBLEM**

The readers/writers problem is defined as follows: there is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

- Any number of readers may simultaneously read the file.
- Only one writer at a time may write to the file.
- If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another, and writers are processes that are required to exclude all other processes, readers and writers alike.

Two solutions may be possible:

### **READERS HAVE PRIORITY**

As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader

process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on wsem. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable readcount is used to keep track of the number of readers, and the semaphore x is used to assure that readcount is updated properly.

```
int readcount=0;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true){

        semWait (x);
        readcount++;
        if(readcount == 1)
            semWait (wsem);
        semSignal (x);

        READUNIT();

        semWait (x);
        readcount--;
    }
}
```

```

    if(readcount == 0)
        semSignal (wsem);
    semSignal (x);
}

void writer()
{
    while (true){
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}

```

## WRITERS HAVE PRIORITY

In the previous solution, readers have priority, that is, once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. Therefore, writers are subject to starvation.

So, we require a solution that guarantees no new readers are allowed access to the data area once at least one writer has declared a desire to write.

For writers, the following semaphores and variables are added to the ones already defined.

- A semaphore ‘rsem’ that inhibits all readers while there is at least one writer desiring access to the data area
- A variable ‘writecount’ that controls the setting of ‘rsem’
- A semaphore ‘y’ that controls the updating of writecount

For readers, one additional semaphore is needed. A long queue must not be allowed to build up on rsem; otherwise writers will not be able to jump the queue. Therefore, only one reader is allowed to queue on rsem, with any additional readers queueing on semaphore z, immediately before waiting on rsem.

Code (Writer Priority) :

```

/* program readersandwriters */
int readcount,writecount; semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1)
            semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer ()
{
    while (true){
        semWait (y);
        writecount++;
        if (writecount == 1)
            semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

```

[Na Bujhle 1](#)

[Na Bujhle 2](#)

## SLEEPING BARBER PROBLEM

The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:

There is a barber shop with one barber and a number of chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

Our solution uses three semaphores, customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), barbers, the number of barbers (0 or 1) who are idle, waiting for customers, and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. The reason for having to wait is that there is no way to read the current value of a semaphore. In this solution, a

customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

When the barber shows up for work in the morning, he executes the procedure `barber`, causing him to block on the semaphore `customers` because it is initially 0. The barber then goes to sleep. He stays asleep until the first customer shows up.

When a customer arrives, he executes the `customer`, starting by acquiring `mutex` to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released `mutex`. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases `mutex` and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, `waiting`. Then he does an `Up` on the semaphore `customers`, thus waking up the barber. At this point, the customer and the barber are both awake. When the customer releases `mutex`, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. There is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, a haircut is given. If not, the barber goes to sleep.

```
#define CHAIRS 5           /* # chairs for waiting customers */

typedef int semaphore;    /* use your imagination */

semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;    /* # of barbers waiting for customers */
semaphore mutex = 1;      /* for mutual exclusion */
int waiting = 0;          /* customer are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);    /* go to sleep if # of customers is 0 */
        down(&mutex);        /* acquire access to "waiting" */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);        /* one barber is now ready to cut hair */
        up(&mutex);          /* release 'waiting' */
        cut_hair();           /* cut hair (outside critical region */
    }
}

void customer(void)
{
    down(&mutex);          /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);    /* wake up barber if necessary */
        up(&mutex);          /* release access to 'waiting' */
        down(&barbers);    /* go to sleep if # of free barbers is 0 */
        get_haircut();       /* be seated and be served */
    } else {
        up(&mutex);          /* shop is full; do not wait */
    }
}
```

 Sleeping barber problem | OS | Lec-62 | Bhanu Priya  
(Don't watch this video please)

## DEADLOCK

(Galvin, Chapter 8.3 to 8.8)

Deadlock is a situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set.

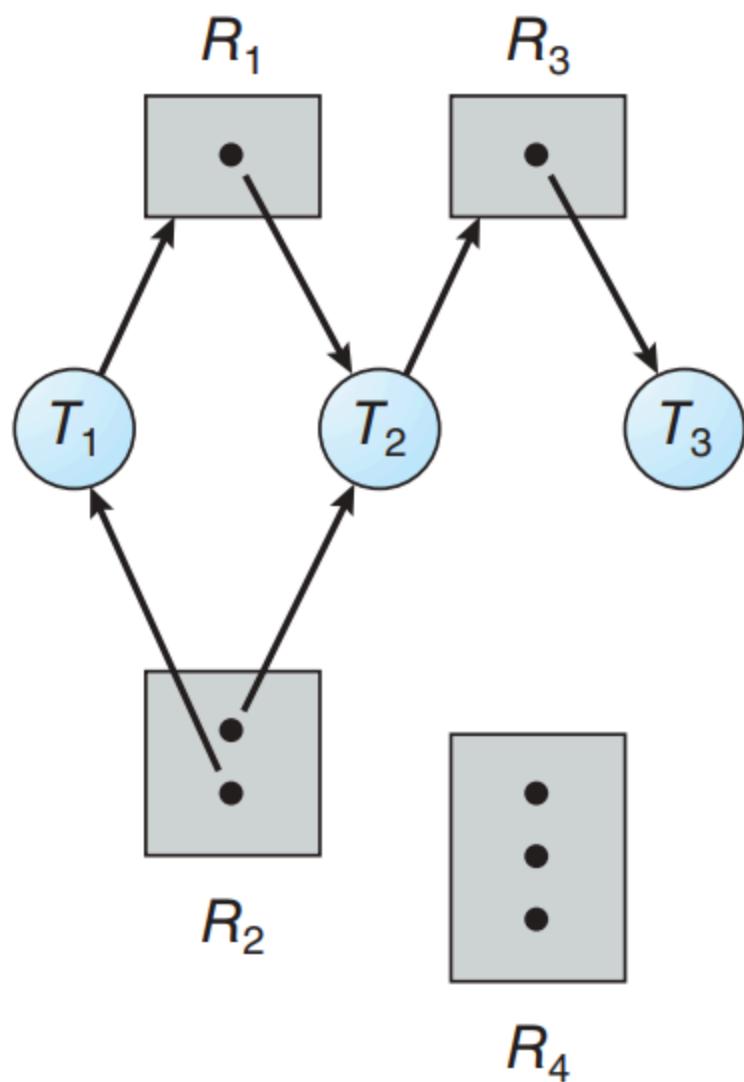
A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- Mutual Exclusion: At least one resource must be held in a non-shareable mode, that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
- Hold & Wait: A thread must be holding at least one resource and waiting to acquire additional resource that are currently being held by other threads.
- No Preemption: Resources cannot be preempted, that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.

- Circular Wait: A set  $\{T_0, T_1, \dots, T_n\}$  of waiting threads must exist such that  $T_0$  is waiting for a resource held by  $T_1$ ,  $T_1$  is waiting for a resource held by  $T_2$ ... $T_{n-1}$  is waiting for a resource held by  $T_n$  and  $T_n$  is waiting for a resource held by  $T_0$

All 4 conditions must hold for a deadlock to occur. The circular wait condition implies the Hold & Wait condition, so the 4 conditions are not completely independent, but it is useful to consider each condition separately.

## RESOURCE ALLOCATION GRAPH



**Figure 8.4** Resource-allocation graph.

The sets  $T$ ,  $R$ , and  $E$ :

- $T = \{T_1, T_2, T_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$

- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Resource instances:
  - One instance of resource type  $R_1$
  - Two instances of resource type  $R_2$
  - One instance of resource type  $R_3$
  - Three instances of resource type  $R_4$
- Thread states:
  - Thread  $T_1$  is holding an instance of resource type  $R_2$  and is waiting for an instance of resource type  $R_1$ .
  - Thread  $T_2$  is holding an instance of  $R_1$  and an instance of  $R_2$  and is waiting for an instance of  $R_3$ .
  - Thread  $T_3$  is holding an instance of  $R_3$ .

If the graph contains no cycles, then no thread in the system is deadlocked.

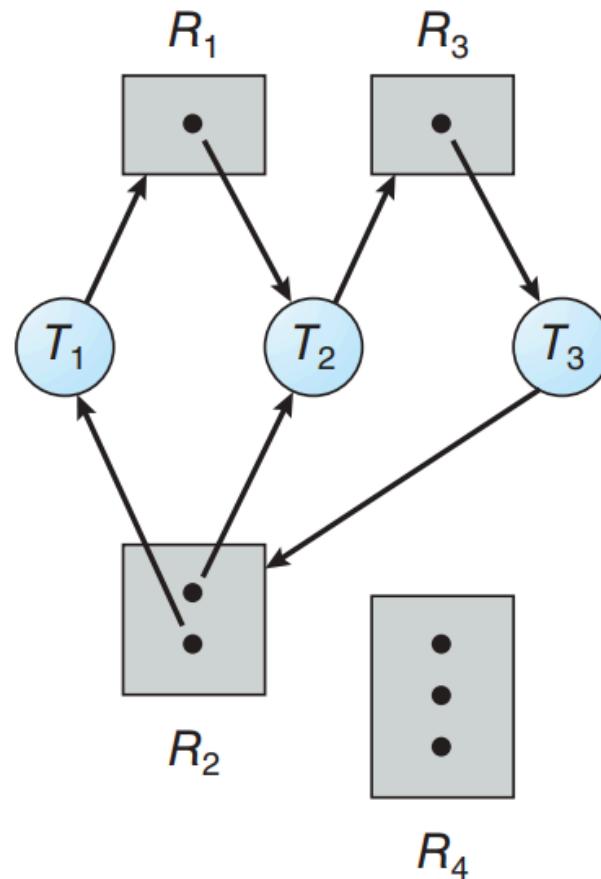
If the graph does contain a cycle, then a deadlock MAY exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

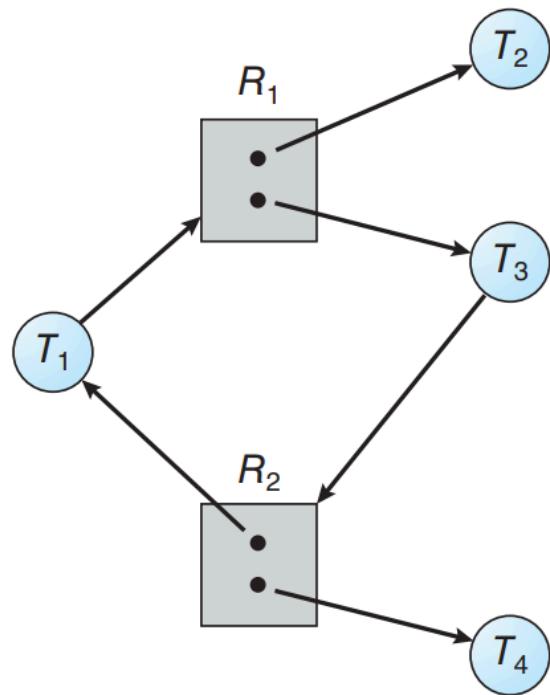
If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each thread involved in the cycle is deadlocked. In

this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.



**Figure 8.5** Resource-allocation graph with a deadlock.

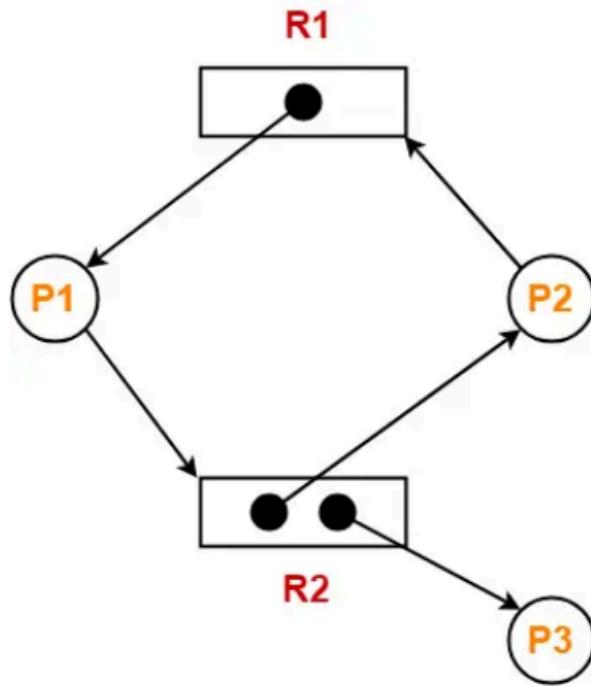


**Figure 8.6** Resource-allocation graph with a cycle but no deadlock.

## SOLVED EXAMPLES ON RESOURCE ALLOCATION GRAPH

Example 01:

Consider the resource allocation graph in the figure-



Find if the system is in a deadlock state otherwise find a safe sequence.

Solution:

- The given resource allocation graph is multi instance with a cycle contained in it.
- So, the system may or may not be in a deadlock state.

Using the given resource allocation graph, we have-

	Allocation		Need	
	R1	R2	R1	R2
<b>Process P1</b>	1	0	0	1
<b>Process P2</b>	0	1	1	0
<b>Process P3</b>	0	1	0	0

$$\text{Available} = [ R1 \ R2 ] = [ 0 \ 0 ]$$

- Since process P3 does not need any resource, so it executes
- After execution, process P3 releases its resources.

Then,

Available

$$= [ 0 \ 0 ] + [ 0 \ 1 ]$$

$$= [ 0 \ 1 ]$$

- With the instances available currently, only the requirement of the process P1 can be satisfied
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$\begin{aligned} &= [01] + [10] \\ &= [11] \end{aligned}$$

- With the instances available currently, the requirement of the process P2 can be satisfied
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

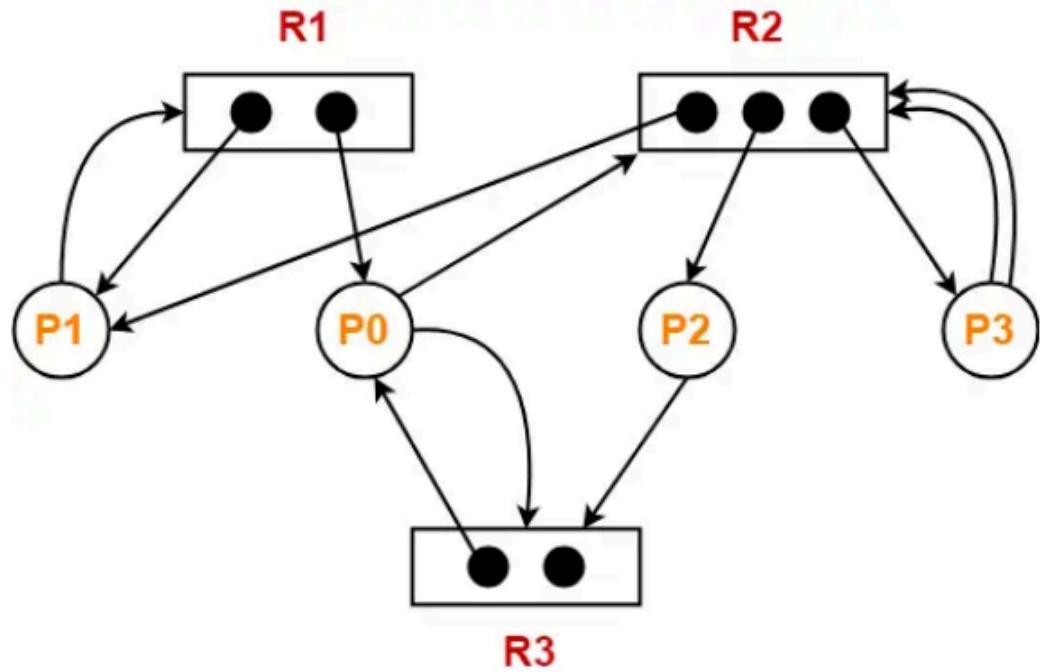
$$\begin{aligned} &= [11] + [01] \\ &= [12] \end{aligned}$$

Thus,

- There exists a safe sequence P3, P1, P2 in which all the processes can be executed.
- So, the system is in a safe state.

## Example 02:

Consider the resource allocation graph in the figure-



Find if the system is in a deadlock state otherwise find a safe sequence.

Solution:

- The given resource allocation graph is multi instance with a cycle contained in it.
- So, the system may or may not be in a deadlock state.

Using the given resource allocation graph, we have-

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
Process P0	1	0	1	0	1	1
Process P1	1	1	0	1	0	0
Process P2	0	1	0	0	0	1
Process P3	0	1	0	0	2	0

$$\text{Available} = [ R1 \ R2 \ R3 ] = [ 0 \ 0 \ 1 ]$$

- With the instances available currently, only the requirement of the process P2 can be satisfied
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [ 0 \ 0 \ 1 ] + [ 0 \ 1 \ 0 ]$$

$$= [ 0 \ 1 \ 1 ]$$

- With the instances available currently, only the requirement of the process P0 can be satisfied
- So, process P0 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$\begin{aligned} &= [011] + [101] \\ &= [112] \end{aligned}$$

- With the instances available currently, only the requirement of the process P1 can be satisfied
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$\begin{aligned} &= [112] + [110] \\ &= [222] \end{aligned}$$

- With the instances available currently, the requirement of the process P3 can be satisfied.
- So, process P3 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$\begin{aligned} &= [2 \ 2 \ 2] + [0 \ 1 \ 0] \\ &= [2 \ 3 \ 2] \end{aligned}$$

Thus,

- There exists a safe sequence P2, P0, P1, P3 in which all the processes can be executed.
- So, the system is in a safe state.

## METHODS FOR DEADLOCK HANDLING

- Just ignore the problem altogether (Ostrich Algorithm)
- Deadlock detection and recovery
- Deadlock Prevention
- Deadlock Avoidance

## THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all.

## Advantages:

- In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than prevention, avoidance, or detection and recovery methods, which must be used constantly and hence costly. Most operating systems including UNIX and WINDOWS use this approach.

## Disadvantages:

- In this case, we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened. The undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run; and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

## DEADLOCK AVOIDANCE

Deadlock avoidance requires that the operating system be given in advance additional information concerning which

resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.

## Safe State

- A state is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a safe sequence.
- A sequence of threads  $\langle T_1, T_2, \dots, T_n \rangle$  is a safe sequence for the current allocation state if, for each  $T_i$ , the resource requests that  $T_i$  can still make can be satisfied by the currently available resources + the resources held by all  $T_j$ , with  $j < i$ . In this situation, if the resources that  $T_i$  needs are not immediately available, then  $T_i$  can wait until all  $T_j$  have finished. When they have finished,  $T_i$  can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When  $T_i$

terminates,  $T_{i+1}$  can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs.

## SOLVED EXAMPLES ON BANKER'S ALGORITHM

Example 01:

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P <sub>0</sub>	0	0	1	2	0	0	1	2	1	5	2	0
P <sub>1</sub>	1	0	0	0	1	7	5	0				
P <sub>2</sub>	1	3	5	4	2	3	5	6				

P <sub>3</sub>	0	6	3	2	0	6	5	2								
P <sub>4</sub>	0	0	1	4	0	6	5	6								

What's the Need Matrix?

Is system in safe state? If yes, then find the safe sequence.

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P <sub>0</sub>	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P <sub>1</sub>	1	0	0	0	1	7	5	0					0	7	5	0
P <sub>2</sub>	1	3	5	4	2	3	5	6					1	0	0	2
P <sub>3</sub>	0	6	3	2	0	6	5	2					0	0	2	0
P <sub>4</sub>	0	0	1	4	0	6	5	6					0	6	4	2

Update 01: Available <1,5,2,0> P<sub>0</sub> can be executed.

Update 02: Available =  $\langle 1,5,2,0 \rangle + \langle 0,0,1,2 \rangle = \langle 1,5,3,2 \rangle$ ;  $P_1$  can't be executed, but  $P_2$  can be executed.

Update 03: Available  $\langle 1,5,3,2 \rangle + \langle 1,3,5,4 \rangle = \langle 2,8,8,6 \rangle$   $P_3$  can be executed.

Update 04: Available  $\langle 2,8,8,6 \rangle + \langle 0,6,3,2 \rangle = \langle 2,14,11,8 \rangle$ ;  $P_4$  can be executed.

Update 05: Available  $\langle 2,14,11,8 \rangle + \langle 0,0,1,4 \rangle = \langle 2,14,12,12 \rangle$ ;  $P_1$  can be executed.

Update 06: Available  $\langle 2,14,12,12 \rangle + \langle 1,0,0,0 \rangle = \langle 3,14,12,12 \rangle$  = Total Resource  $\langle (0+1+1+0+0)+(1), (0+0+3+6+0)+(5), (1+0+5+3+1)+(2), (2+0+4+2+4)+(0) \rangle = \langle 3,14,12,12 \rangle$

Safe Sequence:  $P_0, P_2, P_3, P_4, P_1$

Proces s	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
$P_0$	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0

P <sub>1</sub>	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P <sub>2</sub>	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P <sub>3</sub>	0	6	3	2	0	6	5	2	2	14	11	8	0	0	2	0
P <sub>4</sub>	0	0	1	4	0	6	5	6	2	14	12	12	0	6	4	2

Example 02:

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P <sub>0</sub>	2	0	0	1	4	2	1	2	3	3	2	1
P <sub>1</sub>	3	1	2	1	5	2	5	2				
P <sub>2</sub>	2	1	0	3	2	3	1	6				
P <sub>3</sub>	1	3	1	2	1	4	2	4				
P <sub>4</sub>	1	4	3	2	3	6	6	5				

- a) What's the Need Matrix?
- b) Is system in safe state? If yes, find safe sequence.
- c) If request from  $P_1$  arrives for  $(1,1,0,0)$  can the request be immediately granted?
- d) If request from  $P_2$  arrives for  $(0,0,2,0)$  can the request be immediately granted?

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
$P_0$	2	0	0	1	4	2	1	2	3	3	2	1	2	2	1	1
$P_1$	3	1	2	1	5	2	5	2					2	1	3	1
$P_2$	2	1	0	3	2	3	1	6					0	2	1	3
$P_3$	1	3	1	2	1	4	2	4					0	1	1	2
$P_4$	1	4	3	2	3	6	6	5					2	2	3	3

Update 01: Available  $\langle 3,3,2,1 \rangle$ ;  $P_0$  can be executed.

Update 02: Available  $\langle 3,3,2,1 \rangle + \langle 2,0,0,1 \rangle = \langle 5,3,2,2 \rangle$ ;  $P_3$  can be executed.

Update 03: Available  
 $\langle 5,3,2,2 \rangle + \langle 1,3,1,2 \rangle = \langle 6,6,3,4 \rangle$ ;  $P_4$  can be executed.

Update 04: Available  
 $\langle 6,6,3,4 \rangle + \langle 1,4,3,2 \rangle = \langle 7,10,6,6 \rangle$ ;  $P_1$  can get executed.

