Virtualizing the CPU

Processes and Process as an abstraction for a Program Running

A Process is basically a running Program, A Program by itself is a lifeless thing, it just sits on the disk having a bunch of instructions and some static data. It is the responsibility of the OS to run this program, and transform the program into something useful.

(What is a disk after all: https://www.lenovo.com/in/en/glossary/what-is-a-disk/#:~:text=A%20disk%20is%20a%20storage,for%20long%2Dterm%20data%20storage)

Programs can be of all kinds, not just the ones you write, It can include music players, Games, Web Browser etc.

And we might want to run multiple programs simultaneously. However there is only one or a few CPUs available on a computer, so how can we support multiple programs running concurrently?

In other words how can the OS provide an illusion of many CPUs? The OS does this by virtualizing the CPU. By running one process, stopping it, and running another and so forth the OS can create the illusion that many virtual CPUs exist, while in reality a single CPU or a few CPUs are present. This technique is known as the Time Sharing of CPU.

OS Achieves CPU virtualization by having:

1. Low Level machinery, called mechanisms like Context Switching
2. High Level Intelligence, like policies (Eg. Scheduling Policies)

Important Definitions

1. Time Sharing: It is a technique used by the OS to share a resource, by allowing the resource to be used for a little while by one entity, then for some time by another entity and so on. The resource in question could be the CPU, network link etc, i.e. resources that are shared by many.
2. Space Sharing: When a resource is divided in space between the various entities that wish to use that resource. For example a disk space.

What constitutes a process

1. Address Space: It is the memory that the process can address, it includes the instructions which lie in memory, and the data that the running program reads and writes.
2. Registers:
    a. Program Counter: (also called Instruction Pointer) to keep track of which instruction to run next.
    b. Stack Pointer (or Frame Pointer): So that the process can access the function parameters, local variables and return addresses. Stack pointer is a 16 bit register that stores the memory address of the stack's top location, i.e. it points to the top of the stack, from where the next value will be popped or to where the next value will be pushed. Since each process has its own stack, hence it will have its own stack pointer as well.
3. I/O Information: Such as a list of open files.

How does a Program gets transformed to a process, a deep dive

1. Programs initially reside on the disk (HDD or SSD), these programs include the Code and any static data, the programs are stored in an executable format.
2. To run the program, the Operating System needs to load the program into the memory, into the Address space of the process.

3. How this loading happens depends on the OS, earlier Operating Systems loaded the programs in an eager manner, i.e. load all the data before running the program, however modern OS use lazy loading, i.e. load pieces of Code or data only as they are needed.
4. Once the program is loaded, the OS needs to do the following tasks:
    a. Allocate memory for program's Run Time Stack. The stack is used by programs to store local variables, function parameters and return addresses.
    b. Allocate memory for the program's heap. The heap is used for dynamic memory allocation, via calls to malloc, or free. The heap will be small at first, however as more and requests for memory via malloc come in, the OS will allocate more memory to the heap.
5. I/O Initialization: For example in UNIX systems, each process by default has 3 open file descriptors, for standard input, output and error.
6. Finally, the OS starts the program execution by jumping to the main() routine, OS transfers control of the CPU to the newly created process and thus the program execution begins.

Process States

A process can be in one of the following states:

1. Running: Process is running on a processor, i.e. it is executing its instructions.
2. Ready: The process is ready to run, but for some reasons, the OS has not chosen to run it at the given moment.
3. Blocked: The process is blocked, i.e. it is waiting for some event to complete, for example a process that has initiated an I/O request becomes blocked and other process can use the processor.

Moving a process from Ready to Running State is called Scheduling
While moving a process from Running to Ready state is called Descheduling

If a process becomes blocked (for example, by initiating an I/O request), then the OS will keep the process in the same state until an event occurs, for example I/O completion, at that point the process moves back to the Ready state.

Example for when a process might become blocked:

1. Initiating an I/O request
2. Reading from the disk
3. Waiting for a packet from a Network.

Additional Process States:

a. Initial State: Process is in the initial state when it is being created.
b. Final / Zombie State: The process has exited but has not been cleaned up by the OS. The Final state can be useful, as it allows the parent process to examine the return code of the child process, and check if the just-finished process executed successfully.

Process List: The Process List (or task list) is a data structure used by the OS to keep track of all the running programs in the system. The Individual entries of this process list are called Process Control Blocks (PCBs). The OS needs to track some pieces of information about each process. For example xv6 tracks the following information.

// the information xv6 tracks about each process, this is what a PCB in xv6 looks like:
// including its register context and state
struct proc {
    char *mem; // Start of process memory
    uint sz; // Size of process memory
    char *kstack; // Bottom of kernel stack for this process
    enum proc_state state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    void *chan; // If !zero, sleeping on chan

```
    int killed; // If !zero, has been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    struct context context; // Switch here to run process
    struct trapframe *tf; // Trap frame for the current interrupt
};
```

What is the Kernel stack?
Kernel stack is a per-process memory region maintained in kernel memory, that is used as the stack for execution of functions called internally during the execution of a system call. Note kernel memory is protected memory which can only be accessed when the processor is running in the Kernel mode. Kernel stack is different from the user stack (commonly called stack) which resides in unprotected user memory.

Additional Notes: Orphan and Zombie Processes

Zombie: When a process exits, some process must “wait” on it to get its exit code. The exit code is stored in the process table (PCB), exit code indicates if the process executed successfully or not. The act of reading an exit code by the parent process is called “reaping” the child. Between the time a child exits and is reaped, it is known as a zombie process. This zombie process has completed execution, but has not been cleaned up by the OS yet, and hence continues to occupy space in the process table.

Orphan: If a process exits with its children still running, then the child processes are considered as orphan processes. Orphan processes are immediately adopted by the “init” process (systemd), i.e. the init process will act as the new parent of the orphan process. “init” will automatically reap its children, hence preventing them from becoming zombies.

https://stackoverflow.com/questions/20688982/zombie-process-vs-orphan-process

Process Creation in UNIX systems: fork, exec and wait

=> Fork()

The fork() system call is used to create a new process
Sample Fork Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
      printf("Main Program Initiation\n");
      printf("Parent Process PID:%d\n", getpid());

      int rc = fork();
      if(rc < 0) {
            fprintf(stderr, "Fork Failed\n");
            exit(1);
      } else if(rc == 0) {
            // Inside Child Process
            printf("Child Process PID:%d\n", getpid());
      } else {
            // Parent Process
            printf("Parent Process PID:%d, and I am the Parent of
%d\n", getpid(), rc);
      }

      return 0;
}


% gcc test.c
% ./a.out
Main Program Initiation
Parent Process PID:70814
Parent Process PID:70814, and I am the Parent of 70815
Child Process PID:70815
```

The fork system call returns a value of 0 to the child process, while the parent process receives the PID of the newly created child.

The process created by fork() is an almost exact copy of the calling process, as it inherits a copy of the same CPU registers, Program Counter, and the same open files as the parent process.

=> Wait()

The wait system call is used by the parent to wait till the execution of the child process is completed. wait() will delay the execution of the parent process as wait() system call will not return until the child process has run and exited.
Using a wait() system call we can make the execution order / output deterministic as the child process will always run first. This is because the parent is waiting for the execution of the child to be completed.

Sample Code using wait system call:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
      printf("Main Program Initiation\n");
      printf("Parent Process PID:%d\n", getpid());

      int rc = fork();
      if(rc < 0) {
            fprintf(stderr, "Fork Failed\n");
            exit(1);
      } else if(rc == 0) {
            // Inside Child Process
            printf("Child Process PID:%d\n", getpid());
      } else {
            // Parent Process
```

```
            // Wait for Child Process to run first
            int rc_wait = wait(NULL);
            printf("Parent Process PID:%d,"
                     " and I am the Parent of %d,"
                     " Waiting for process: %d\n",
                    getpid(), rc, rc_wait);
      }

      return 0;
}


Main Program Initiation
Parent Process PID:77002
Child Process PID:77003
Parent Process PID:77002, and I am the Parent of 77003, Waiting
for process: 77003
```

=> Exec()

The exec system call is useful when you want to run a program that is different from the calling program inside the child process.
The process created via fork() will just run a copy of the calling program , i.e. calling fork alone is useful only if you want to keep running copies of the same program.

Given the name of an executable (e.g. wc) and some arguments, exec will load the code (and static data) from that executable and overwrite the current code segment (and the current static data) with it. It then re-initialises the heap, stack and other parts of the address space of the process. Now, the OS can run the program, passing in any arguments as the argv of the process.

exec does not create a new process, it transforms the currently running program inside the process into a different running program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
	printf("Main Program Initiation\n");
	printf("Parent Process PID:%d\n", getpid());

	int rc = fork();
	if(rc < 0) {
		fprintf(stderr, "Fork Failed\n");
		exit(1);
	} else if(rc == 0) {
		// Inside Child Process
		printf("Child Process PID:%d\n", getpid());

		// Use exec to run a different program inside the child process
		char* args[3];
		args[0] = "wc"; // name of executable
		args[1] = "test.c"; // arguments
		args[2] = NULL; // marks end of array

		execvp(args[0], args);
	} else {
		// Parent Process
		// Wait for Child Process to run first
		int rc_wait = wait(NULL);
		printf("Parent Process PID:%d,"
			" and I am the Parent of %d,"
			" Waiting for process: %d\n", getpid(), rc, rc_wait);
	}