Update 05: Available  
 $\langle 7,10,6,6 \rangle + \langle 3,1,2,1 \rangle = \langle 10,11,8,7 \rangle$ ;  $P_2$  can get executed.

Update 06: Available  
 $\langle 10,11,8,7 \rangle + \langle 2,1,0,3 \rangle = \langle 12,12,8,10 \rangle$  = Total Resources =  $\langle (2+3+2+1+1)+(3), (0+1+1+3+4)+(3), (0+2+0+1+3)+(2), (1+1+3+2+2)+(1) \rangle = \langle 12,12,8,10 \rangle$

Safe Sequence:  $P_0, P_3, P_4, P_1, P_2$

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P <sub>0</sub>	2	0	0	1	4	2	1	2	3	3	2	1	2	2	1	1
P <sub>1</sub>	3	1	2	1	5	2	5	2	5	3	2	2	2	1	3	1
P <sub>2</sub>	2	1	0	3	2	3	1	6	6	6	3	4	0	2	1	3
P <sub>3</sub>	1	3	1	2	1	4	2	4	7	10	6	6	0	1	1	2
P <sub>4</sub>	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3

c) If request from P<sub>1</sub> arrives for (1,1,0,0), system will first pretend that it accepts the request and apply banker's algorithm to check whether a deadlock occur in future. If deadlock arises accepting the request, it won't accept the request.

Updated table after pretending that the request is accepted:

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P <sub>0</sub>	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
P <sub>1</sub>	4	2	2	1	5	2	5	2	4	2	2	2	1	0	3	1
P <sub>2</sub>	2	1	0	3	2	3	1	6	5	5	3	4	0	2	1	3
P <sub>3</sub>	1	3	1	2	1	4	2	4	6	9	6	6	0	1	1	2
P <sub>4</sub>	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3

Safe Sequence: P<sub>0</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>1</sub>, P<sub>2</sub>

The request will be granted.

d) Won't be granted. Because then the available vector would be  $\langle 3,3,0,1 \rangle$  the need vector for process P<sub>4</sub> would be  $\langle 2,2,1,3 \rangle$  and the allocation vector for process P<sub>4</sub> would be  $\langle 1,4,5,2 \rangle$  and with this much available resources, system would not be able to produce a safe sequence.

### **Problem-01:**

A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST?

1. P0
2. P1
3. P2
4. None of the above since the system is in a deadlock

	Alloc			Request		
	X	Y	Z	X	Y	Z
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

**Solution:**

According to question-

- Total = [ X Y Z ] = [ 5 5 5 ]
- Total \_Alloc = [ X Y Z ] = [ 5 4 3 ]

Now,

Available

$$\begin{aligned} &= \text{Total} - \text{Total\_Alloc} \\ &= [ 5 5 5 ] - [ 5 4 3 ] \\ &= [ 0 1 2 ] \end{aligned}$$

### **Step-01:**

- With the instances available currently, only the requirement of the process P1 can be satisfied.
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then,

Available

$$\begin{aligned} &= [ 0 1 2 ] + [ 2 0 1 ] \\ &= [ 2 1 3 ] \end{aligned}$$

## **Step-02:**

- With the instances available currently, only the requirement of the process P0 can be satisfied.
- So, process P0 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [2 \ 1 \ 3] + [1 \ 2 \ 1]$$

$$= [3 \ 3 \ 4]$$

## **Step-03:**

- With the instances available currently, the requirement of the process P2 can be satisfied.
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [3 \ 3 \ 4] + [2 \ 2 \ 1]$$

$$= [5 \ 5 \ 5]$$

Thus,

- There exists a safe sequence P1, P0, P2 in which all the processes can be executed.
- So, the system is in a safe state.
- Process P2 will be executed at last.

Thus, Option (C) is correct.

### **Problem-03:**

A system has 4 processes and 5 allocatable resource. The current allocation and maximum needs are as follows-

	Allocated					Maximum				
	A	1	0	2	1	1	1	1	2	1
B	2	0	1	1	0	2	2	2	1	0
C	1	1	0	1	1	2	1	3	1	1
D	1	1	1	1	0	1	1	2	2	0

If Available = [ 0 0 X 1 1 ], what is the smallest value of x for which this is a safe state?

Solution:

Let us calculate the additional instances of each resource type needed by each process.

We know,

$$\text{Need} = \text{Maximum} - \text{Allocation}$$

So, we have-

	Need				
A	0	1	0	0	2
B	0	2	1	0	0
C	1	0	3	0	0
D	0	0	1	1	0

### Case-01: For X = 0

If  $X = 0$ , then-

Available

= [ 0 0 0 1 1 ]

- With the instances available currently, the requirement of any process can not be satisfied.
- So, for  $X = 0$ , system remains in a deadlock which is an unsafe state.

## **Case-02: For X = 1**

If X = 1, then-

Available

$$= [0\ 0\ 1\ 1\ 1]$$

### **Step-01:**

- With the instances available currently, only the requirement of the process D can be satisfied.
- So, process D is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [0\ 0\ 1\ 1\ 1] + [1\ 1\ 1\ 1\ 0]$$

$$= [1\ 1\ 2\ 2\ 1]$$

- With the instances available currently, the requirement of any process can not be satisfied.
- So, for X = 1, system remains in a deadlock which is an unsafe state.

## Case-02: For X = 2

If X = 2, then-

Available

$$= [0\ 0\ 2\ 1\ 1]$$

### Step-01:

- With the instances available currently, only the requirement of the process D can be satisfied.
- So, process D is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [0\ 0\ 2\ 1\ 1] + [1\ 1\ 1\ 1\ 0]$$

$$= [1\ 1\ 3\ 2\ 1]$$

### Step-02:

- With the instances available currently, only the requirement of the process C can be satisfied.
- So, process C is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

$$= [1\ 1\ 3\ 2\ 1] + [1\ 1\ 0\ 1\ 1]$$

$$= [2\ 2\ 3\ 3\ 2]$$

### **Step-03:**

- With the instances available currently, the requirement of both the processes A and B can be satisfied
- So, processes A and B are allocated the requested resources one by one.
- They complete their execution and then free up the instances of resources held by it.

Then-

Available

$$\begin{aligned} &= [2 \ 2 \ 3 \ 3 \ 2] + [1 \ 0 \ 2 \ 1 \ 1] + [2 \ 0 \ 1 \ 1 \ 0] \\ &= [5 \ 2 \ 6 \ 5 \ 3] \end{aligned}$$

Thus,

- There exists a safe sequence in which all the processes can be executed.
- So, the system is in a safe state.
- Thus, minimum value of X that ensures system is in safe state = 2.

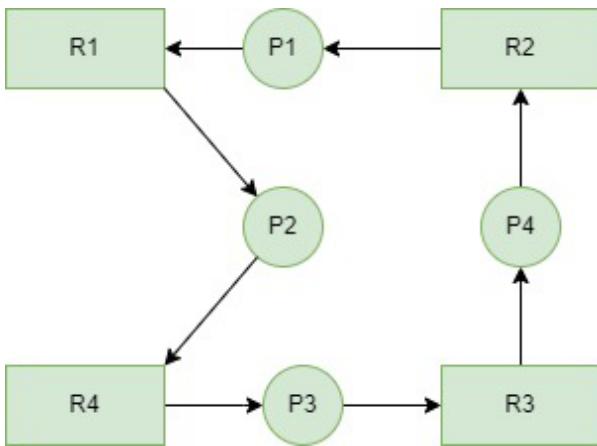
## DEADLOCK DETECTION

If we have single instances of each resource, then we use Wait-For Graph to detect deadlock.

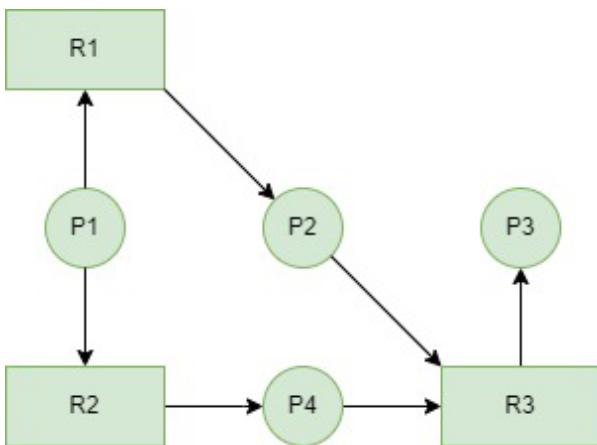
If we have several instances of a resource type, then we use Banker's / Safety Algorithm.

### 1. Wait-For Graph Detection Algorithm:

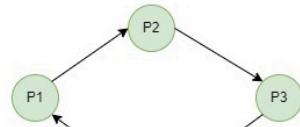
- Step 1: Take the first process ( $P_i$ ) from the resource allocation graph and check the path in which it is acquiring resource ( $R_i$ ), and start a wait-for-graph with that particular process.
- Step 2: Make a path for the Wait-for-Graph in which there will be no Resource included from the current process ( $P_i$ ) to next process ( $P_j$ ), from that next process ( $P_j$ ) find a resource ( $R_j$ ) that will be acquired by next Process ( $P_k$ ) which is released from Process ( $P_j$ ).
- Step 3: Repeat Step 2 for all the processes.
- Step 4: After completion of all processes, if we find a closed-loop cycle then the system is in a deadlock state, and deadlock is detected.



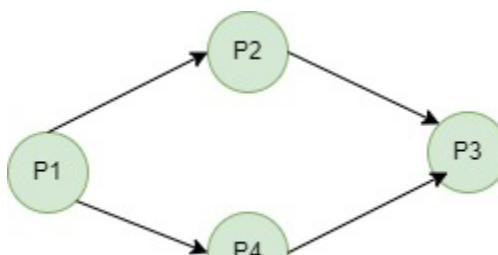
Resource Allocation Graph



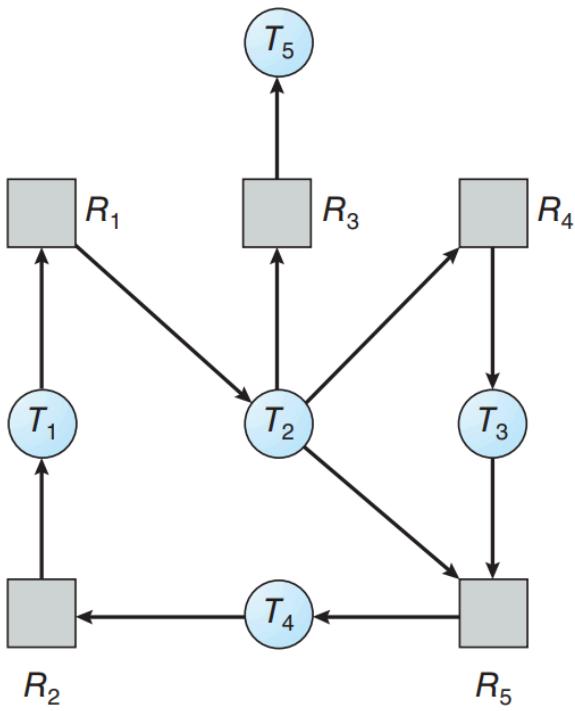
Resource Allocation Graph



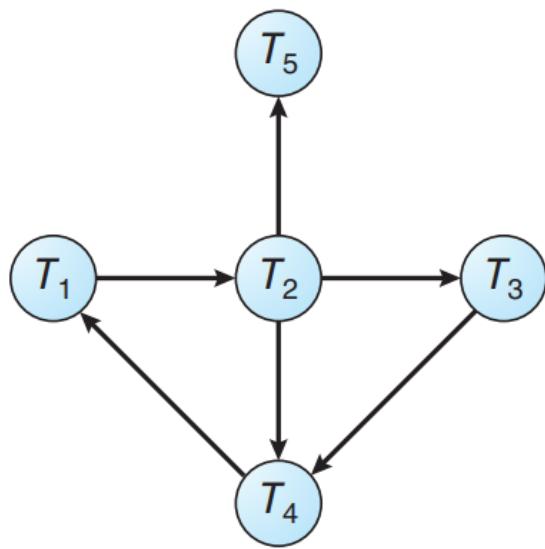
Equivalent Wait-For Graph  
(Cycle Exists, Deadlock detected)



Equivalent Wait-For Graph(Cycle does not exist, Safe state)



Resource Allocation Graph



Equivalent Wait-For Graph  
(Cycle Exists, Deadlock detected)

2. Banker's Algorithm: same as before

## DEADLOCK RECOVERY

Two options for breaking a deadlock -

1. Process & Thread Termination

- Abort all deadlocked processes: This method clearly will break the deadlock cycle but at great expense. The deadlocked processes may have been computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state. Similarly, if the process was in the midst of updating shared data while holding a mutex lock, the system must restore the status of the lock as being available, although no guarantees can be made regarding the integrity of the standard data.

We should abort those processes whose termination incur the minimum cost. Many factors may affect which process is chosen, including:

- What the priority of the process is.
- How long the process has computed and how much longer the process will compute before completing its designated task
- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
- How many more resources the process needs in order to complete
- How many process will need to be terminated

## 2. Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

- Selecting a victim: As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the

number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

- Rollback: If we preempt a resource from a process, clearly it cannot continue with its normal execution; it is missing some needed resource. We must rollback the process to some safe state and restart it from that state. Since it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
- Starvation: How can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address.

We must ensure that a process can be picked as a victim only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

## DEADLOCK PREVENTION

By ensuring that at least one of the 4 necessary conditions cannot hold, we can prevent the occurrence of a deadlock.

### 1. Attacking the Mutual Exclusion Condition

- If no resource were ever assigned exclusively to a single process, we would never have any deadlocks. Shareable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read only files are a good example of a shareable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- In general, however, we cannot prevent deadlocks by denying the mutual exclusion condition, because some resources for example CD-ROM drives are intrinsically non-shareable.

## 2. Attacking the Hold & Wait Condition

To ensure that the hold and wait condition never occurs in the system, we must guarantee that whenever a process requests a resource, it does not hold any other resources.

There are 2 ways to make this possible.

- Each process is required to request and be allocated all its resources before it begins execution.
- Each process is to request resources only when it has none, that is, before a process can request any additional resources, it must temporarily release all the resources it is currently allocated. After that, it will try to get everything it needs all at once.

Disadvantage of the 1<sup>st</sup> way:

- Many processes do not know how many resources they will need until they have started running.
- Resources will not be used optimally with this approach.

Disadvantage of the 2<sup>nd</sup> way:

- Starvation is possible. A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

### 3. Attacking the Non-Preemption Condition

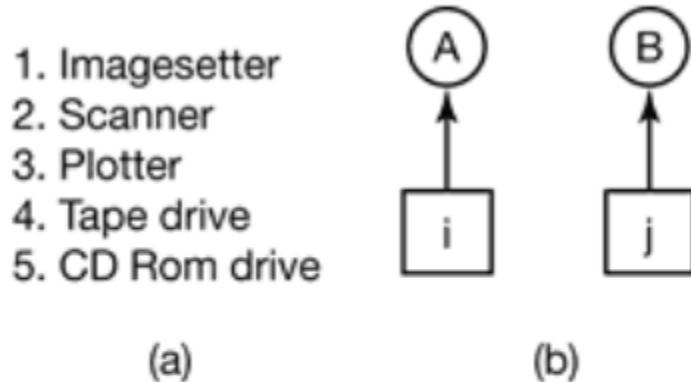
This technique is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drivers. Just think about this situation: a process has been assigned a printer and is in the middle of printing its output, but right at that moment the printer is forcibly taken away because a needed plotter is not available!!!!

### 4. Attacking the Circular Wait Condition

One way to ensure that circular wait condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

For example, we can provide a global numbering of all the resources and then there might be two protocols to be followed:

- Each process can request resources only in an increasing order of enumeration.



**Figure 3.8: (a)** Numerically ordered resources.  
**(b)** A resource graph.

A process may request first a scanner (no 2) and then a tape drive (no 4) but it may not request first a plotter (no 3) and then a scanner (no 2)

- No process can request a resource lower than what it is already holding.

For example, if a process is holding resources 2 and 4, it cannot request for resource 3 until it releases resource 4. However, it might release all of its resources and then request for any resource in the list.

Disadvantage of attacking the circular wait condition:

- It may be impossible to find an ordering that satisfies everyone.
- When the resources include process table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses maybe so large that no ordering could possibly work.

## DINING PHILOSOPHERS PROBLEM

Five philosophers live in a house, where a table is set for them. The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a

lack of manual skill, each philosopher requires two forks to eat spaghetti.

The eating arrangements are simple : a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

The problem: Devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation (in this case, the term has literal as well as algorithmic meaning!).

## FIRST SOLUTION TO DINING PHILOSOPHERS

```

/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}

```

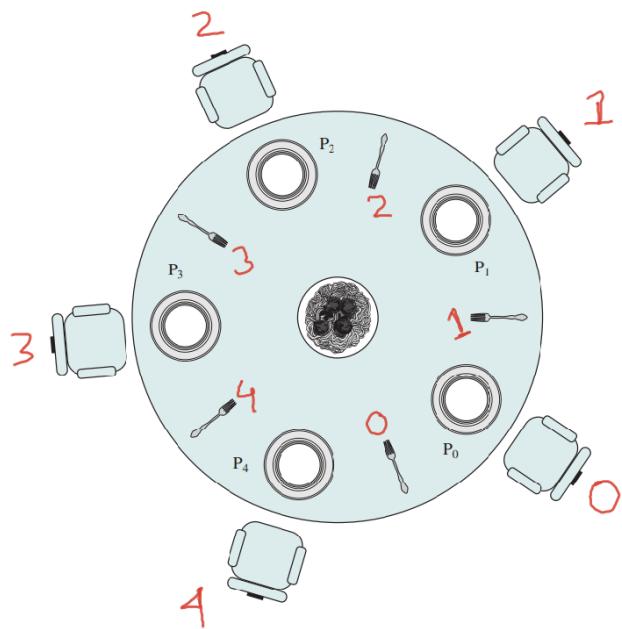
Vai ye kya hai?

At first, the fork array is initialized to 1 [ for obvious reasons xD , yeah to solve critical section problem and here the shared resources are the forks ]

So, fork [5] = {1,1,1,1,1}

Philosophers are marked from 0 to 4 [ 5 philosophers in total, also the forks are marked from 0 to 4]

Now the scenario will look like this:



Suppose philosopher 4 comes and calls `wait(fork[4])`. As `fork[4]` was set to 1, it becomes 0 after the wait operation and fork 4 is given to philosopher 4.

After that, philosopher 4 calls `wait(fork[(4+1)%5])` that is, `wait(fork[0])`. As `fork[0]` was set to 1, it becomes 0 after the wait operation and fork 0 is given to philosopher 4.

So, at the moment philosopher 4 is holding both the fork 0 and fork 4.

Suppose now comes the philosopher 0 and trying to get the fork 0.

So, philosopher 0 will call `wait(fork[0])` and decrement the value from 0 to -1 and thus get blocked.

Therefore, mutual exclusion is ensured.

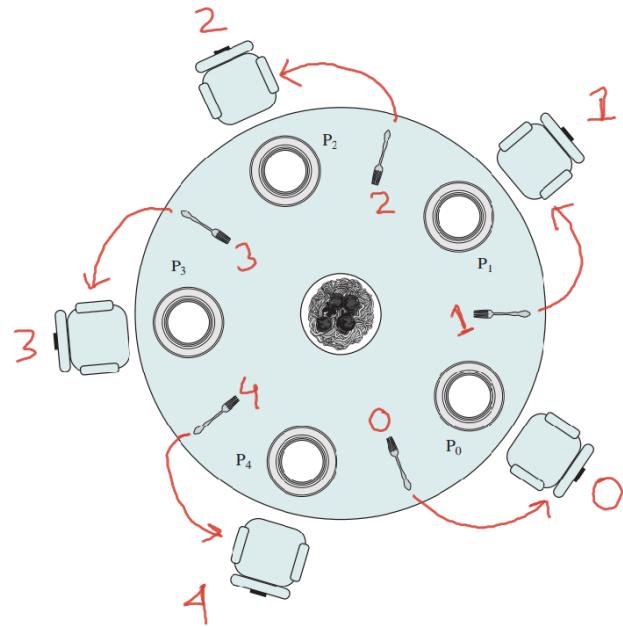
Then, the philosophers will release their forks

But the first solution to dining philosophers problem leads to DEADLOCK, how???

Let's remember the concept of context switch. Assume that, if all of the philosophers are hungry at the same time, they all sit down and calls the first wait operation and for each of them, after the first wait operation gets executed, there occurs a context switch.

If that happens, then each philosopher will hold the fork left to them and when they reach out for the other fork, it will already be picked up by another philosopher.

This scenario will look like this:

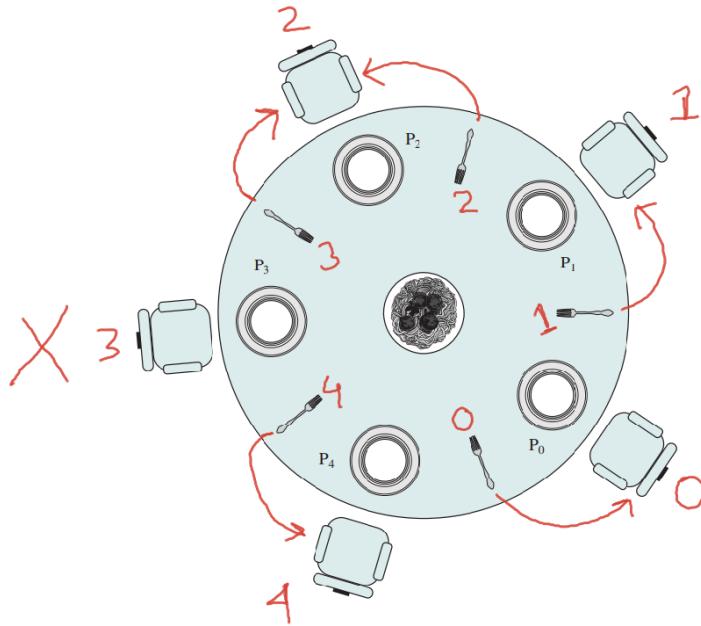


In this undignified position, all philosophers starve.

To overcome the risk of deadlock,

- We could buy five additional forks
- Or, teach the philosophers to eat spaghetti with just one fork
- Or, we could consider adding an attendant who only allows 4 philosophers at a time into the dining room. With at most 4 seated philosophers, at least 1 philosopher will have access to 2 forks. This solution is free of deadlock and starvation.

This scenario will look like this:



Code:

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1),
              philosopher (2), philosopher (3),
              philosopher (4));
}
```

## VARIOUS TERMS RELATED TO PROCESS

### Throughput

- The number of processes that are completed per time unit is called throughput.

### Turnaround Time

- The interval from the time of submission of a process to the time of completion is the turnaround time.
- Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.
- **Turnaround time = Completion Time - Arrival Time**

### Waiting Time

- Waiting time is the sum of the periods spent waiting in the ready queue.
- **Waiting time = Turnaround time - Execution Time**  
(CPU Time)

### Response Time

- The time from the submission of a request until the first response is produced.
- Response time is the time it takes to start responding, not the time it takes to output the response.

## SCHEDULING

When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the Scheduler and the algorithm it uses is called the Scheduling algorithm.

### First Come First Serve (FCFS) Algorithm

In FCFS scheduling,

- The process which arrives first in the ready queue is firstly assigned the CPU.
- In case of a tie, process with smaller process ID is executed first.

- FCFS is always non-preemptive in nature.
- FCFS does not lead to starvation.
- FCFS does not consider the priority or burst time of the processes.
- FCFS suffers from CONVOY EFFECT.

## SOLVED PROBLEMS ON FCFS

- 1) Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Soltn:

Time 0 to 2 -> P3

Time 2 to 3 -> CPU Idle

Time 3 to 7 -> P1

Time 7 to 10 -> P5

Time 10 to 13 -> P2

Time 13 to 14-> P4

Process Id	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time (TA-BT)	Response Time (CPU ALT - AT)
P1	3	4	7	4	0	0
P2	5	3	13	8	5	5
P3	0	2	2	2	0	0
P4	5	1	14	9	8	8
P5	4	3	10	6	3	3

$$\text{Average waiting time} = (0+5+0+8+3)/5 = 16/5$$

$$\text{Average turn around time} = (4+8+2+9+6)/5 = 29/5$$

2) Consider the set of 6 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	1
P4	3	4
P5	4	5
P6	5	2

If the CPU scheduling policy is FCFS and there is 1 unit of overhead in scheduling the processes, find the efficiency of the algorithm.

Time 0 to 1 -> Overhead

Time 1 to 4 -> P1

Time 4 to 5 -> Overhead

Time 5 to 7 -> P2

Time 7 to 8 -> Overhead

Time 8 to 9 -> P3

Time 9 to 10 -> Overhead

Time 10 to 14 -> P4

Time 14 to 15 -> Overhead

Time 15 to 20 -> P5

Time 20 to 21 -> Overhead

Time 21 to 23 -> P6

Useless time = 6 units

Total time = 23 units

Useful time = 17 units

Efficiency =  $17/23 * 100\%$

- 3) Three processes P1, P2, P3 with process time 20,30,10 respectively. Each process uses the first 30% of its process time in CPU, then 50% in I/O and last 20% again in CPU. Find average waiting time, turn around time and response time if system follows FCFS scheduling algorithm.

Soltn :

[https://drive.google.com/file/d/1vK0Ld2tV1P8LUUEALNle\\_Fz9NoFakkwF/view?usp=sharing](https://drive.google.com/file/d/1vK0Ld2tV1P8LUUEALNle_Fz9NoFakkwF/view?usp=sharing)

## CONVOY EFFECT

If processes with higher burst time arrived before the processes with smaller burst time, then the smaller processes have to wait for a long time for longer processes to release the CPU. This phenomenon is called Convoy Effect.

Let's understand this with an example:

Case - I:

Process Id	Arrival Time	Burst Time	Completion Time	TurnAround Time	Waiting Time (TA-BT)
P1	0	50	50	50	0
P2	1	1	51	50	49
P3	1	2	53	52	50

$$\text{Average waiting time} = (0+49+50)/3 = 33$$

Case - II:

Process Id	Arrival Time	Burst Time	Completion Time	TurnAround Time	Waiting Time (TA-BT)
P1	1	50	53	52	2
P2	0	1	1	1	0
P3	0	2	3	3	1

$$\text{Average waiting time} = (2+0+1)/3 = 1$$

In case I, process with bigger burst time (P1) arrived before the processes with smaller burst time (P2,P3), that's why, the average waiting time is 33 times bigger than that of in case II.

## NON-PREEMPTIVE SHORTEST JOB FIRST SCHEDULING

Out of all available processes, CPU is assigned to the process having smallest burst time.

In case of a tie, it is broken by FCFS algorithm.

## SOLVED PROBLEMS ON NON PREEMPTIVE SJF

- 1) Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn around time.

Soltn:

Time 0 to 6 -> P4

Time 6 to 7 -> P1

Time 7 to 9 -> P3

Time 9 to 12 -> P5

Time 12 to 16 -> P2

Process Id	Arrival Time	Burst Time	Completion Time	TurnAround Time	Waiting Time (TA-BT)
P1	3	1	7	4	3
P2	1	4	16	15	11
P3	4	2	9	5	3
P4	0	6	6	6	0
P5	2	3	12	10	7

$$\text{Average waiting time} = (3+11+3+0+7)/5 = 24/5$$

$$\text{Average turn around time} = (4+15+5+6+10)/5 = 8$$

- 2) Three processes P1, P2, P3 with process time 20,30,10 respectively. Each process uses the first 30% of its process time in CPU, then 50% in I/O and last 20% again in CPU. Find average waiting time, turn

around time and response time if system follows SJF scheduling algorithm.

Soltn:

[https://drive.google.com/file/d/1vK0Ld2tV1P8LUUEALNle\\_Fz9NoFakkwF/view?usp=sharing](https://drive.google.com/file/d/1vK0Ld2tV1P8LUUEALNle_Fz9NoFakkwF/view?usp=sharing)

PREEMPTIVE SHORTEST JOB FIRST SCHEDULING

Or

SHORTEST REMAINING TIME FIRST (SRTF)

If the newly arrived process has smaller burst time than the remaining time of the currently running process, then the preemption would occur.

SRTF provides with the minimum average waiting time and average turn around time.

SRTF provides better response time than FCFS.

SRTF gives maximum throughput.

SRTF provides a standard for other algorithms in case of average waiting time.

SRTF cannot be implemented practically.

SRTF suffers from starvation with processes having larger burst time.

### SOLVED PROBLEMS ON PREEMPTIVE SJF/SRTF

- 1) Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival Time	Burst Time
P1	2	1
P2	1	5
P3	4	1
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF preemptive (SRTF), calculate the average waiting time and average turn around time.

Soltn :

TIME 0 TO 1 : P4 (RT OF P4 = 5)

TIME 1 TO 2 : P4 ( P4,P2 ON Q, RT OF P4 = 4)

TIME 2 TO 3 : P1 (P4,P2,P5 ON Q, P1 EXITS)

TIME 3 TO 4: P5 (P4,P2,P3 ON Q, RT OF P5 = 2)

TIME 4 TO 5: P3 (P4,P2,P5 ON Q, P3 EXITS)

TIME 5 TO 7: P5 (P4,P2 ON Q, P5 EXITS)

TIME 7 TO 11: P4 ( P2 ON Q, P4 EXITS)

TIME 11 TO 16: P2

Process Id	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time (TA-BT)	Response Time
P1	2	1	3	1	0	0
P2	1	5	16	15	10	10
P3	4	1	5	1	0	0

P4	0	6	11	11	5	0
P5	2	3	7	5	2	1

$$\text{Average waiting time} = (0+10+0+5+2)/5 = 17/5$$

$$\text{Average turn around time} = (1+15+1+11+5)/5 = 33/5$$

## PREDICTION OF BURST TIME

The main drawback of SJF/SRTF is that the actual burst time of the processes cannot be known in advance.

There are several techniques which try to predict the burst time for the processes so that the SJF/SRTF algorithms can be implemented.

Based On Process Size:

- Burst time of the already executed process of similar size is taken as the burst time for the process to be executed.

Based On Process Type:

- OS (3-5 units), Interactive (5-8 units), Background (15-20 units) etc

Based On Simple Averaging:

- Burst time for the process to be executed is taken as the average of all the processes that are executed till now.
- Given ‘n’ processes  $P_1, P_2, \dots, P_n$  and burst time of each process  $P_i$  as  $t_i$  then predicted burst time for process  $P_{n+1}$  is given as

$$T_{n+1} = \frac{1}{n} \sum_{i=1}^n t_i$$

Based On Exponential Averaging:

- Given ‘n’ processes  $P_1, P_2, \dots, P_n$  and burst time of each process  $P_i$  as  $t_i$  then predicted burst time for process  $P_{n+1}$  is given as

$$T_{n+1} = \alpha t_n + (1-\alpha) * T_n$$

Where  $\alpha$  is called the smoothening factor ( $0 \leq \alpha \leq 1$ )

$t_n$  = actual burst time of process  $P_n$

$T_n$  = predicted burst time of process  $P_n$

## SOLVED EXAMPLE ON BURST TIME PREDICTION

Calculate the predicted burst time using exponential averaging for the fifth process if the predicted burst time for the first process is 10 units and actual burst time of the first four processes is 4, 8, 6 and 7 units respectively. Given  $\alpha = 0.5$

$$T_5 = 0.5 \times 7 + 0.5 \times T_4 \Rightarrow T_5 = 6.875$$

$$T_4 = 0.5 \times 6 + 0.5 \times T_3 \Rightarrow T_4 = 6.75$$

$$T_3 = 0.5 \times 8 + 0.5 \times T_2 \Rightarrow T_3 = 7.5$$

$$T_2 = 0.5 \times 4 + 0.5 \times T_1 \Rightarrow T_2 = 7$$

$$T_1 = 10$$

## ROUND ROBIN (RR) SCHEDULING ALGORITHM

CPU is assigned to the process on the basis of FCFS for a fixed amount of time. This fixed amount of time is called as Time Quantum.

After this time expires, the running process is preempted and sent to the ready queue.

Then the processor is assigned to the next arrived process.

It is always preemptive in nature.

It gives the best performance in terms of Average Response Time and thus best for time sharing systems, client server architectures and interactive systems.

It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.

With decreasing value of time quantum,

- Number of context switches increases
- Response time decreases
- Chances of starvation decreases

With increasing value of time quantum,

- Number of context switch decreases
- Response time increases
- Chances of starvation increases

## SOLVED EXAMPLES ON RR SCHEDULING

- 1) Consider the set of 6 processes whose arrival time and burst time are given below-

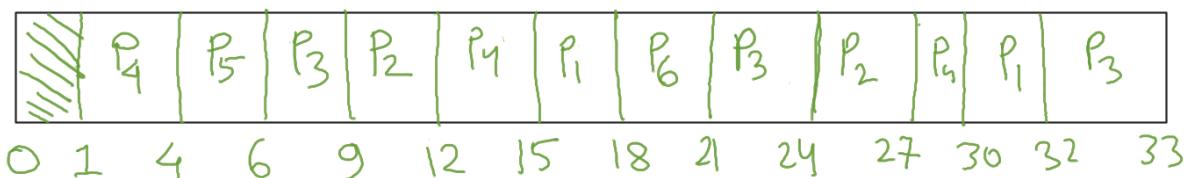
Process Id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

If the CPU scheduling policy is Round Robin with time quantum = 3, calculate the average waiting time and average turn around time.

Soltn:

Ready Queue : P<sub>6</sub>, P<sub>3</sub>, P<sub>2</sub>, P<sub>4</sub>, P<sub>5</sub>, P<sub>6</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>

$P_4 \rightarrow \text{Remaining}(6)$ $P_3 \rightarrow \text{Rem}(4)$ $P_2 \rightarrow \text{Rem}(3)$	$P_4 \rightarrow \text{Rem}(3)$ $P_1 \rightarrow \text{Rem}(2)$ $P_3 \rightarrow \text{Rem}(1)$
---	---



Process Id	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time (TA-BT)
P1	5	5	32	27	22
P2	4	6	27	23	17
P3	3	7	33	30	23
P4	1	9	30	29	20
P5	2	2	6	4	2
P6	6	3	21	15	12

$$\text{Average waiting time} = (22+17+23+20+2+12)/6 = 16$$

$$\text{Average turn around time} = (27+23+30+29+4+15)/6 = 64/3$$

2) Consider the set of 5 processes whose arrival time and burst time are given below-

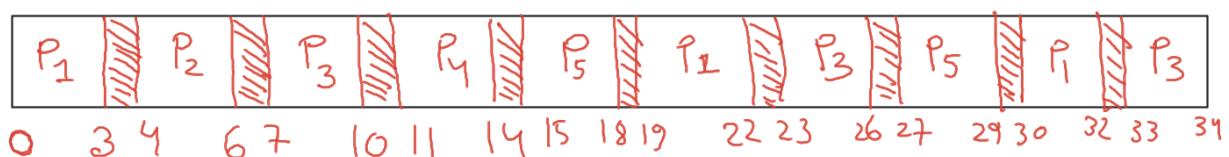
Process Id	Arrival Time	Burst Time

P1	0	8
P2	0	2
P3	0	7
P4	0	3
P5	0	5

If context switch time is 1 unit and time quantum = 3,  
 Find the average waiting time, turn around time, response  
 time, number of context switches and CPU utilization if  
 system uses Round Robin Scheduling algorithm.

Ready Queue: P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>5</sub>, P<sub>1</sub>, P<sub>3</sub>, P<sub>5</sub>, P<sub>1</sub>, P<sub>3</sub>

$$\left. \begin{array}{l} P_1 \rightarrow \text{Rem}(5) \\ P_3 \rightarrow \text{Rem}(4) \\ P_5 \rightarrow \text{Rem}(2) \end{array} \right| \left. \begin{array}{l} P_1 \rightarrow \text{Rem}(2) \\ P_3 \rightarrow \text{Rem}(1) \end{array} \right.$$



Process Id	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time (TA-BT)	Response Time
P1	0	8	32	32	24	0
P2	0	2	6	6	4	4
P3	0	7	34	34	27	7
P4	0	3	14	14	11	11
P5	0	5	29	29	24	15

Number of context switches = 9

Average waiting time =  $(24+4+27+11+24)/5 = 18$

Average turnaround time =  $(32+6+34+14+29)/5 = 23$

Average response time =  $(0+4+7+11+15)/5 = 37/5$

CPU Utilization =  $25/34 * 100 = 73.5\%$

## NONPREEMPTIVE PRIORITY SCHEDULING

Out of all available processes, CPU is assigned to the process having the highest priority.

In case of a tie, it is broken by FCFS scheduling.

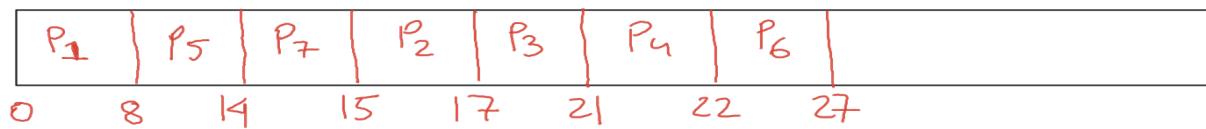
## SOLVED EXAMPLES ON NONPREEMPTIVE PRIORITY SCHEDULING

- 1) Consider the set of 7 processes whose arrival time and burst time are given below-

Process Id	Arrival Time	Burst Time	Priority
P1	0	8	3
P2	1	2	4
P3	3	4	4
P4	4	1	5
P5	5	6	2
P6	6	5	6
P7	10	1	1

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time. (Lesser number represents higher priority)

~~P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>5</sub>, P<sub>6</sub>, P<sub>7</sub>~~



Process Id	Arrival Time	Burst Time	Priority	Completion Time	Turn Around Time	Waiting time
P1	0	8	3	8	8	0
P2	1	2	4	17	16	14
P3	3	4	4	21	18	14
P4	4	1	5	22	18	17
P5	5	6	2	14	9	3
P6	6	5	6	27	21	16

P7	10	1	1	15	5	4
----	----	---	---	----	---	---

$$\text{Average waiting time} = (0+14+14+17+3+16+4)/7 = \\ 68/7$$

$$\text{Average turn around time} = \\ (8+16+18+18+9+21+5)/7 = 95/7$$

## SOLVED EXAMPLES ON PREEMPTIVE PRIORITY SCHEDULING

- 1) Consider the set of 7 processes whose arrival time and burst time are given below-

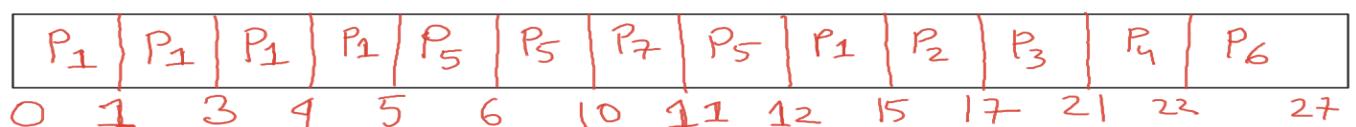
Process Id	Arrival Time	Burst Time	Priority
P1	0	8	3
P2	1	2	4
P3	3	4	4

P4	4	1	5
P5	5	6	2
P6	6	5	6
P7	10	1	1

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Lesser number represents higher priority)

Soltn:

$$\begin{aligned} P_1 &\rightarrow \text{Rem}(3) \\ P_5 &\rightarrow \text{Rem}(1) \end{aligned}$$



Process Id	Arrival Time	Burst Time	Priority	Completion Time	Turn Around Time	Waiting time
P1	0	8	3	15	15	7
P2	1	2	4	17	16	14
P3	3	4	4	21	18	14
P4	4	1	5	22	18	17
P5	5	6	2	12	7	1
P6	6	5	6	27	21	16
P7	10	1	1	11	1	0

$$\begin{aligned}\text{Average Waiting Time} &= (7+14+14+17+1+16+0)/5 \\ &= 69/5\end{aligned}$$

$$\begin{aligned}\text{Average Turn Around Time} &= \\ &(15+16+18+18+7+21+1)/5 = 96/5\end{aligned}$$

MORE PRACTICES

❖ Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:

- a. FCFS
- b. RR
- c. Multilevel feedback queues

FCFS discriminates against short jobs since any short jobs arriving after long jobs will have a longer waiting time.

RR treats all jobs equally giving them equal bursts of the CPU time so short jobs will be able to leave the system faster since they will finish first.

Multilevel feedback queues work similar to the RR algorithm- they discriminate favourably toward short jobs.

❖ Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

I/O bound programs have the property of performing only a small amount of computation before performing IO. Such programs typically don't use up their entire CPU quantum.

CPU bound programs use their entire quantum without performing any blocking I/O operations.

Consequently, one could make better use of the computer's resources by giving higher priority to I/O bound programs and allow them to execute ahead of the CPU bound programs.

- ❖ Discuss how the following pairs of scheduling criteria conflict in certain settings.
  - a. CPU utilization and response time
  - b. Average turnaround time and maximum waiting time
  - c. I/O device utilization and CPU utilization

CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could however result in increasing the response time for processes.

Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could however

starve long running tasks and thereby increase their waiting time.

CPU utilization is maximized by running long running CPU bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

---

## MEMORY MANAGEMENT

In a uniprogramming system, main memory is divided into 2 parts: one part for the OS and other part for the program currently being executed.

But in a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes.

And this task of subdivision is carried out dynamically by the operating system and is known as Memory Management.

## WHY MEMORY MANAGEMENT IS VITAL?

Effective memory management is vital in a multiprogramming system because if only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O, and the processor will be idle. Thus, memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

Requirements that memory management is intended to satisfy are:

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Organization

## RELOCATION

Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his/her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. Once a program is swapped out to disk, it would be quite limiting to specify that when it is next swapped back in, it must be placed in the same main memory region as before. Instead, we may need to relocate the process to a different area of memory.

## PROTECTION

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process for reading or writing purposes without permission.

Satisfaction of the relocation requirement increases the difficulty of satisfying the protection requirement.

Because the location of a program in main memory is unpredictable, it is impossible to check absolute addresses at compile time to assure protection. Furthermore, most programming languages allow the dynamic calculation of addresses at run time. Hence, all memory references generated by a process must be checked at run time to ensure they refer only to the memory space allocated to that process.

The memory protection requirement must be satisfied by the processor (hardware) than the operating system (software). This is because the OS cannot anticipate all of the memory references that a program will make. Even if such anticipation were possible, it would be prohibitively time consuming to screen each program in advance for possible memory-reference violations. Thus it is only possible to assess the permissibility of a memory reference (data or branch) at the time of execution of the instruction making the reference. To accomplish this, the processor hardware must have that capacity.

## SHARING

Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory. If a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program, rather than have its own separate copy. Processes that are cooperating on some task may need to share access to the same data structure. The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection.

## **FIXED MEMORY PARTITIONING**

In most schemes for memory management, we can assume the OS occupies some fixed portion of main memory, and the rest of main memory is available for use by multiple processes. The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries.

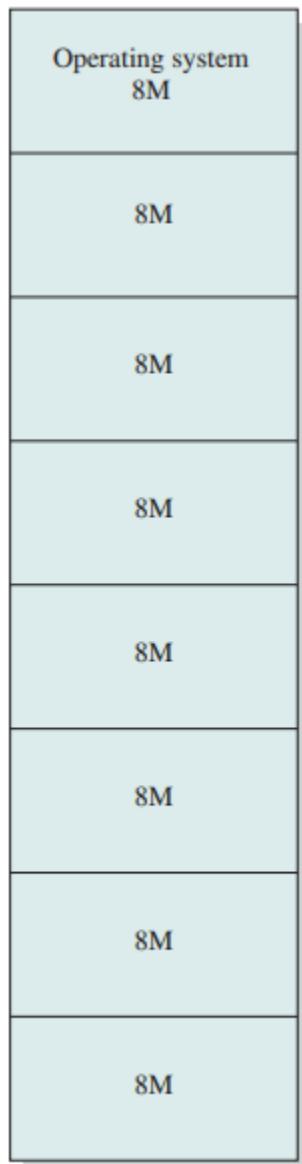
## **EQUAL SIZE PARTITIONS**

In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full, and no process is in the Ready or Running state, the OS can swap a process out of any of the partitions and load in another process.

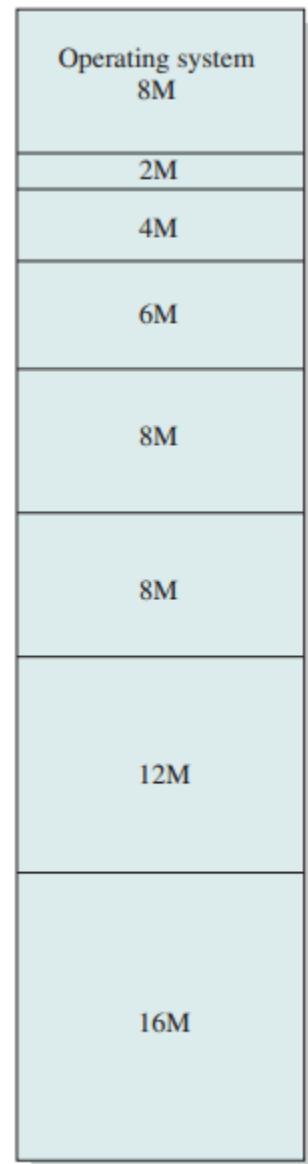
There are two difficulties with the use of equal-size fixed partitions:

- A program may be too big to fit into a partition. In this case, the programmer must design the program with the use of overlays so only a portion of the program need be in main memory at any one time. When a module is needed that is not present, the user's program must load that module into the program's partition, overlaying whatever programs or data are there.
- Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. For instance, there may be a program whose length is less than 2MB, yet it occupies an 8MB partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller

than the partition, is referred to as internal fragmentation.



(a) Equal-size partitions



(b) Unequal-size partitions

With equal size partitions, the placement of processes in memory is trivial. As long as there is any available partition, a process can be loaded into that partition.

Because all partitions are of equal size, it does not matter which partition is used. If all partitions are occupied with processes that are not Ready to run, then one of these processes must be swapped out to make room for a new process. Which one to swap out is a scheduling decision.

## UNEQUAL SIZE PARTITIONS

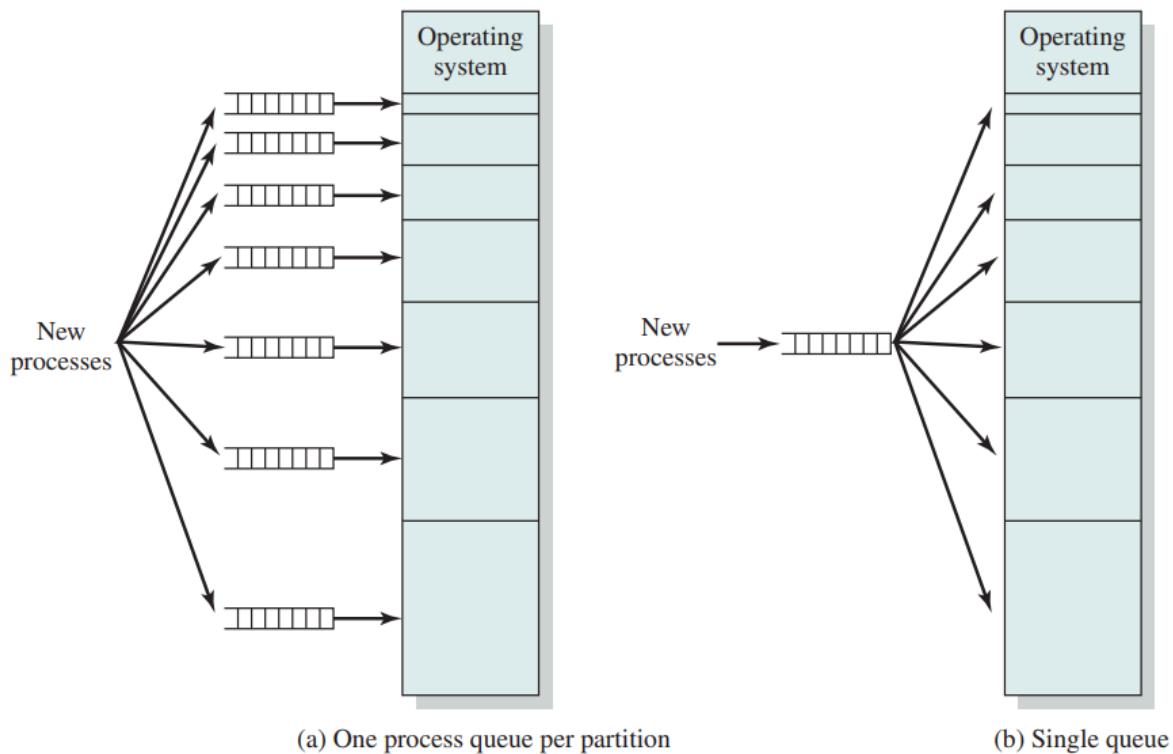
Both of the problems of equal-size partitions can be lessened, though not solved by using unequal-size partitions. In the figure above, programs as large as 16MB can be accommodated without overlays. Partitions smaller than 8MB allow smaller programs to be accommodated with less internal fragmentation.

With unequal size partitions, there are 2 possible ways to assign processes to partitions.

The simplest way is to assign each process to the smallest partition within which it will fit. In this case, a scheduling queue is needed for each partition to hold swapped-out

processes destined for that partition. The advantage of this approach is that processes are always assigned in such a way as to minimize wasted memory within a partition (internal fragmentation).

[ this approach assumes that one knows the maximum amount of memory that a process will require. This is not always the case. If it is not known how large a process may become, the only alternatives are an overlay scheme or the use of virtual memory. ]



In the figure, consider a case where there are no processes with a size between 12MB and 16MB at a certain point in time. In that case, the 16MB partition will remain unused, even though some smaller process could have been assigned to it. Thus a preferable approach would be to employ a single queue for all processes. When it is time to load a process into main memory, the smallest available partition that will hold the process is selected. If all partitions are occupied, then a swapping decision must be made.

Preference might be given to swapping out the smallest partition that will hold the incoming process. It is also possible to consider other factors, such as priority and a preference for swapping out blocked processes versus ready processes.

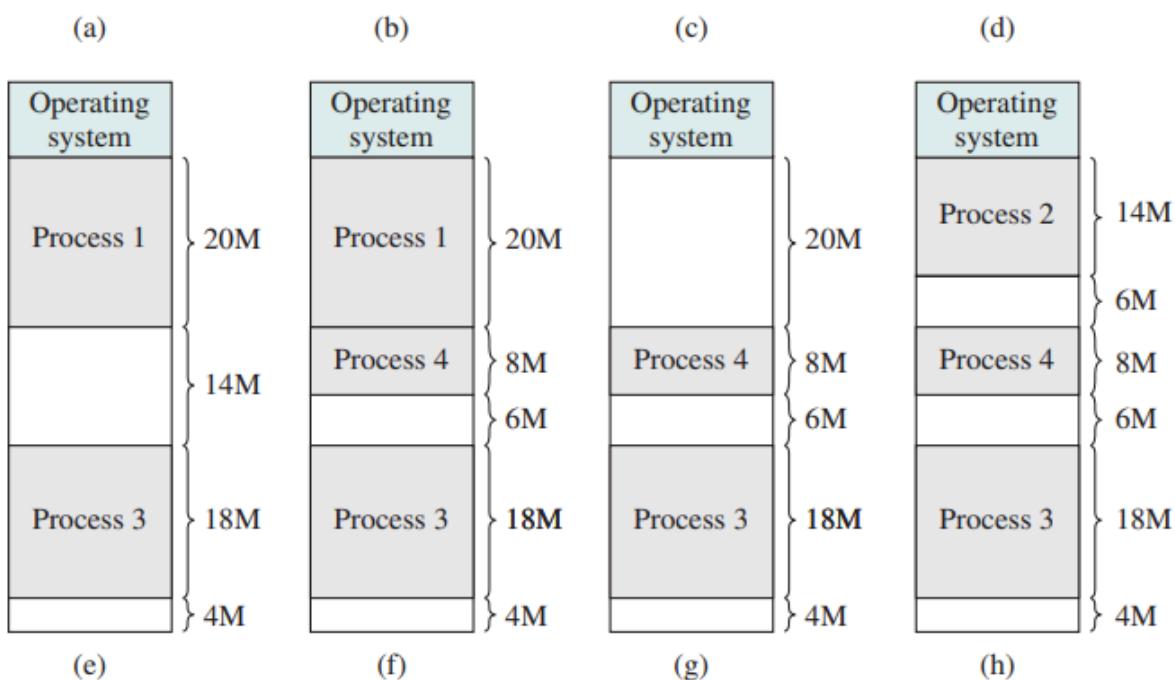
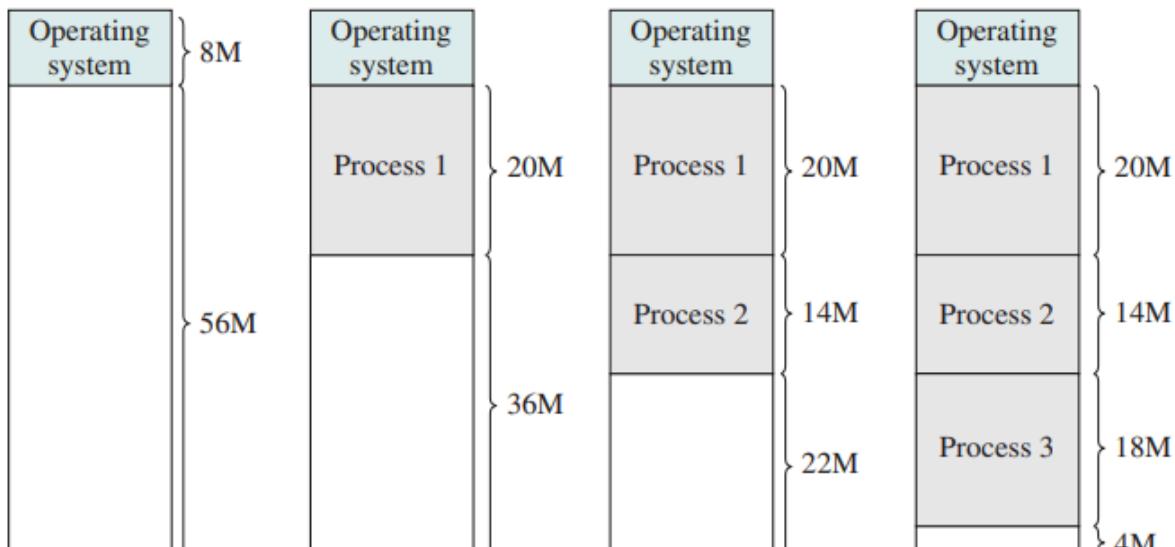
The use of unequal-size partitions provides a degree of flexibility to fixed partitioning. In addition, it can be said that fixed partitioning schemes are relatively simple and require minimal OS software and processing overhead. But there are disadvantages:

- The number of partitions specified at system generation time limits the number of active processes in the system.

- Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently. In an environment where the main storage requirement of all jobs is known beforehand, this may be reasonable, but in most cases, it is an inefficient technique.

## DYNAMIC PARTITIONING

In this approach, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.



Situation (a): Initially the main memory is empty, except for the OS

Situation (b,c,d): The first 3 processes are loaded in, starting where the operating system ends and occupying

just enough space for each process. This leaves a “HOLE” at the end of memory that is too small for a fourth process.

Situation (e,f): At some point, none of the processes in memory is ready. The OS swaps out process 2, which leaves sufficient room to load a new process, process 4. Because process 4 is smaller than process 2, another small hole is created.

Situation (g,h): Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the READY-SUSPENDED state, is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out and swaps process 2 back in.

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, memory utilization declines. This phenomenon is referred to as EXTERNAL FRAGMENTATION, indicating the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier.

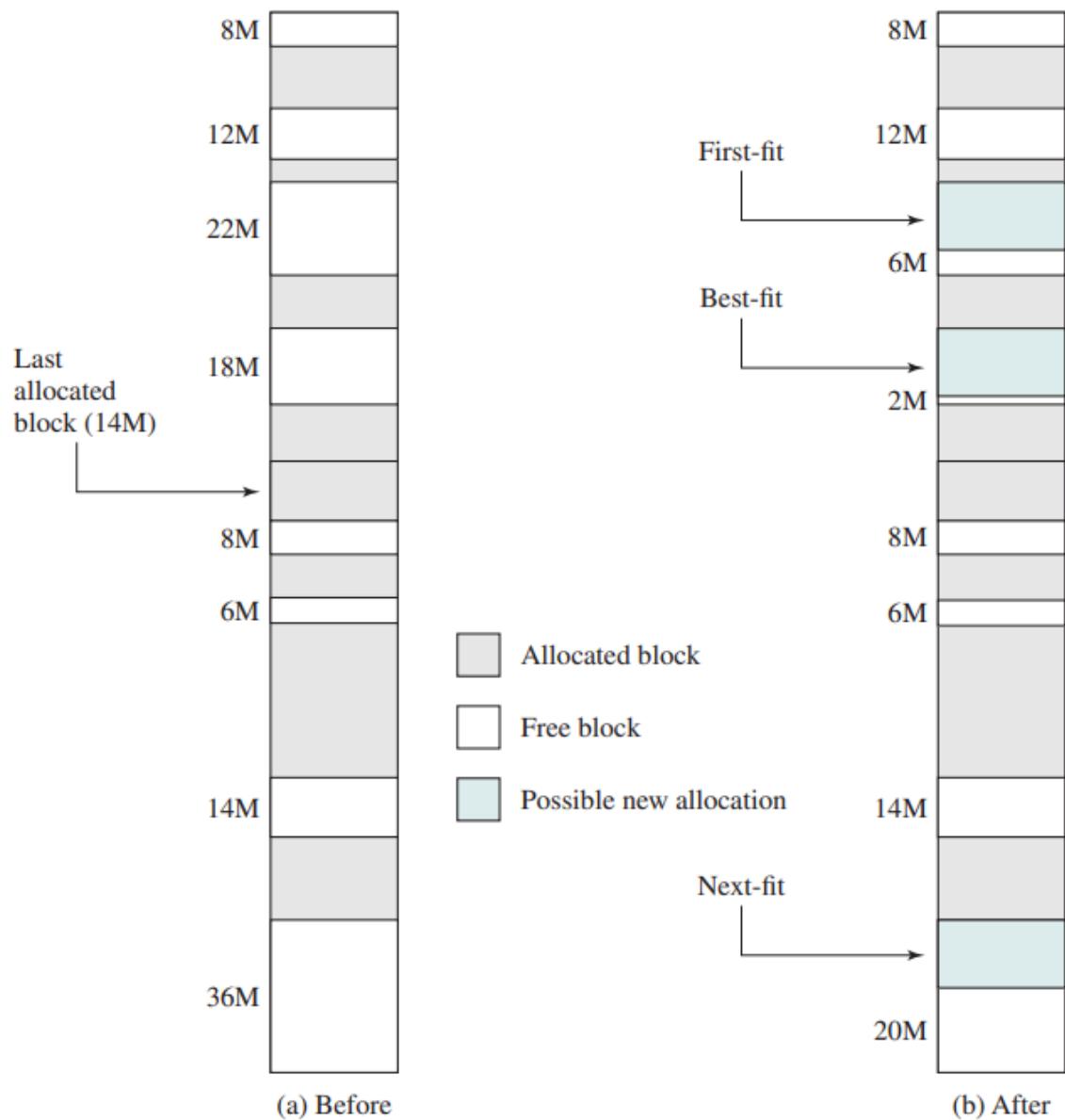
## One technique for overcoming external fragmentation is COMPACTI

- From time to time, the OS shifts the processes so they are contagious and all of the free memory is together in one block. For example, in the figure (h) compaction will result in a block of free memory of length 16MB. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time-consuming procedure and wasteful of processor time. Note that, the compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a program from one region to another in main memory, without invalidating the memory references in the program.

When it is time to load or swap a process into main memory and if there is more than one free block of memory of sufficient size, then the OS must decide which free block to allocate.

3 placement algorithms that might be considered are

- Best Fit: Chooses the block that is closest in size to the request.
- First Fit: Begins to scan memory from the beginning and chooses the first available block that is large enough.
- Next Fit: Begins to scan memory from the location of the last placement and chooses the next available block that is large enough.



## BEST IS THE WORST?

The first fit algorithm is not only the simplest but usually the best and fastest as well.

The next fit algorithm tends to produce slightly worse results than the first fit. The next fit algorithm will more frequently lead to an allocation from a free block at the end of a memory. The result is that the largest block of free memory which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus, compaction maybe required more frequently with next fit.

On the other hand, the first fit algorithm may litter the front end with small free partitions that need to be searched over on each subsequent first pass.

The best fit algorithm, despite its name, is usually the worst performer. Because this algorithm looks for the smallest block that will satisfy the requirement, it guarantees that the fragment left behind is as small as possible. Although each memory request always wastes the smallest amount of memory, the result is that main memory is quickly littered by blocks too small to satisfy memory allocation requests. Thus, memory compaction must be done more frequently than with the other algorithms.

## SOLVED EXAMPLES ON MEMORY PARTITIONING

- 1) Consider a fixed partitioning scheme with equal-size partitions of  $2^{16}$  bytes and a total main memory size of  $2^{24}$  bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

Number of partitions would be  $= 2^{24}/2^{16} = 2^8$

Number of bits required for the pointer = 8 bits

- 2) Another placement algorithm for dynamic partitioning is referred to as worst-fit. In this case, the largest free block of memory is used for bringing in a process.
  - a. Discuss the pros and cons of this method compared to first-, next-, and best-fit.
  - b. What is the average length of the search for worst-fit?
- 3) This diagram shows an example of memory configuration under dynamic partitioning, after a number of placement and swapping-out operations have been carried out. Addresses go from left to right; gray areas indicate blocks occupied by processes; white areas indicate free memory blocks. The last process

placed is 2 Mbytes and is marked with an X. Only one process was swapped out after that.



- a. What was the maximum size of the swapped-out process?
- b. What was the size of the free block just before it was partitioned by X?
- c. A new 3-Mbyte allocation request must be satisfied next. Indicate the intervals of memory where a partition will be created for the new process under the following four placement algorithms: best-fit, first-fit, next-fit, and worst-fit. For each algorithm, draw a horizontal segment under the memory strip and label it clearly

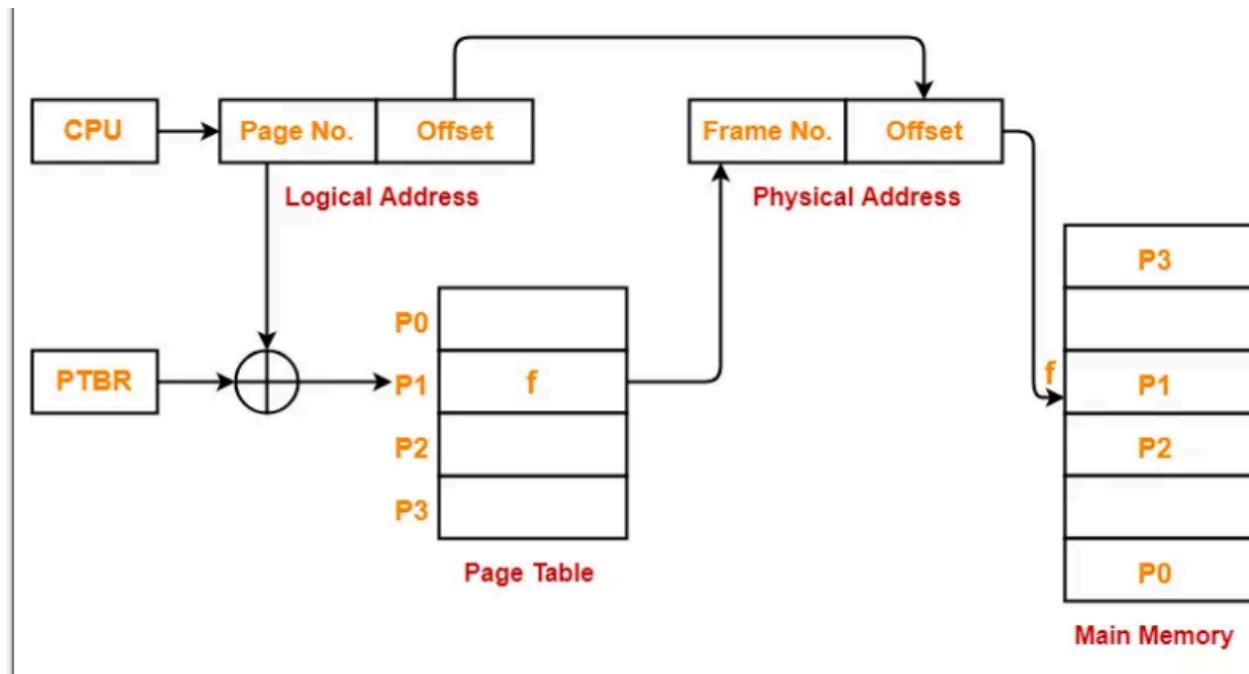
## PAGING

Frame: A fixed length block of main memory.

**Page:** A fixed length block of data that resides in secondary memory. A page of data may temporarily be copied into a frame of main memory.

- Paging is a fixed partitioning scheme.
- In paging, secondary memory and main memory are divided into equal fixed size partitions.
- The partitions of secondary memory are called Pages.
- The partitions of main memory are called Frames.
- Each process is divided into parts where each part is same as page size.
- The pages of process are stored in the frames of main memory depending upon their availability.
- CPU always generates a logical address.
- A physical address is needed to access the main memory.
- Logical address consists of two parts. Page Number which specifies the specific page of the process from which CPU wants to read the data. And page offset which specifies the specific word on the page that CPU wants to read.
- For the page number generated by CPU, Page Table provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.

- The frame number combined with the page offset forms the required Physical Address.
- Frame number specifies the specific frame where the required page is stored.
- Page offset specifies the specific word that has to be read from that page.



The advantages of Paging are:

- It allows to store parts of a single process in a non-contiguous fashion.

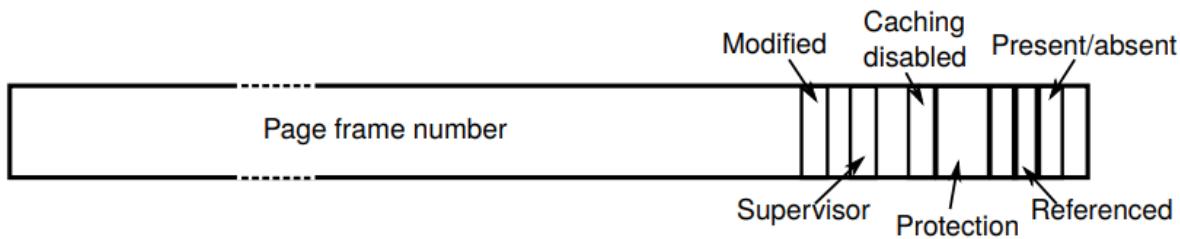
- It solves the problem of external fragmentation.

The disadvantages of Paging are:

- It suffers from internal fragmentation due to fixed sized partitioning scheme.
- There is an overhead of maintaining a page table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required.

## STRUCTURE OF A PAGE TABLE ENTRY

The exact layout of a page table entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine.



## Present/Absent Bit

- Indicates whether the entry is valid and can be used.
- If this bit is 0, the virtual page to which the entry belongs is not currently in memory.
- Accessing a page table entry with this bit set to 0 causes a page fault.

## Protection Bit

- Tells what kinds of accesses are permitted.
- In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only.
- A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing and executing the page.

## Supervisor bit

- Indicates whether the page is accessible only to privileged code or also to user programs.
- Any attempt by a user program to access a supervisor page will result in a fault.

## Modified bit

- Keeps track of page usage.

- When a page is written to, the hardware automatically sets the Modified bit. This bit is of value when the OS decides to reclaim a page frame.
- If the page in it has been modified (is “Dirty”), it must be written back to nonvolatile storage.
- If it has not been modified (is “Clean”) it can just be abandoned, since the copy on disk or SSD is still valid.
- This bit is called the “Dirty Bit” since it reflects the page’s state.

## Referenced Bit

- Is set whenever a page is referenced, either for reading or writing.
- Its value is used to help the OS choose a page to evict when a page fault occurs.
- Pages that are not being used are far better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms.

## Caching Disabled Bit

- Allows caching to be disabled for the page.
- Important for pages that map onto device registers rather than memory.
- If the OS is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is

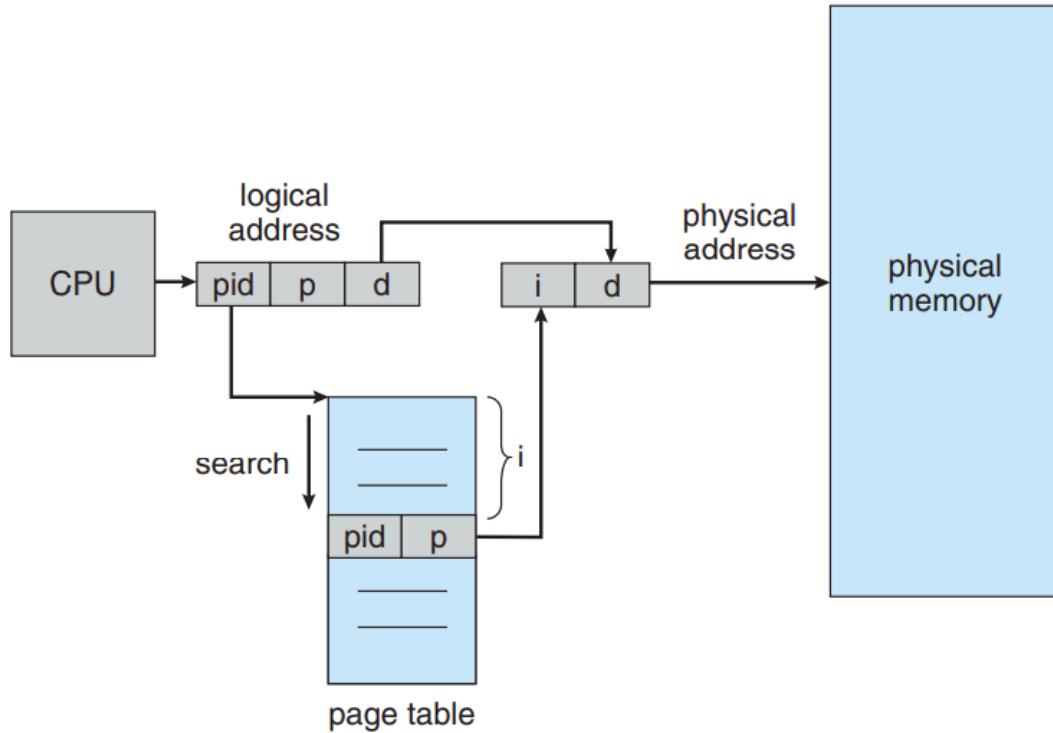
essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off.

- Machines that have a separate I/O space and do not use memory-mapped I/O do not need this bit.

## INVERTED PAGING

Inverted page table consists of a one-page table entry for every frame of the main memory. So, the number of page table entries in the inverted page table reduces to the number of frames in physical memory and a single page table is used to represent the paging information of all the processes.

Through the inverted page table, the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together. This technique is called Inverted paging as the indexing is done concerning the frame number instead of the logical page number.



In inverted page table indexing is done with frame numbers instead of the logical page number. Each entry in the page table contains the following fields.

**Frame No:** It specifies the Frame where the page no is actually present in main memory.

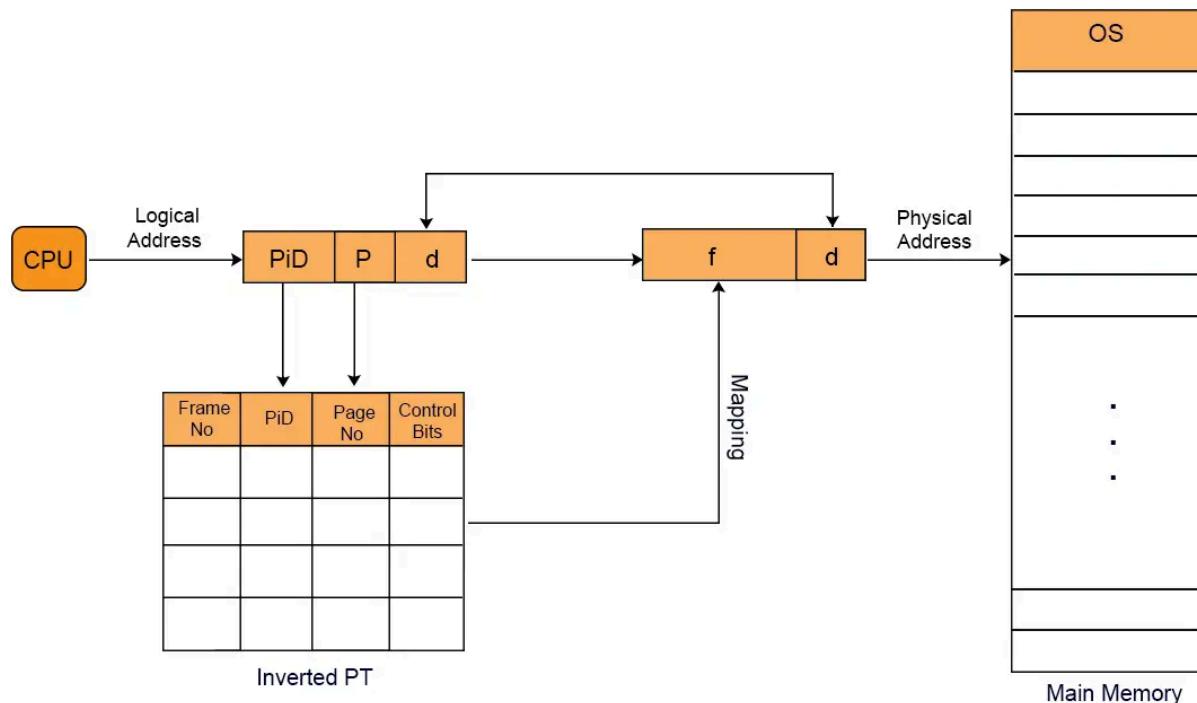
**Page number:** It specifies the page number which is required.

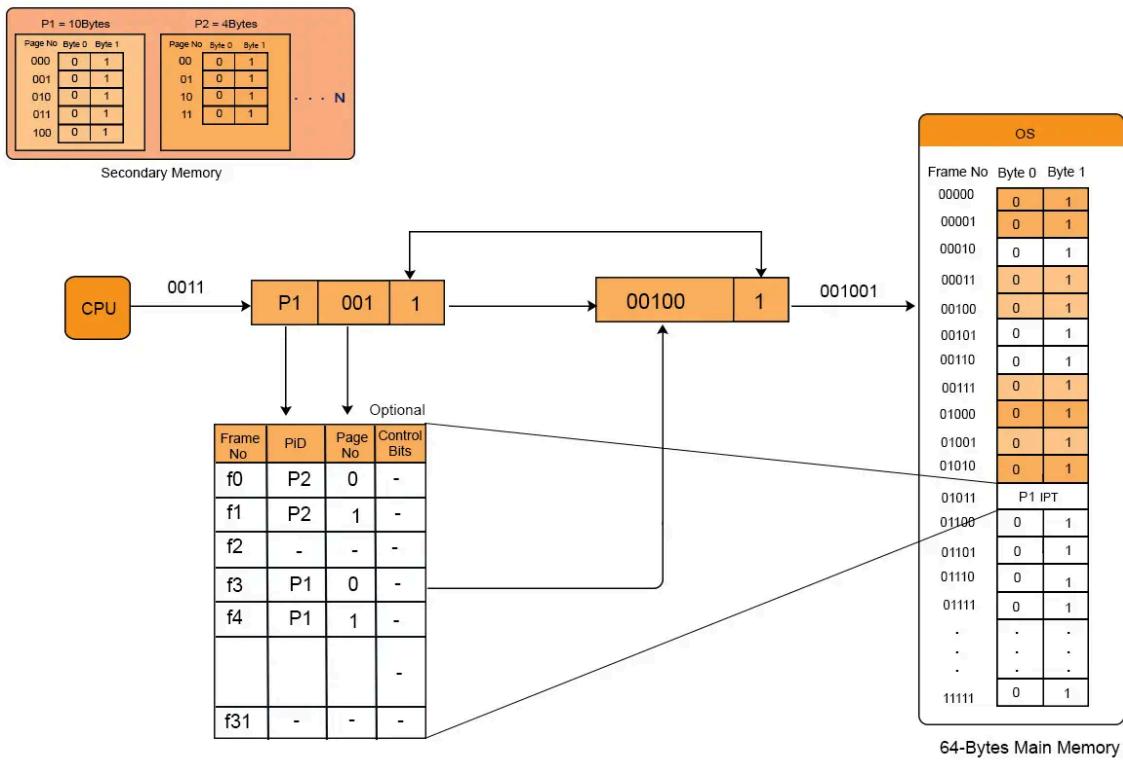
**Process id:** An inverted page table contains pages of all the processes in execution. So page number may be same of different process but Process Id of each process is unique.

Through Process ID we get the desired page of that process.

Control bits –These bits are used to store paging table entry information. These include the valid/invalid bit, protection, dirty bit, reference bits, and other information bit.

**Chained pointer:**It is possible when two or more processes share some part of main memory. In simple words, when two or more logical pages need to map to same Page Table Entry then a chaining pointer is used.





Note: Number of Entries in Inverted page table = Number of frames in Physical Address Space(PAS).

- Inverted page tables typically reduce the amount of memory required to store the page tables to a size bound of physical memory. The maximum number of entries could be the number of frames in the physical memory.
- Inverted Page tables are sorted in order of frame number but the memory look-up takes place concerning

the virtual address, so, it usually takes a longer time to find the appropriate entry. Often these page tables are implemented using hash data structures for a faster lookup.

## SOLVED EXAMPLES ON PAGING

Formulas:

Physical Address Space	$= \text{Size of Main Memory}$ $= \text{Total Number of frames} \times \text{Frame Size}$ $= \text{Total Number of Frames} \times \text{Page Size}$
Number of Frames in Main Memory $= 2^x$	Number of bits in Frame Number $= x$ bits
Page size $= 2^x$ Bytes	Number of bits in Page Offset $= x$ bits
Size of main memory $= 2^x$ Bytes	Number of bits in physical address $= x$ bits
Virtual Address Space	Size of process

Number of pages the process is divided	$= \text{Process size} / \text{Page size}$
Process size = $2^x$ Bytes	Number of bits in virtual address space = x bits
Size of page table	$= \text{Number of entries in page table} \times \text{Page table entry size}$ $= \text{Number of pages the process is divided} \times (\text{Number of bits in frame number} + \text{Number of bits used for optional field if any})$
Size of memory when the given address consists of n bits and the memory is m-byte addressable	$= 2^n \times m$ Bytes

- 1) Calculate the size of memory if its address consists of 22 bits and the memory is 2-byte addressable.

$$\text{Size of memory} = 2^{22} \times 2 \text{ Bytes} = 2^{23} \text{ Bytes} = 2^{20} \times 2^3 \text{ Bytes} = 8\text{MB}$$

2) Calculate the number of bits required in the address for memory having size of 16 GB. Assume the memory is 4-byte addressable.

Let the number of bits required in the address = n bits

$$\text{Now, } 2^n \times 4 = 2^4 \times 2^{30}$$

$$\Rightarrow 2^n = 2^{32}$$

$$\Rightarrow n = 32$$

3) Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is

Size of the page table = number of page entries x size of each entry

$$= (2^{32} / (4 \times 2^{10})) \times 4 = 2^{20} \times 4 \text{ Bytes} = 4\text{MB}$$

4) Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB, what is the approximate size of the page table?

Size of physical memory =  $64\text{MB} = 2^6 \times 2^{20} \text{ Bytes} = 2^{26} \text{ Bytes}$

Number of bits in physical address = 26

Number of frames in main memory =  $2^{26}/(2^2 \times 2^{10}) = 2^{14}$

Number of bits in frame number = 14

Number of bits in page offset = 26-14 = 12

Process size =  $2^{32} \text{ Bytes}$

Number of entries in page table =  $2^{32}/(2^2 \times 2^{10}) = 2^{20}$

Page table size = Number of entries in page table x page table entry size

= Number of entries in page table x Number of bits in frame number

=  $2^{20} \times 14 \text{ bits}$

=  $2^{20} \times 16 \text{ bits} \text{ [approximately]}$

=  $2^{20} \times 2 \text{ bytes}$

= 2MB

- 5) In a virtual memory system, size of virtual address is 32-bit, size of physical address is 30-bit, page size is 4 Kbyte and size of each page table entry is 32-bit. The main memory is byte addressable. The maximum number of bits that can be used for storing protection and other information in each page table entry = ?

Size of physical address 30 bits

Size of main memory =  $2^{30}$  bytes

Page size = 4KB =  $2^2 \times 2^{10}$  bytes =  $2^{12}$  bytes

Number of bits in page offset = 12

Number of bits in frame number = 18

Size of each page table entry = 32 = Number of bits in frame number + additional bits to store protection and other information = 18 + x => x = 32 - 18 = 14

- 6) Consider a memory-management system based on paging. The total size of the physical memory is 2 GB, laid out over pages of size 8 KB. The logical address space of each process has been limited to 256 MB.
- a. Determine the total number of bits in the physical address.

- b. Determine the number of bits specifying page replacement and the number of bits for page frame number.
  - c. Determine the number of page frames.
  - d. Determine the logical address layout.
- a) Size of physical memory =  $2\text{GB} = 2 \times 2^{30} \text{ Bytes} = 2^{31} \text{ Bytes}$
- Total number of bits in physical address = 31
- b) Page size =  $8\text{KB} = 2^3 \times 2^{10} \text{ Bytes} = 2^{13} \text{ Bytes}$
- Number of bits in page offset = 13
- Number of bits in frame number =  $31 - 13 = 18$
- c) Number of page frames =  $2^{18} = 2^8 \times 2^{10} = 256\text{k}$
- d) Logical address space for each process =  $256\text{MB} = 2^8 \times 2^{20} \text{ Bytes} = 2^{28} \text{ Bytes}$

Number of bits in logical address = 28

Number of bits in page offset = 13

Number of bits in page number =  $28 - 13 = 15$

## MULTILEVEL PAGING

Multilevel paging is a paging scheme where there exists a hierarchy of page tables.

The need for multilevel paging arises when

- The size of page table is greater than the frame size.
- As a result, the page table cannot be stored in a single frame in main memory.

In multilevel paging, the page table having size greater than the frame size is divided into several parts.

The size of each part is same as frame size.

The pages of page table are then stored in different frames of main memory.

To keep track of the frames storing the pages of the divided page table, another page table is maintained.

As a result, the hierarchy of page tables get generated.

Multilevel paging is done until the level is reached where the entire page table can be stored in a single frame.

## SOLVED EXAMPLES ON MULTILEVEL PAGING

- 1) Consider a system using paging scheme where

Logical Address space = 4GB

Physical Address space = 16TB

Page size = 4KB

Find out how many levels of page table will be required.

- 2) Consider a system using multilevel paging scheme.

The page size is 16 KB. The memory is byte addressable and virtual address is 48 bits long. The page table entry size is 4 bytes.

How many levels of page table will be required?

Give the divided physical address and virtual address.

3) Consider a system using multilevel paging scheme.

The page size is 1 MB. The memory is byte addressable and virtual address is 64 bits long. The page table entry size is 4 bytes.

Find how many levels of page table will be required?

Give the divided physical address and virtual address.

Solutions: [EK HI KAAFI HAI](#)

4) Consider a three level paging scheme. The page size is 4 KB and page table entry size is 4 bytes. The size of outermost page table is 4 KB. Give the division of virtual address space.

Q3 3 level paging

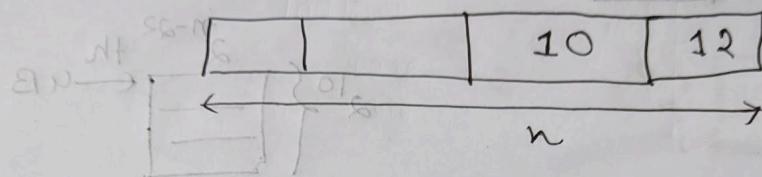
Page size 4KB

Page Table Entry size 4B

Q4 Size of outmost page table 4KB

Virtual Address

Space = ?



$$\text{Page size} = 4\text{KB} = 2^2 \times 2^{10} \text{B} = 2^{12} \text{Bytes}$$

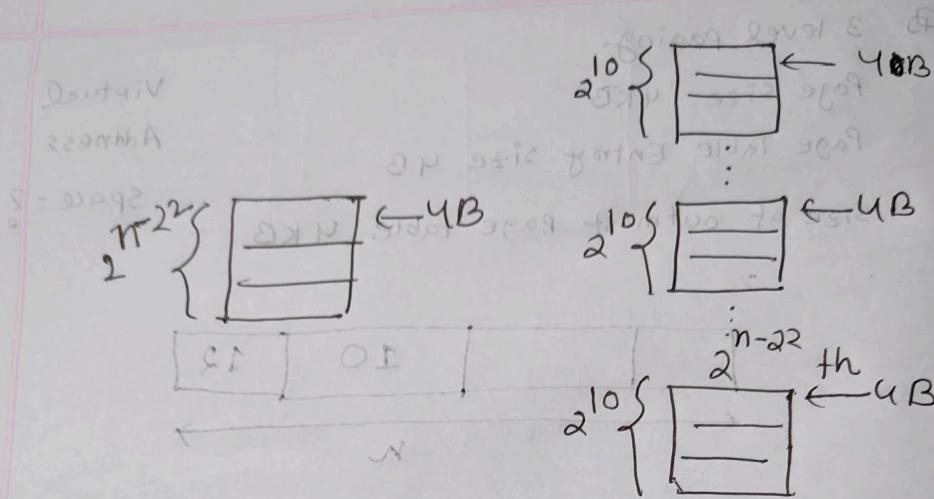
$$\text{Process size} = 2^n \text{B} = \text{Total memory for process}$$

$$\text{Number of pages} = \frac{2^n \text{B}}{2^{12} \text{B}} = 2^{n-12} \text{Bytes}$$

Page Table size =  ~~$2^{n-10} \times 4$~~   $n-12$

$$= \frac{2^{n-10}}{2^{12}} = 2^{n-22} \text{Bytes}$$

Number of page tables after division =  $\frac{2^{n-10}}{2^{12}} = 2^{n-22}$  page tables



$$\text{Size of outer-1 table} = 2^{n-22} \times 4B$$

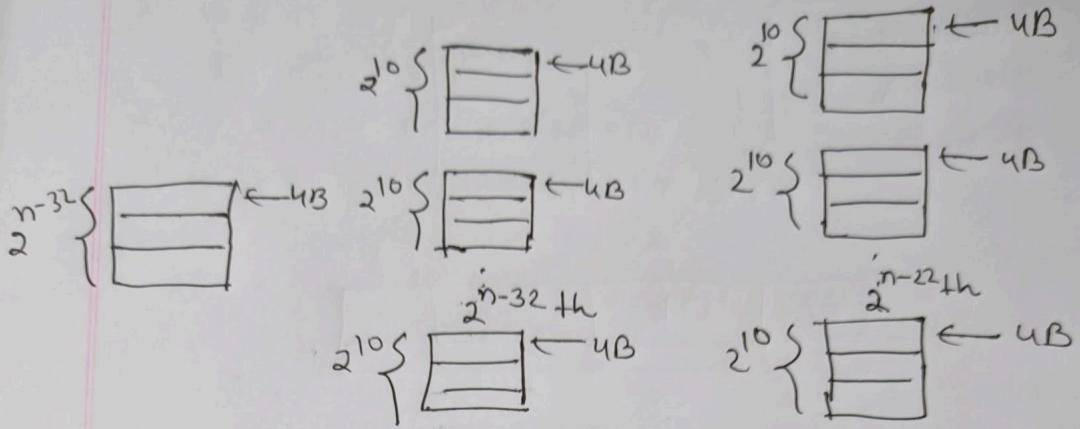
$\frac{2^{n-22} \times 4B}{2^{n-12}} = 2^{n-20} \text{ Bytes}$

$$\text{Number of pages} = \frac{2^{n-20}}{2^{12}} = 2$$

~~tables after dividing outer table-1~~

$$= \frac{10}{8192} = \frac{1}{8192}$$

~~bytes per page~~



Given that the outer table-2 size = 4 KB

$$\therefore 2^{n-32} \times 4 B = 2^{12} B \\ = 2^{12} \text{ Bytes}$$

$$\Rightarrow n-30 = 12$$

$$\therefore \boxed{n = 42}$$

10	10	10	12
----	----	----	----

## TRANSLATION LOOKASIDE BUFFER

Translation Lookaside Buffer is a solution that tries to reduce the effective access time.

Being a hardware, the access time of TLB is very less as compared to the main memory.

TLB consists of two columns - Page Number & Frame Number

When CPU generates a logical address containing Page Number and Page offset, TLB is checked to see if it contains an entry for the referenced page number. The referenced page number is compared with the TLB entries all at once.

If TLB contains an entry for the referenced page number, a TLB hit occurs. In this case, TLB entry is used to get the corresponding frame number for the referenced page number.

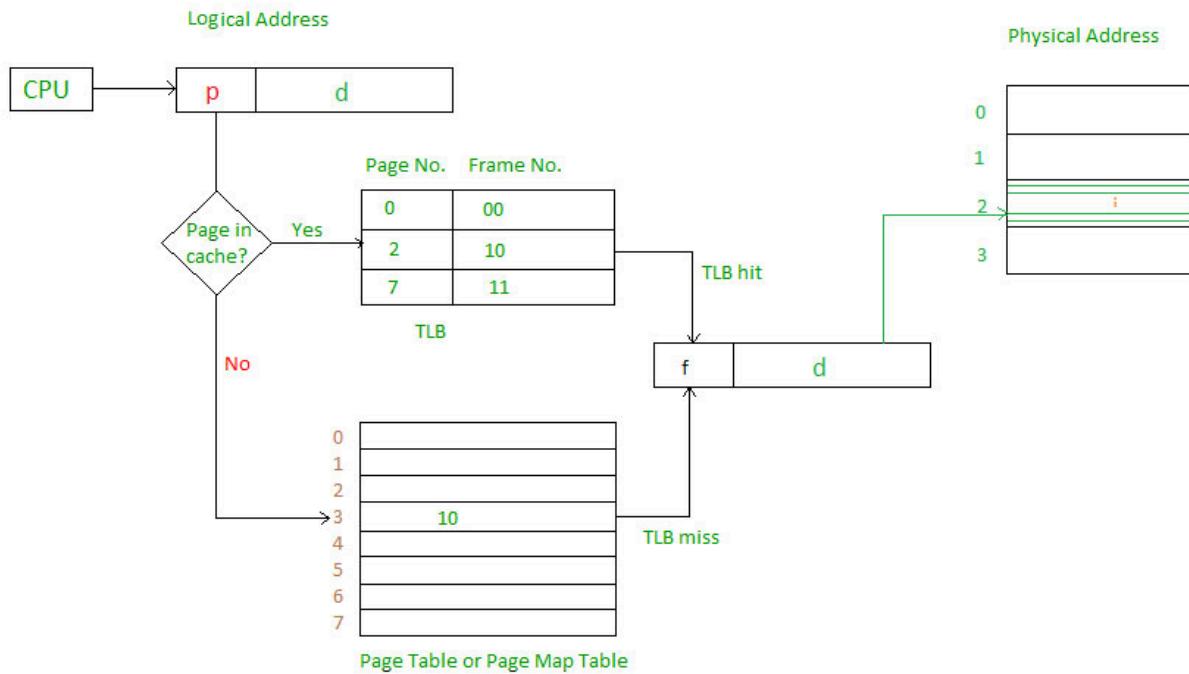
If TLB does not contain an entry for the referenced page number, a TLB miss occurs. In this case, page table is used to get the corresponding frame number for the referenced page number. Then TLB is updated with the page number and frame number for future references.

Whenever context switching occurs, the entire content of TLB is flushed and deleted. TLB is again updated with the currently running process.

Initially TLB is empty, so TLB miss is frequent. With every access from the page table, TLB is updated. After some time, TLB hits increases and TLB misses reduces.

TLB reduces the effective access time. Only one memory access is required when TLB hit occurs.

TLB can hold the data of only one process at a time. When context switches occur frequently, the performance of TLB degrades due to low hit ratio.



**Effective Access Time (Without Page Fault) = Hit ratio of TLB x {Access time of TLB + Access time of main memory} + Miss ratio of TLB x { Access time of TLB + (L+1) x Access time of main memory } }**

**Effective Access Time (With Page Faults) = Page Fault Rate x (Effective Access Time without page fault + Page fault service time) + (1-Page Fault Rate) x Effective Access Time without page fault**

## SOLVED EXAMPLES ON TLB

- 1) Consider a single level paging scheme with a TLB.

Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If effective memory access time is 130 ns, TLB hit ratio is = ?

$$130 = H \times (20 + 100) + (1 - H) \times \{(1 + 1) \times 100 + 20\}$$

$$H = 0.9 \text{ or } 90\%$$

- 2) Consider a three level paging scheme with a TLB.

Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is = ?

$$EAT = 0.8 \times (20 + 100) + 0.2 \times (20 + (3 + 1) \times 100)$$

$$EAT = 180 \text{ ns}$$

3) Let the page fault service time be 10 ms in a computer with average memory access time being 20 ns. If one page fault is generated for every  $10^6$  memory accesses, what is the effective access time for the memory?

$$\text{EAT with page fault} = \frac{1}{10^6} \times (20 + 10^7) + (1 - \frac{1}{10^6}) \times 20 = 30\text{ns}$$

4) Suppose the time to service a page fault is on the average 10 milliseconds, while a memory access takes 1 microsecond. Then, a 99.99% hit ratio results in average memory access time of-

$$\text{Page Fault Rate} = 1 - \text{Hit Ratio} = 1 - 99.99/100 = .0001$$

$$\text{EAT with page fault} = .0001 \times (1 + 10 \times 10^{-3}) + (1 - .0001) \times 1 = 2 \text{ microseconds}$$

5) Consider a system with a two-level paging scheme in which a regular memory access takes 150 nanoseconds and servicing a page fault takes 8 milliseconds. An

average instruction takes 100 nanoseconds of CPU time and two memory accesses. The TLB hit ratio is 90% and the page fault rate is one in every 10,000 instructions. What is the effective average instruction execution time?

$$\text{EAT without page fault} = 0.9 (0+150) + 0.1 (0+(2+1)\times 150) = 180 \text{ ns}$$

$$\text{EAT with page fault} = 1/10^4 (180 + 8 \times 10^6) + (1 - 1/10^4) \times (180) = 980 \text{ ns}$$

$$\text{Effective average instruction execution time} = 100 + 2 * 980 = 2060 \text{ ns}$$

6) Assume a simple paging system with  $2^{32}$  bytes of physical memory,  $2^{48}$  bytes of logical address space and pages that are  $2^{24}$  bytes in size. Further assume that each page table entry contains 4 bits indicating protection and validity of the entry.

(i) How many bits are used for the frame number and how many for the frame offset?

(ii) What is the total size of the page table in number of bits?

(iii) Assume that the working set of a typical process is fixed throughout the process lifetime and consists of 20 pages. How many entries would you suggest for the Translation Lookaside Buffer(TLB) for this system? What would its total size be in number of bits? Explain your answer.

(iv) Further assume that TLB search time is 20ns, TLB hit ratio is 80% and memory access time is 100ns. How many page table levels would you need to achieve an effective access time of 160ns, and why?

Soltn:

i) Page size =  $2^{24}$  Bytes

Number of bits in page offset = Number of bits in frame offset = 24

Physical memory size =  $2^{32}$  Bytes

Number of bits in physical address = 32

Number of bits in frame number =  $32 - 24 = 8$

ii) Total size of the page table in number of bits = (Logical Memory Size / Page size ) \* Page Table Entry Size  
=  $(2^{48}/2^{24}) * (\text{number of bits in frame number} + \text{protection and validity bits}) = 2^{24} * (8+4) = 12*2^{24}$  bits

iii) TLB total size = Number of pages in fixed working set \* Page Table Entry Size =  $20*(8+4)$  bits = 240 bits

iv) Effective Access Time (Without Page Fault) = Hit ratio of TLB x {Access time of TLB + Access time of main memory} + Miss ratio of TLB x { Access time of TLB +  $(L+1) * \text{Access time of main memory}$  }

$$160 = 0.8*(20+100) + 0.2*(20+(L+1)*100)$$

$$L = \text{number of levels} = 2$$



<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>M</b>	<b>M</b>	<b>H</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

Number of page faults = 6

Hit Ratio =  $4/10 = \frac{2}{5}$

Miss Ratio =  $\frac{3}{5}$

Least Recently Used (LRU) Algorithm:

<b>4</b>	<b>7</b>	<b>6</b>	<b>1</b>	<b>7</b>	<b>6</b>	<b>1</b>	<b>2</b>	<b>7</b>	<b>2</b>
4	4	4	1	1	1	1	1	1	1
	7	7	7	7	7	7	2	2	2
		6	6	6	6	6	6	7	7
<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>M</b>	<b>M</b>	<b>H</b>

Number of page faults = 6

Hit Ratio =  $4/10 = \frac{2}{5}$

Miss Ratio =  $\frac{3}{5}$

Optimal Page Algorithm:

<b>4</b>	<b>7</b>	<b>6</b>	<b>1</b>	<b>7</b>	<b>6</b>	<b>1</b>	<b>2</b>	<b>7</b>	<b>2</b>
4	4	4	1	1	1	1	1	1	1
	7	7	7	7	7	7	7	7	7
		6	6	6	6	6	2	2	2
<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>M</b>	<b>H</b>	<b>H</b>

Number of page faults = 5

Hit ratio =  $5/10 = \frac{1}{2}$

Miss ratio =  $\frac{1}{2}$

For a reference string consisting of repeated sequence of page numbers

- Optimal Page Replacement Algorithm replaces the most recently used page to minimize the page faults.

This is because most recently page will be required after the longest time.

- Thus, Optimal page replacement algorithm acts as Most Recently Used (MRU) page replacement algorithm.

Consider the following reference string consisting of a repeated sequence of page numbers-

1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5,  
6

Optimal Page Replacement == Most Recently Used Page Replacement Example

1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	3	3	3	3	
	2	2	2	2	2	2	2	2	2	2	2	2	2	3	4	4	4	4	4	4	4	4	4	4
		3	3	3	3	3	3	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
			4	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
M	M	M	M	M	M	H	H	H	M	M	H	H	M	M	H	H	H	M	M	H	H	H	H	H

Number of page faults = 12

Hit Ratio =  $12/24 = \frac{1}{2}$

Miss Ratio =  $12/24 = \frac{1}{2}$

For a reference string where first half string is a mirror image of other half

- Optimal page replacement algorithm replaces the least recently used page or firstly arrived page to minimize page faults.
- This is because such a page will be required after the longest time.
- Thus, Optimal page replacement algorithm acts as LRU and FIFO page replacement algorithm.

Consider the following reference string where first half string is a mirror image of other half string-

1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1

<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>
1	1	1	1	5	5	5	5	9	9	9	9	9	5	5	5	5	1
	2	2	2	2	6	6	6	6	6	6	6	6	6	6	6	2	2
		3	3	3	3	7	7	7	7	7	7	7	7	7	3	3	3
			4	4	4	4	8	8	8	8	8	8	8	4	4	4	4
<b>M</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>H</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>							

Number of page fault = 14

Hit Ratio = 4/18

Miss Ratio = 14/18

## BELADY'S ANOMALY

The number of page faults should either decrease or remain same on increasing the number of frames in main memory. But sometimes the unusual behaviour is noticed, that is, on increasing the number of frames in main memory, the number of page faults also increases.

Belady's Anomaly is the phenomenon of increasing the number of page faults on increasing the number of frames in main memory.

FIFO replacement algorithm suffers from Belady's Anomaly [ not necessarily always ]

5	4	3	2	1	4	3	5	4	3	2	1	5
5	5	5	2	2	2	3	3	3	3	3	1	1
	4	4	4	1	1	1	5	5	5	5	5	5
		3	3	3	4	4	4	4	4	2	2	2
M	M	M	M	M	M	M	M	H	H	M	M	H

Number of page faults = 10

<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>4</b>	<b>3</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>5</b>
5	5	5	5	1	1	1	1	1	1	2	2	2
	4	4	4	4	4	4	5	5	5	5	1	1
		3	3	3	3	3	3	4	4	4	4	5
			2	2	2	2	2	2	3	3	3	3
<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>H</b>	<b>H</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>	<b>M</b>

Number of page faults = 11

## SEGMENTATION

Like paging, segmentation is another non-contiguous memory allocation technique. In segmentation, process is not divided into fixed size pages. Rather, the process is divided into modules for better visualization.

Segmentation is a variable size partitioning scheme. In segmentation secondary memory and main memory is divided into partitions of unequal size. The size of partitions depend on the length of modules.

The partitions of secondary memory are called as Segments.

## SEGMENT TABLE

Segment table has two columns.

First column stores the size or length or limit of the segment.

Second column stores the base address or starting address of the segment in the main memory.

Segment table is stored as a separate segment in the main memory. STBR( segment table base register) stores the base address of the segment table.

## LOGICAL TO PHYSICAL ADDRESS: SEGMENTATION

CPU generates a logical address consisting of Segment Number and Segment offset.

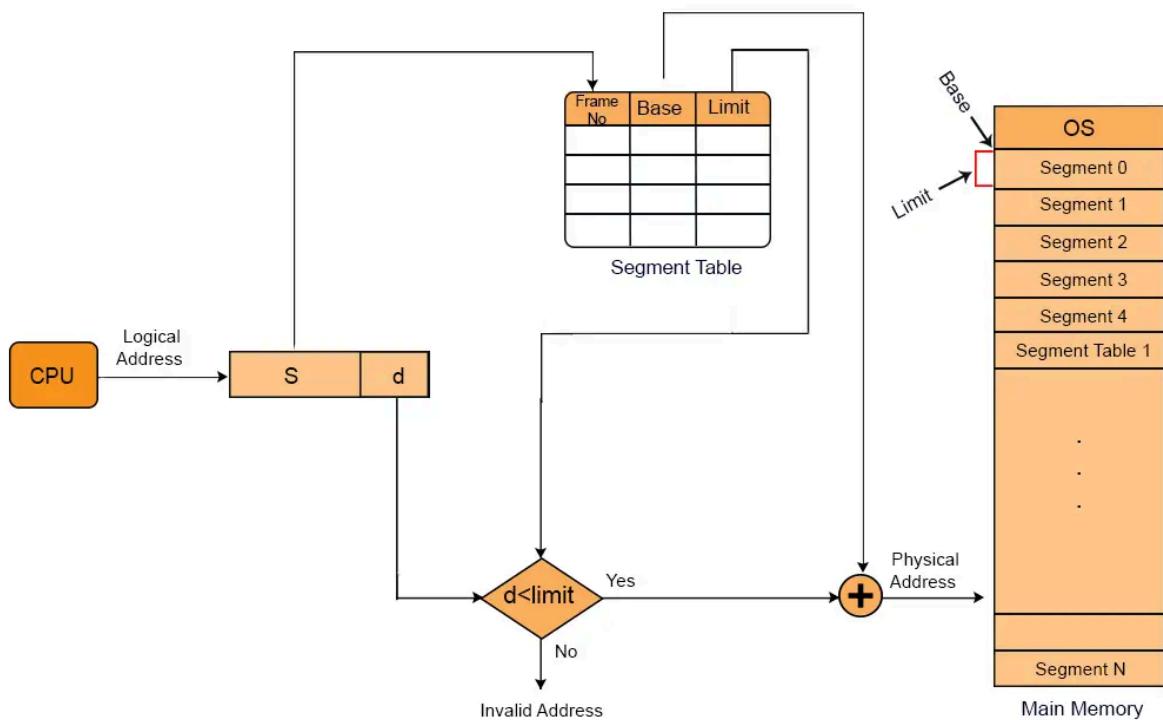
Segment number specifies the specific segment of the process from which CPU wants to read the data.

Segment offset specifies the specific word in the segment that CPU wants to read.

For the generated segment number, corresponding entry is located in the Segment table. Then segment offset is compared with the limit/size/length of the segment.

If segment offset  $\geq$  limit, then a trap is generated.

If segment offset  $<$  limit, then it is considered to be a valid request. And segment offset is added with the base address of the segment. The result obtained after addition is the address of the memory location storing the required word.



## ADVANTAGES OF SEGMENTATION

- It allows to divide the program into modules which provides better visualization.
- Segment table consumes less space as compared to Page Table.
- It solves the problem of Internal Fragmentation.

## DISADVANTAGES OF SEGMENTATION

- There is an overhead of maintaining a segment table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required.
- It suffers from external fragmentation as the free space gets broken down into smaller pieces with the processes being loaded and removed from the main memory.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

## SOLVED EXAMPLE ON SEGMENTATION

Consider the following segment table-

Segment No	Base	Length
0	1219	700

1	2300	14
2	90	100
3	1327	580
4	1952	96

Which of the following logical address will produce trap addressing error?

1. 0, 430
2. 1, 11
3. 2, 100
4. 3, 425
5. 4, 95

Calculate the physical address if no trap is produced.

1. Segment number = 0, Segment Offset = 430

Segment offset must always lie in the range [0,700-1] that is, [0,699] and 430 does lie within the range.

So, no trap will be produced.

$$\text{Physical Address} = 1219 + 430 = 1649$$

2. Segment number = 1, Segment Offset = 11

Segment offset must always lie in the range [0,14-1] that is, [0,13] and 11 does lie within the range.

So, no trap will be produced.

$$\text{Physical Address} = 2300 + 11 = 2311$$

3. Segment number = 2, Segment Offset = 100

Segment offset must always lie in the range [0,100-1] that is, [0,99] and 100 does not lie within the range.

So, trap will be produced.

4. Segment number = 3, Segment Offset = 425

Segment offset must always lie in the range [0,580-1] that is, [0,579] and 425 does not lie within the range.

So, no trap will be produced.

$$\text{Physical Address} = 1327 + 425 = 1752$$

5. Segment number = 4, Segment Offset = 95

Segment offset must always lie in the range [0,96-1] that is, [0,95] and 95 does lie within the range.

So, no trap will be produced.

$$\text{Physical Address} = 1952 + 95 = 2047$$

## THRASHING

## INPUT/OUTPUT

## ACCESSING I/O CONTROLLERS

Each controller has a few registers that are used for communicating with the CPU.

By writing into these registers, the OS can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action.

By reading from these registers, the OS can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the OS can read and write.

The issue thus arises of how the CPU communicates with the control registers and also with the device data buffers.

3 approaches are there

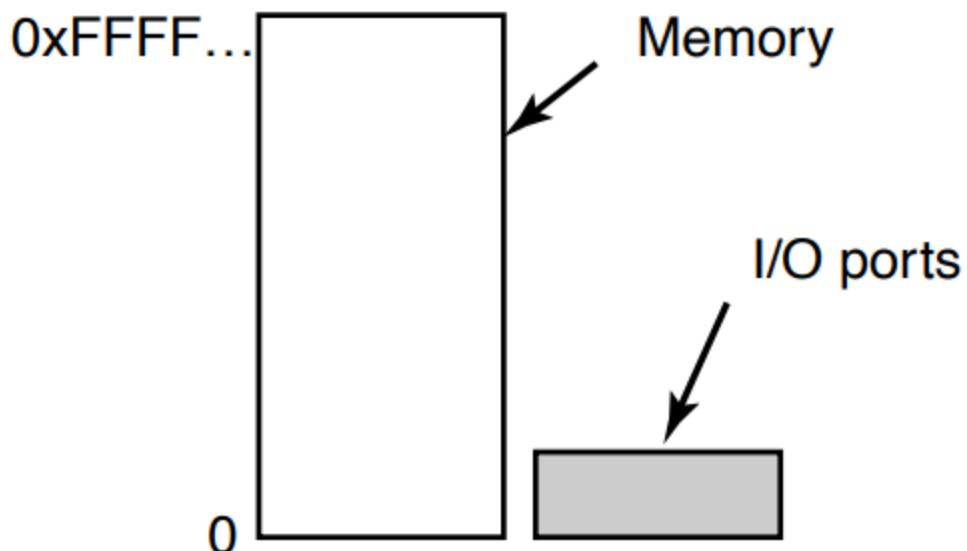
- Separate I/O and Memory Space
- Memory Mapped I/O
- Hybrid

## SEPARATE I/O & MEMORY SPACE

Each control register is assigned an I/O port number, an 8 or 16 bit integer. The set of all I/O ports form the I/O port space, which is protected so that ordinary user programs cannot access it.

In this scheme, the address space for memory and I/O are different, as shown in the figure below:

## Two address



The instructions IN R0, 4 and MOV R0,4 are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0, whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples refer to different and unrelated address spaces.

## MEMORY MAPPED I/O

The second approach is to map all the control registers into the memory space as shown in the figure below:

## One address space



Each control register is assigned a unique memory address to which no memory is assigned. In most systems, the assigned addresses are at or near the top of the address space.

## ADVANTAGES OF MEMORY MAPPED I/O

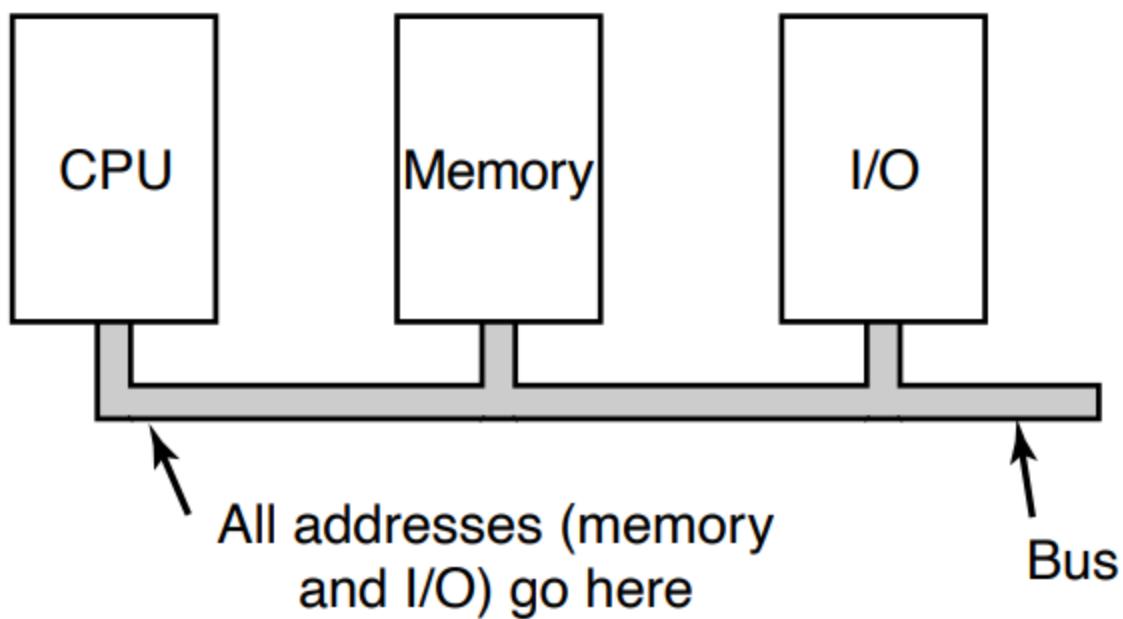
- Device control registers are just variables in memory and can be addressed in C the same way as any other variables. Thus, with memory mapped I/O, an I/O device driver can be written entirely in C. Without memory mapped I/O, some assembly code is needed.

- With memory mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the OS has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space.
- Every instruction that can reference memory can also reference control registers.

## DISADVANTAGES OF MEMORY MAPPED I/O

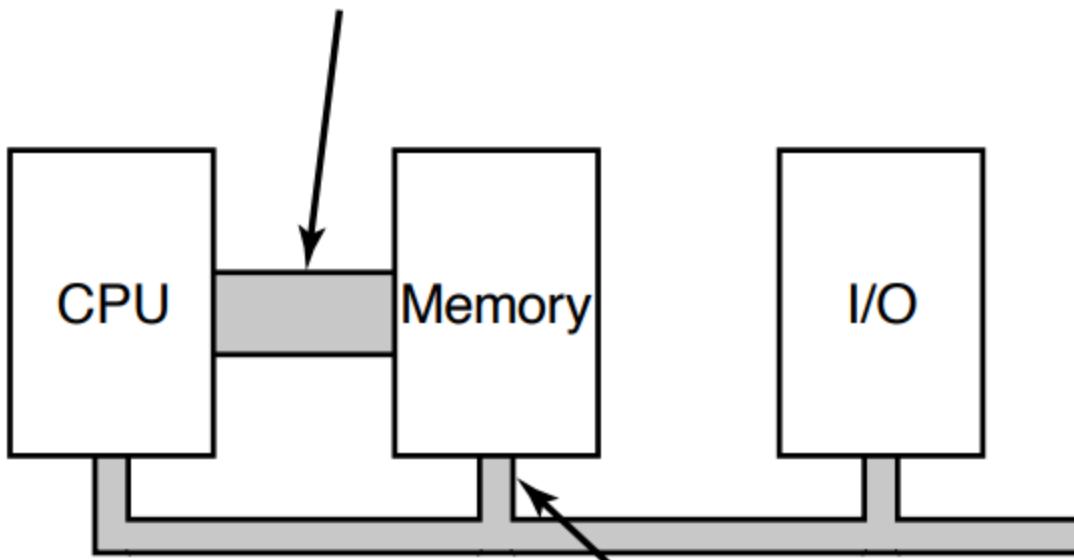
- Most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous.
- If there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to.

If the computer has a single bus, as in the figure below, having everyone look at every address is straightforward.



However, in modern computers, there is a dedicated high speed memory bus, as shown in the figure below:

CPU reads and writes of memory go over this high-bandwidth bus



This memory port is to allow I/O devices access to memory

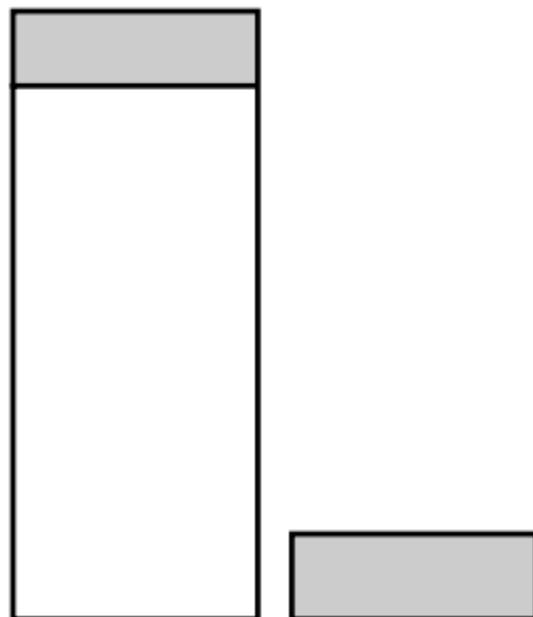
The bus is tailored to optimize memory performance, with no compromises for the sake of slow I/O devices.

The trouble with having a separate memory bus on memory-mapped machines is that the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding to them.

## HYBRID

A hybrid scheme, with memory mapped I/O data buffers and separate I/O ports for the control registers, is shown in the figure below:

**Two address spaces**



The x86 uses this architecture, with addresses 640K to 1M-1 being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 to 64k-1.

## DIRECT MEMORY ACCESS (DMA)

The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU's time. So, a different scheme, called DMA is often used.

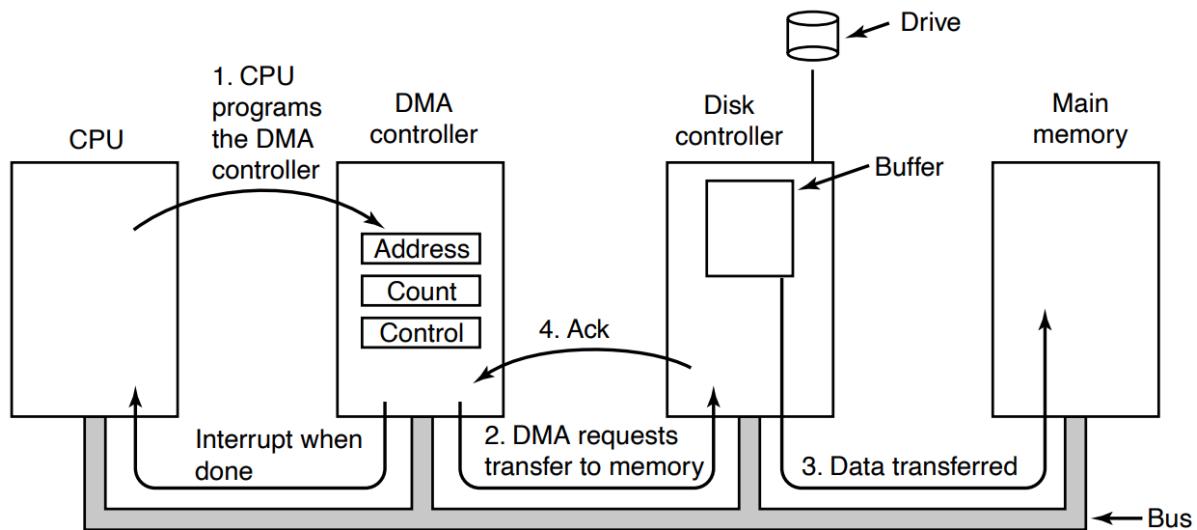
Direct Memory Access is a technique for transferring data within main memory and external device without passing it through the CPU. DMA is a way to improve processor activity and I/O transfer rate by taking-over the job of transferring data from processor, and letting the processor to do other tasks.

The OS can only use DMA if the hardware has a DMA controller.

The DMA controller has access to the System Bus independent of the CPU. It contains several registers that can be written and read by the CPU.

- Memory address register
- A byte count register
- One or more control registers which specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the

transfer unit (byte at a time or word at a time) and the number of bytes to transfer in one burst.



## HOW DMA WORKS?

### Step 1

- The CPU programs the DMA controller by setting its registers so it knows what to transfer where.
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

### Step 2

- The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.

### Step 3

- Typically, the memory address to write to is on the bus' address lines, so when the disk controller fetches the next word from its internal buffer, it knows where to write it.

### Step 4

- When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus.

The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 to 4 are then repeated until the count reaches to 0. At that time, the DMA controller is interrupts the CPU to let it know that transfer is now complete.

## WORD AT A TIME BUS MODE

The DMA controller requests the transfer of one word and gets it. If the CPU also wants the bus, it has to wait.

The mechanism is called CYCLE STEALING because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly.

## BLOCK BUS MODE

The DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called BURST MODE.

It is more efficient than CYCLE STEALING because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition.

The downside of burst mode is that it can block the CPU and other devices for a substantial period if a long burst is being transferred.

## PROGRAMMED I/O

The programmed I/O was the most simple type of I/O technique for the exchanges of data or any types of communication between the processor and the external devices.

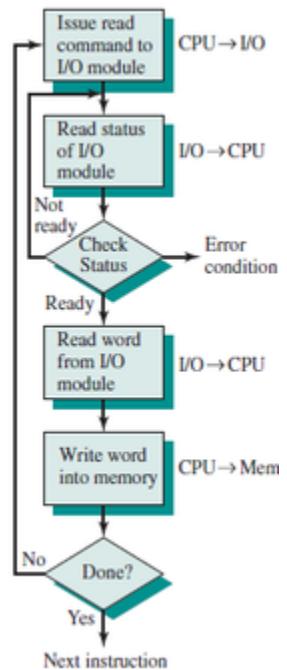
With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.

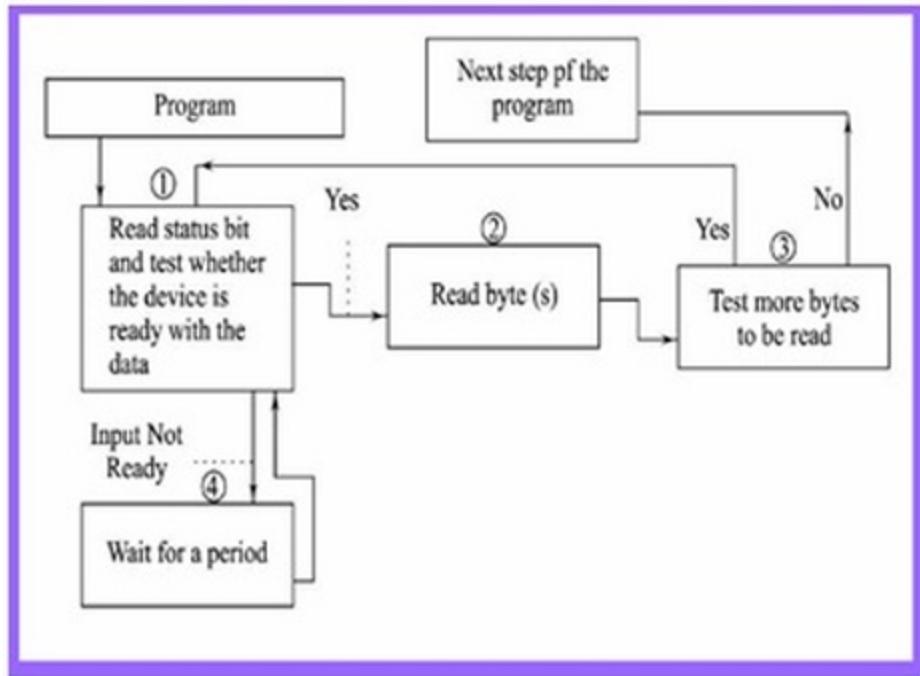
When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time.

## PROGRAMMED I/O INPUT DATA TRANSFER

1. Each input is read after testing whether the device is ready with the input ( a state reflected by a bit in a status register)
2. The program waits for the ready status by repeatedly testing the status bit and till all targeted bytes are read from the input device.

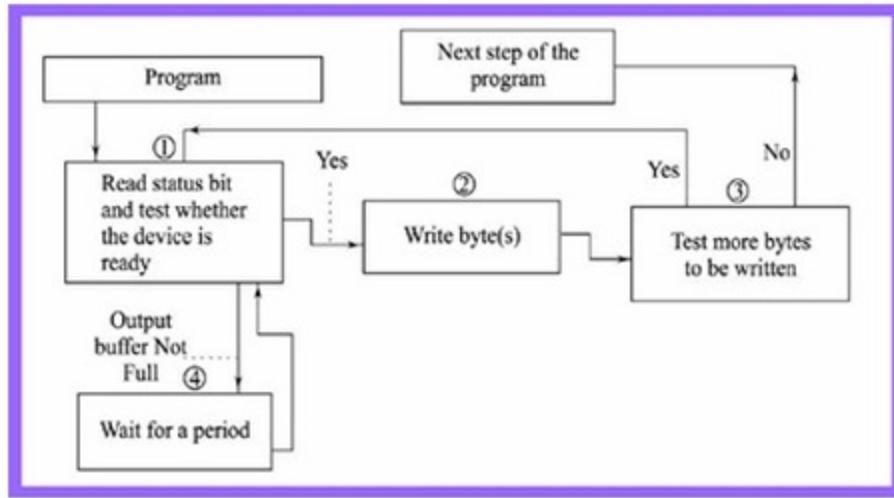
3. The program is in busy state only after the device gets ready, else in wait state.





## PROGRAMMED I/O OUTPUT DATA TRANSFER

1. Each output written after first testing whether the device is ready to accept the byte at its output register or output buffer is empty.



2. The program waits for the ready status by repeatedly testing the status bit(s) and till all the targeted bytes are written to the device.
3. The program in busy (non-waiting) state only after the device gets ready else wait state.

## ADVANTAGES & DISADVANTAGES OF PROGRAMMED I/O

### Advantages

- Simple to implement
- Very little hardware support

### Disadvantages

- Busy waiting
- Ties up CPU for long period with no useful work

## INTERRUPT DRIVEN I/O

Interrupt I/O is a way of controlling input/output activity whereby a peripheral or terminal that needs to make or receive a data transfer sends a signal. This will cause a program interrupt to be set.

The interrupt technique requires more complex hardware and software but makes far more efficient use of the computer's time and capacities.

For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports or memory mapping.

For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

Here the CPU works on its given tasks continuously. When an input is available, such as when someone types a key on the keyboard, then the CPU is interrupted from its work to take care of the input data. The CPU can work continuously on a task without checking the input devices, allowing the devices themselves to interrupt it as necessary.

There are 2 main problems for interrupt I/O, which are:

- There are multiple I/O modules, how should the processor determine the device that issued the interrupt signal?
- How does the processor decide which module to process when multiple interrupts have occurred?

There are 4 main ways to counter these problems, which are:

- Multiple Interrupt Lines
- Software Poll
- Daisy Chain (Hardware Poll, Vectored)
- Bus Arbitration (Vectored)

## OPERATING SYSTEM DESIGN ISSUES

Two objectives are paramount in designing the I/O facility.

1. Efficiency : Efficiency is important because I/O operations often form a bottleneck in a computing system, because most I/O devices are extremely slow compared with main memory and the processor.

One way to tackle this problem is Multiprogramming which allows some processes to be waiting on I/O operations while another process is executing.

However, even with the vast size of main memory in today's machines, it will still often be the case that I/O is not keeping up with the activities of the processor.

Swapping is used to bring in additional ready processes to keep the processor busy, this in itself is an I/O operation.

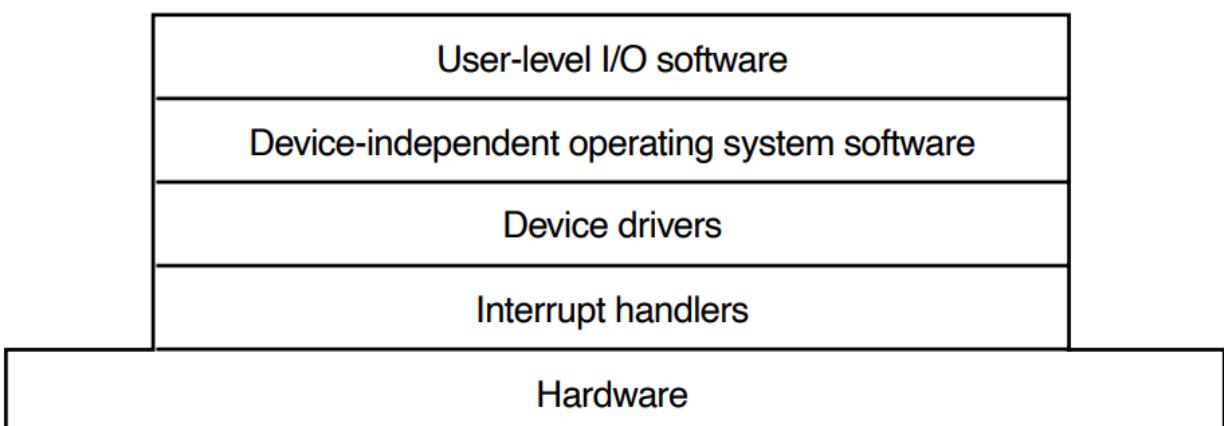
2. Generality: In the interests of simplicity and freedom from error, it is desirable to handle all devices in a uniform manner. This applies both to the way in which processes view I/O devices, and to the way in which OS manages I/O devices and operations. Because of the

diversity of device characteristics, it is difficult in practice to achieve true generality.

What can be done is to use a hierarchical, modular approach to the design of the I/O function. This approach hides most of the details of device I/O in lower level routines so user processes and upper levels of OS see devices in terms of general functions such as read, write, open, close, lock and unlock.

## I/O SOFTWARE LAYERS

I/O software is typically organized in 4 layers as shown in the figure below:



## INTERRUPT HANDLERS

While programmed I/O is occasionally useful, for most I/O, interrupts are an unpleasant fact of life and cannot be avoided.

They should be hidden away, deep in the bowels of the operating system, so that as little of the OS as possible knows about them.

The best way to hide them is to have the driver starting an I/O operation block until the I/O has completed and the interrupt occurs. When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that was waiting for it.

A series of steps must be performed in software after the hardware interrupt has completed.

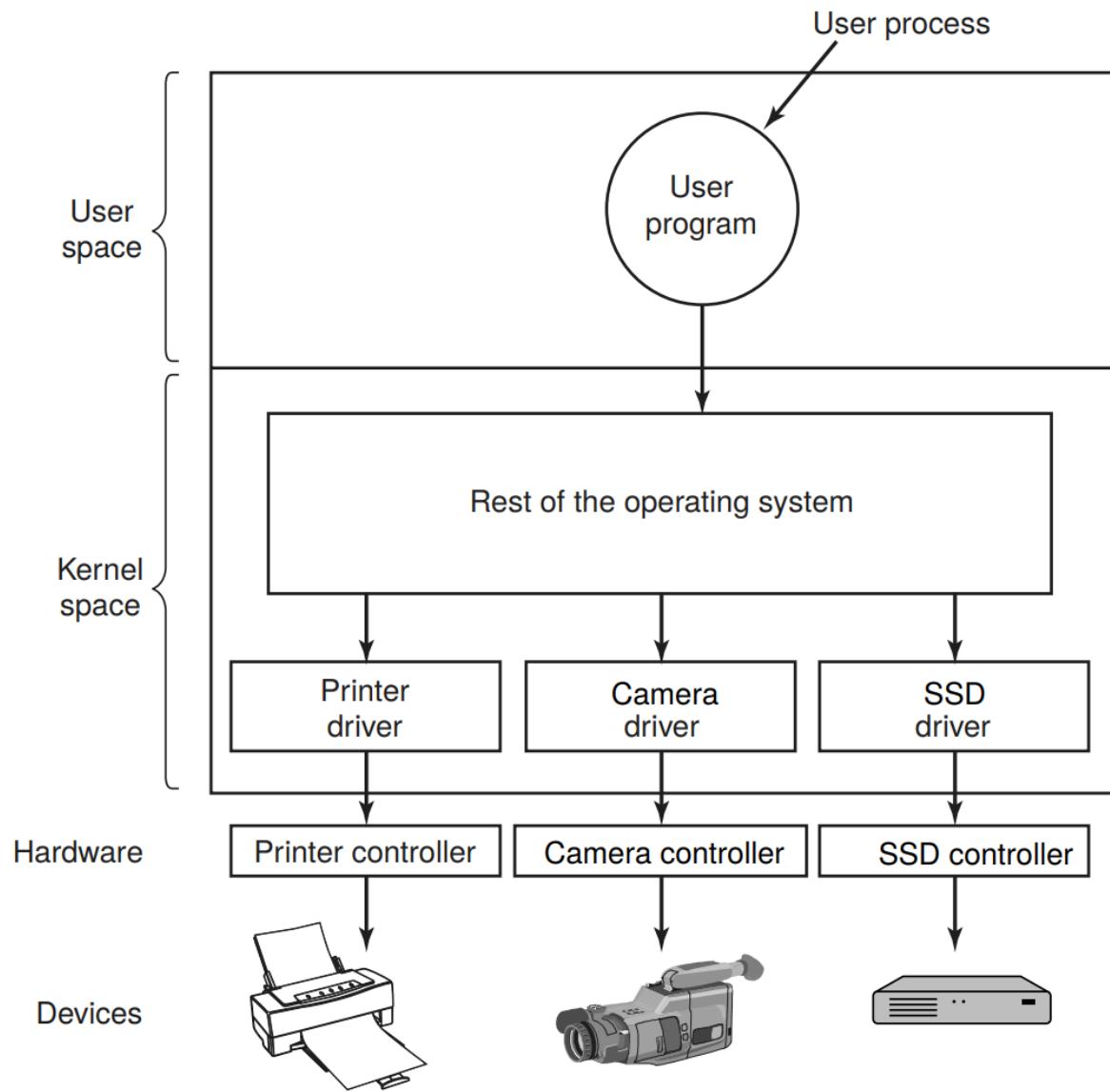
1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt-service procedure.  
Doing this may involve setting up the TLB, MMU, and a page table.

3. Set up a stack for the interrupt service-procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt-service procedure. Typically, it will extract information from the interrupting device controller's registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

## DEVICE DRIVERS

Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the DEVICE DRIVER, is generally written by the device's manufacturer and delivered along with the device. Since each OS needs its own drivers, device manufacturers commonly supply drivers for several popular OS.

Device drivers are normally positioned below the rest of the OS, as is illustrated below:



Most OS define a standard interface that all **BLOCK DRIVERS** (contains multiple data blocks that can be addressed independently) must support and a second interface that all **CHARACTER DRIVERS** (generates or accepts a stream of characters) must support. These

interfaces consist of a number of procedures that the rest of the OS can call to get the driver to do work for it.

The scheme where the OS is a single binary program that contains all of the drivers it will need compiled into it was the norm for years with UNIX systems because they were run by computer centers and I/O devices rarely changed. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary.

OS starting with MSDOS went over to a model in which drivers were dynamically loaded into the system during execution.

The most obvious function of a device driver is to accept abstract read and write requests from the device independent software above it and see that they are carried out. Additionally, the driver must initialize the device, if needed. It may also need to manage its power requirements and log events.

Controlling the device means issuing a sequence of commands to it. After the commands have been issued, one of two situations will apply.

- In many cases, the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it.
- In other cases, however, the operation finishes without delay, so the driver does not need to block.

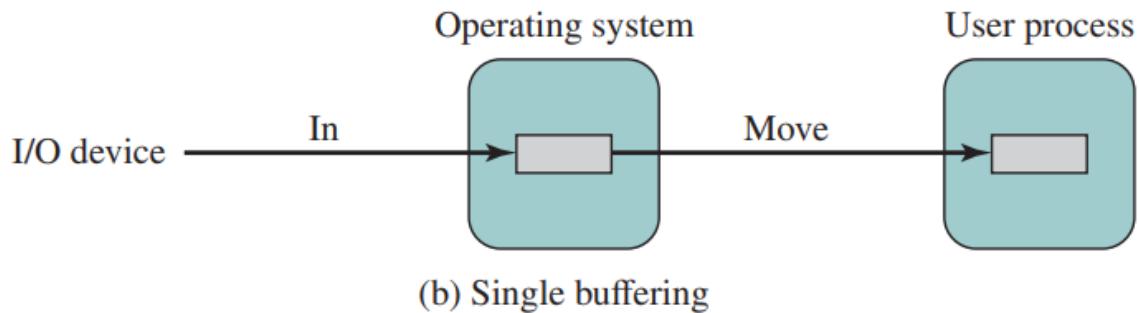
Drivers have to be REENTRANT, meaning that a running driver has to expect that it will be called a second time before the first call has completed.

REENTRANT - code that can be executed by more than one thread or CPU at the same time.

## DEVICE INDEPENDENT I/O SOFTWARE

### SINGLE BUFFER

When a user process issues an I/O request, the OS assigns a buffer in the system portion of main memory to the operation.



For stream-oriented I/O

- Single buffering scheme is used in a Line-at-a-time or a Byte-at-a-time fashion.
- Line-at-a-time operation is appropriate for scroll-mode terminals. With this form of terminal, user input is one line at a time, with a carriage return signaling the end of a line, and output to the terminal is similarly one line at a time.
- Byte-at-a-time operation is used on forms-mode terminals when each keystroke is significant and for many other peripherals such as sensors and controllers.

For block oriented devices

- Input transfers are made to the system buffer.
- When the transfer is complete, the process moves the block into user space and immediately requests another block. This is called **READING AHEAD**, or

anticipated input; it is done in the expectation that the block will eventually be needed.

- The user process can be processing one block of data while the next block is being read in.
- The OS is able to swap the process out because the input operation is taking place in system memory rather than user process memory.
- The OS must keep track of the assignment of system buffers to user processes.

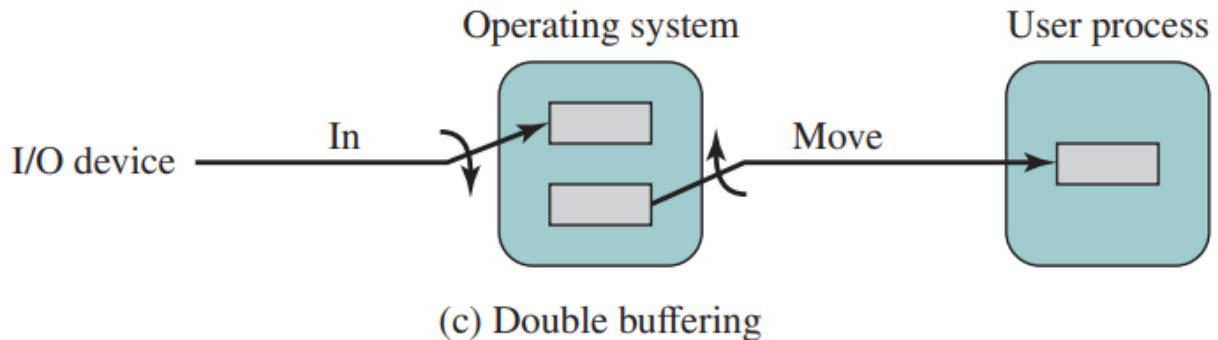
Suppose  $T$  is the time required to input one block, and  $C$  is the computation time that intervenes between input requests.

Without buffering, the execution time per block is essentially  $T+C$ .

With a single buffer, the time is  $\max[C, T] + M$ , where  $M$  is the time required to move the data from the system buffer to user memory.

## DOUBLE BUFFER

An improvement over single buffer can be had by assigning two system buffers to the operation.



A process now transfers data to or from one buffer while the operating system empties or fills the other.

For block oriented transfer

- The execution time =  $\max [C, T]$
- It is therefore possible to keep the block oriented device going at full speed if  $C \leq T$ .
- On the other hand, if  $C > T$ , double buffering ensures that the process will not have to wait on I/O.

For steam-oriented input

- For line-at-a-time I/O, the user process need not be suspended for input or output, unless the process runs ahead of the double buffers.

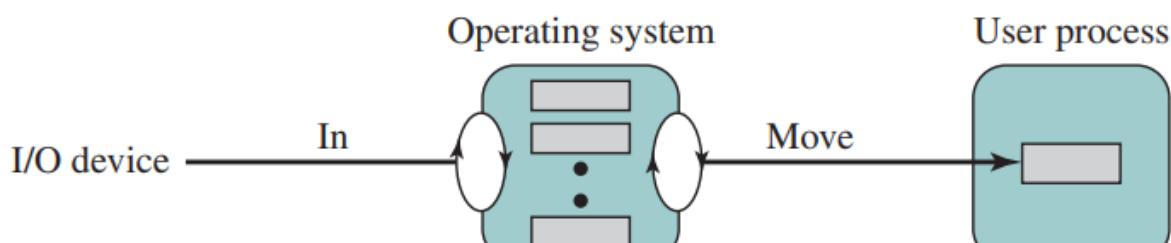
- For byte-at-a-time operation, the double buffer offers no particular advantage over a single buffer of twice the length.

## CIRCULAR BUFFER

A double buffer scheme should smooth out the flow of data between an I/O device and a process. If the performance of a particular process is the focus of our concern, then we would like for the I/O operation to be able to keep up with the process.

Double buffering may be inadequate if the process performs rapid bursts of I/O. In this case, the problem can often be alleviated by using more than two buffers.

When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer, with each individual buffer being one unit in the circular buffer.



(d) Circular buffering

## FILE SYSTEMS

Files are logical units of information created by processes. A disk will usually contain thousands or even millions of them, each one independent of the others.

Processes can read existing files and create new ones if need be.

Information stored in files must be PERSISTENT, that is, not be affected by process creation and termination. A file should disappear only when its owner explicitly removes it.

## FILE NAMING

When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

Most modern systems support file names of up to 255 characters.

Some file systems distinguish between upper- and lowercase letters, whereas others do not. UNIX falls in the first category; the old MS-DOS falls in the second.

Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the file extension and usually indicates something about the file.

<b>Extension</b>	<b>Meaning</b>
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

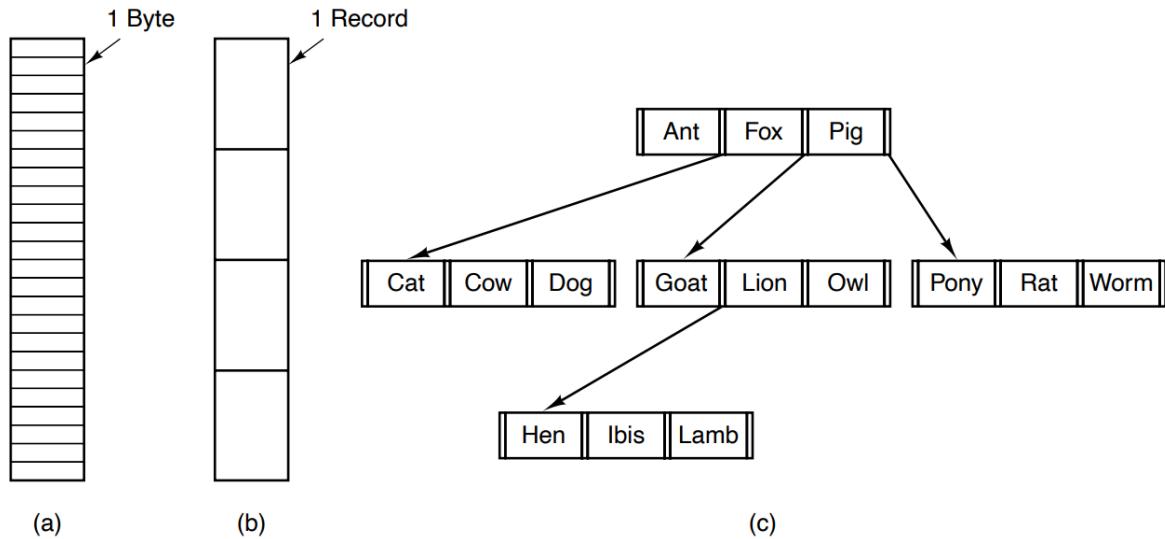
## FILE STRUCTURE

## Byte Sequence

- The file is an unstructured sequence of bytes.
- The OS does not care what is in the file, all it sees are bytes.
- Any meaning must be imposed by user-level programs.
- Both UNIX and Windows use this approach.
- Provides the maximum amount of flexibility.
- The OS does not help, but it also does not get in the way.

## Record Sequence

- A file is a sequence of fixed length records, each with some internal structure.
- Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.



## Tree

- A file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record.
- The tree is sorted on the key field, to allow rapid searching for a particular key.
- The basic operation is not to get the “next” record, although that is also possible, but to get the record with a specific key.

## FILE ACCESS

## Sequential Access

- Early operating systems provided only one kind of file access: sequential access.
- In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order.
- Sequential files could be rewound, however, so they could be read as often as needed.
- Sequential files were convenient when the storage medium was magnetic tape rather than disk.

## Random Access

- When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position.
- Files whose bytes or records can be read in any order are called Random Access Files.
- Random access files are essential for database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without

having to read the records for thousands of other flights first.

- Two methods can be used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position. The latter method is used in UNIX and Windows.

## FILE ATTRIBUTES

<b>Attribute</b>	<b>Meaning</b>
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

The first 4 attributes relate to the file's protection and tell who may access it and who may not.

Hidden files, do not appear in listings of all the files.

The archive flag is a bit that keeps track of whether the file has been backed up recently. The backup program clears it,

and the operating system sets it whether a file is changed. In this way, the backup program can tell which files need backing up.

The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record length, key-position, and key-length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The times keep track of when the file was created, most recently accessed, and most recently modified.

The current size tells how big the file is at present. Some old mainframe operating systems required the maximum size to be specified when the file was created, in order to let the operating system reserve the maximum amount of storage in advance.

## FILE OPERATIONS

1. Create: The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete: When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open: Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. Close: When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5. Read: Data are read from file. Usually, the bytes come from the current position. The caller must specify how

many data are needed and must also provide a buffer to put them in.

6. Write: Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append: This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but some systems have this call.
8. Seek: For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. Get attributes: Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files.

When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes: Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.
11. Rename: This call is not essential because a file that needs to be renamed can be copied and then the original file deleted. However, renaming a 50-GB movie by copying it and then deleting the original will take a long time.

## DISK SCHEDULING ALGORITHM

Disk scheduling is a technique used by the operating system to schedule multiple requests for accessing the disk.

The purpose of disk scheduling algorithms is to reduce the total seek time.

## DISK PERFORMANCE PARAMETERS

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track.

Track selection involves moving the head in a movable head system or electronically selecting one head on a fixed-head system.

On a movable-head system, the time it takes to position the head at the track is known as **Seek Time**.

[Alternative Def: Seek time is the time required to move the disk arm to the required track. The seek time consists of two key components: the initial startup time, and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed.]

In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **Rotational Delay or Rotational Latency**.

[Alternative Def: Rotational delay is the time required for the addressed area of the disk to rotate into a position where it is accessible by the read/write head.]

The sum of the seek time, if any, and the rotational delay equals the **Access Time**, which is the time it takes to get into position to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head, this is

the data transfer portion of the operation. The time required for the transfer is the **Transfer Time**.

Transfer time to or from the disk depends upon the rotation speed of the disk in the following fashion:

$$T = \frac{b}{rN}$$

Where

T = transfer time

b = number of bytes to be transferred

N = number of bytes on a track

r = rotation speed, in revolutions per second

Thus, Total Average Access Time,  $T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$

$T_s$  = average seek time

Consider a disk with an advertised average seek time of 2ms, rotation speed 10,000rpm and 512 Byte sectors with 320 sectors/track.

Suppose we wish to read a file consisting of 2560 sectors for a total of 1.3MB.

First let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 8 adjacent tracks ( 8 tracks \* 320 sectors/track = 2560 sectors). This is known as Sequential Organization.

The time to read the first track is as follows:

Average seek = 2ms

$$r = \text{rotation speed in rps} = 10000/60 = 500/3 \text{ rps}$$

$$\begin{aligned}\text{Rotational Delay} &= (1/2r) \text{ in seconds} = (1/2r)*1000 \text{ in ms} \\ &= 1/(2*500/3)*1000 = 3\text{ms}\end{aligned}$$

$$\begin{aligned}\text{Read 320 sectors} &= b/rN \text{ in seconds} = b/rN * 1000 \text{ in ms} \\ 512*320/(500/3)*(512*320) * 1000 &= 6\text{ms}\end{aligned}$$

$$\text{Total} = (2+3+6) \text{ ms} = 11\text{ms}$$

Suppose the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep

up the flow from the disk. Then, at most, we need to deal with rotational delay for each succeeding track. Thus, each successive track is read in  $3+6 = 9\text{ms}$

To read the entire file, total time =  $11 + (7*9) = 74\text{ms}$

Now, let us calculate the time required to read the same data using random access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have:

Average seek time =  $2\text{ms}$

Rotational Delay =  $3\text{ms}$

Read 1 sector =  $6\text{ms}/320 = 0.01875\text{ms}$

Total time =  $2560 * (2+3+0.01875) = 12848\text{ms}$

## FCFS DISK SCHEDULING ALGORITHM

This algorithm entertains requests in the order they arrive in the disk queue.

It does not cause starvation to any request.

It results in increased total seek time, and is inefficient.

## SOLVED EXAMPLE ON FCFS DISK SCHEDULING

1) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The FCFS scheduling algorithm is used. The head is initially at cylinder number 53. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is = ?

Total head movements incurred while servicing these requests

Sequence:

[53->98->183->41->122->14->124->65->67]

$$\begin{aligned}
 &= (98 - 53) + (183 - 98) + (183 - 41) + (122 - 41) + \\
 &\quad (122 - 14) + (124 - 14) + (124 - 65) + (67 - 65) \\
 &= 45 + 85 + 142 + 81 + 108 + 110 + 59 + 2 \\
 &= 632
 \end{aligned}$$

## SSTF DISK SCHEDULING ALGORITHM

Shortest Seek Time First algorithm services that request next which requires least number of head movements from its current position regardless of the direction.

- It breaks the tie in the direction of the head movement.
- It reduces the total seek time as compared to FCFS.
- It provides increased throughput.
- It provides less average response time and waiting time.
- There is an overhead of finding out the closest request.

- The requests which are far from the head might starve for the CPU.
- It provides high variance in response time and waiting time.
- Switching the direction of head frequently slows down the algorithm.

## SOLVED EXAMPLE ON SSTF DISK SCHEDULING

1) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SSTF scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is = ?

Total head movements incurred while servicing these requests

Sequence

[53->65->67->41->14->98->122->124->183]

$$= (65 - 53) + (67 - 65) + (67 - 41) + (41 - 14) + (98 - 14) + (122 - 98) + (124 - 122) + (183 - 124)$$

$$= 12 + 2 + 26 + 27 + 84 + 24 + 2 + 59$$

$$= 236$$

2) Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence-

4, 34, 10, 7, 19, 73, 2, 15, 6, 20

Assuming that the head is currently at cylinder 50, what is the time taken to satisfy all requests if it takes 1 ms to move from one cylinder to adjacent one and shortest seek time first policy is used?

Total head movements incurred while servicing these requests

Sequence:

[50->34->20->19->15->10->7->6->4->2->73]

$$= (50 - 34) + (34 - 20) + (20 - 19) + (19 - 15) + (15 - 10) + (10 - 7) + (7 - 6) + (6 - 4) + (4 - 2) + (73 - 2)$$

$$= 16 + 14 + 1 + 4 + 5 + 3 + 1 + 2 + 2 + 71$$

$$= 119$$

Time taken for one head movement = 1 msec. So,

Time taken for 119 head movements

$$= 119 \times 1 \text{ msec}$$

$$= 119 \text{ msec}$$

## SCAN DISK SCHEDULING ALGORITHM

This algorithm scans all the cylinders of the disk back and forth. Head starts from one end of the disk and move

towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction and move towards the starting end servicing all the requests in between. The same process repeats. Also known as Elevator Algorithm.

- It does not lead to starvation.
- It provides low variance in response time and waiting time.
- It causes long waiting time for the cylinders just visited by the head.
- It causes the head to move till the end of the disk even if there are no requests to be serviced.

## SOLVED EXAMPLE ON SCAN DISK SCHEDULING

- 1) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SCAN scheduling algorithm is used. The head is

initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is = ?

Total head movements incurred while servicing these requests

Sequence:

[53->65->67->98->122->124->183->199->41->14]

$$\begin{aligned} &= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + \\ &\quad (124 - 122) + (183 - 124) + (199 - 183) + (199 - 41) \\ &\quad + (41 - 14) \end{aligned}$$

$$\begin{aligned} &= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 158 + 27 \\ &= 331 \end{aligned}$$

## C-SCAN DISK SCHEDULING ALGORITHM

Circular scan algorithm is just an improved version of the SCAN algorithm.

Head starts from one end of the disk and move towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction. It then returns to the starting end without servicing any request in between. The same process repeats.

- The waiting time for the cylinders just visited by the head is reduced as compared to the SCAN algorithm.
- It provides uniform waiting time.
- It provides better response time.
- It causes more seek movements as compared to SCAN algorithm.
- It causes the head to move till the end of the disk even if there are no requests to be serviced.

## SOLVED EXAMPLE ON C-SCAN DISK SCHEDULING

- 1) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The C-SCAN scheduling algorithm is used. The head is

initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is = ?

Total head movements incurred while servicing these requests

[53->65->67->98->122->124->183->199->0->14->41]

$$\begin{aligned} &= (65 - 53) + (67 - 65) + (98 - 67) + (122 - 98) + \\ &(124 - 122) + (183 - 124) + (199 - 183) + (199 - 0) \\ &+ (14 - 0) + (41 - 14) \end{aligned}$$

$$= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 199 + 14 + 27$$

$$= 386$$

## LOOK DISK SCHEDULING ALGORITHM

Look algorithm is another improved version of the SCAN algorithm.

Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between. After reaching the last request at the other hand, head reverses its direction. It then returns to the first request at the starting end servicing all the requests in between. The same process repeats.

- It does not cause the head to move till the ends of the disk when there are no requests to be serviced.
- It provides better performance as compared to SCAN algorithm.
- It does not lead to starvation.
- It provides low variance in response time and waiting time.
- There is an overhead of finding the end requests.
- It causes long waiting time for the cylinders just visited by the head.

## SOLVED EXAMPLE ON LOOK DISK SCHEDULING

- 1) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The

LOOK scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. The total head movement (in number of cylinders) incurred while servicing these requests is = ?

Sequence

[53->65->67->98->122->124->183->41->14]

Total head movements incurred while servicing these requests

$$= (65-53) + (67-65) + (98-67) + (122-98) + (124-122) + (183-124) + (183-41) + (41-14)$$

$$= 299$$

## RAID

The RAID scheme consists of 7 levels, zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

- RAID (Redundant Array of Independent Disks) is a set of physical disk drives viewed by the OS as a single logical drive.
- Data are distributed across the physical drives of an array in a scheme known as striping.
- Redundant disk capacity is used to store Parity Information, which guarantees data recoverability in case of a disk failure.

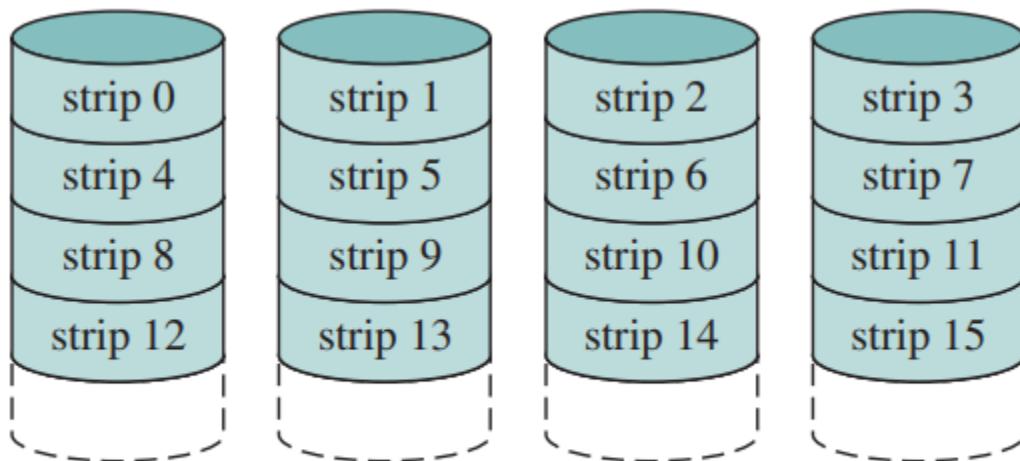
## RAID LEVEL 0

RAID level 0 is not a true member of the RAID family, because it doesn't include redundancy to improve performance or provide data protection.

The data are striped across the available disks. All user and system data are viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round robin to consecutive physical disks in the RAID array. A set of logical consecutive strips that maps

exactly one strip to each array member is referred to as a Stripe. In an n-disk array, the first n logical strips are physically stored as the first strip on each of the n disks, forming the first stripe, the second n strips are distributed as the second strips on each disk and so on.

The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to n strips for that request can be handled in parallel, greatly reducing the I/O transfer time.



(a) RAID 0 (nonredundant)

## RAID LEVEL 1

In RAID 1 redundancy is achieved by the simple expedient of duplicating all the data. Each logical strip is mapped to

two separate physical disks so every disk in the array has a mirror disk that contains the same data.

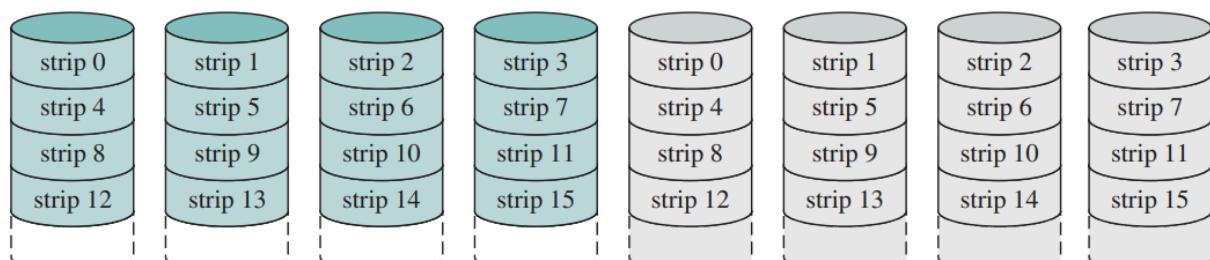
There are a number of positive aspects to the RAID 1 organization:

- A read request can be serviced by either of the two disks that contain the requested data, whichever one involves the minimum seek time + rotational latency.
- A write request requires both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no “write penalty” with RAID 1. RAID levels 2 through 6 involve the use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as update the actual strip in question.
- Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the disk that it supports.

Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time backup of all data so in the event of a disk failure, all of the critical data is still immediately available.

In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double of that of RAID 0. However, if a substantial fraction of the I/O requests are write requests, then there may be no significant performance gain over RAID 0. RAID 1 may also provide improved performance over RAID 0 for data transfer-intensive applications with a high percentage of reads. Improvement occurs if the application can split each read request so both disk members participate.



(b) RAID 1 (mirrored)

[[DATABASE E PORSILAM](#)]