	return 0;
}
```

```
Main Program Initiation
Parent Process PID:79197
Child Process PID:79198
```

```
      36     123     833 test.c
Parent Process PID:79197, and I am the Parent of 79198, Waiting
for process: 79198
```

Do note, since exec changes the currently running program hence any statements beyond the exec call in the original program will not be executed.

execl and execlp: The 'l' stands for a 'list of arguments'
execl - Pass full file path to the executable. Arguments are passed directly
execlp - Use the path variables (environment variables). Arguments are passed directly.

execvp: The 'v' stands for a vector of arguments'

Remember in the list / vector of arguments, the executable should always be the first argument.

execlp("ping", "ping", "-c 4", "www.google.com", NULL);

Interprocess Communication using Pipes

Pipe is basically an in-memory file, which processes can read and write to.
The pipe system call provides 2 file descriptors, one for reading (read-end of the pipe), and one for writing (write-end of the pipe).
The file descriptors are inherited by the child process, i.e. it needs to manage the descriptors independently of the parent process.
fd[0] is the read-end of the pipe
fd[1] is the write-end of the pipe
Closing fd[0] or fd[1] in the child process does not affect the corresponding descriptor in the parent and vice-versa.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
int main(int argc, char* argv[]) {
    int fd[2];
    // fd[0] - read end
    // fd[1] - write end
    if(pipe(fd) != 0) {
        fprintf(stderr, "Pipe creation error\n");
    }

    int id = fork();
    if(id < 0) {
        fprintf(stderr, "Pipe creation error\n");
    } else if(id == 0) {
        close(fd[0]);
        int x;
        printf("Please provide the input\n");
        scanf("%d", &x);
        write(fd[1], &x, sizeof(int));
        close(fd[1]);
    } else if(id > 0) {
        close(fd[1]);
        int y;
        read(fd[0], &y, sizeof(int));
        printf("Parent process, i have read the number it is:
%d\n", y);
        close(fd[0]);

        wait(NULL);
    }
}
```

$./a.out

Please provide the input

45

Parent process, i have read the number it is: 45

FIFOs (Named Pipes)

Pipes can only be used if the processes are in the same hierarchy, to communicate b/w any two arbitrary processes we can use FIFOs (or named pipes).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>

int main(int argc, char* argv[]) {
    if(mkfifo("myfifo1", 0777) == -1) {
        if(errno != EEXIST) {
            fprintf(stderr, "File cannot be created\n");
        }
    }

    int fd = open("myfifo1", O_WRONLY);
    int x = 97;
    if(write(fd, &x, sizeof(int)) == -1) {
        fprintf(stderr, "write failed\n");
    }

    close(fd);
}
```

Side: UNIX Shell

The shell is basically just a user program, it shows a prompt and waits for the user to type something into it. When the user types a command (executable, plus any arguments), the shell figures out where in the file system the executable resides, then calls fork() to create a new child process to run the command, and calls exec to load the executable into the new process. The shell then calls wait(), to wait for the execution of the newly created child

process to complete. When the child completes, the wait() system call returns, and the shell prints out a prompt again, ready for your next command.

Output Redirection in UNIX Shell

A file descriptor is just an integer value, which is private or unique per process, i.e. a process cannot have 2 files with the same file descriptor. However two different processes can have the same file descriptors. Think of a file descriptor as a pointer to an object of type FILE.

Once we create a file, we can read / write to it by using the file descriptor of that file, given that we have the permission to do so.

In UNIX systems, whenever we create a process, certain file descriptors are automatically opened for us.

File Descriptor	File
0	STDIN
1	STDOUT
2	STDERR

These file descriptors are opened by default, when we create a process.

The separation b/w fork and exec allows the shell to alter the environment of the about to be run program, this is useful in case of Output Redirection. Example: prompt> wc p3.c > newfile.txt

Here we would like the output of “wc p3.c” to be redirected to newfile.txt, rather than the screen (Standard Output). The way shell accomplishes this is by closing the Standard Output and opening the file newfile.txt, before the call to exec. We can also use the dup2 system call, to achieve this by duping

the file descriptor of the newly opened file to the file descriptor for STDOUT_FILENO (1).

Sending Signals to a Process

The kill() system call is used to send signals to the processes, including directives to die, pause etc. Processes must use the signal() system call to catch various signals. When a process receives a signal it suspends its normal execution and runs a particular piece of code in response to the signal.

Keystroke Combinations and the signals sent:
Ctrl + C: SIGINT (interrupt) - Terminates the process
Ctrl + Z: SIGTSTP (stop) - Pauses the process mid execution, can be resumed later with fg command.

```
int main(int argc, char* argv[]) {
    int rc = fork();
    if(rc < 0) {
        fprintf(stderr, "Fork failed\n");
    } else if(rc == 0) {
        while(1) {
            printf("Some thing is fox here\n");
            usleep(50000);
        }
    } else {
        sleep(1);
        kill(rc, SIGKILL);
        wait(NULL);
    }
}
```

Signal handling:

```c
#define _XOPEN_SOURCE 700

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>

void handle_sigusr(int sig) {
    printf("Hint: Your are a moron\n");
}

int main(int argc, char* argv[]) {
    int id = fork();
    if(id < 0) {
        fprintf(stderr, "Fork failed\n");
    } else if(id == 0) {
        sleep(5);
        kill(getppid(), SIGUSR1);
    } else {
        struct sigaction sa = {0};
        sa.sa_handler = &handle_sigusr;
        sa.sa_flags = SA_RESTART;
        sigaction(SIGUSR1, &sa, NULL);

        int x;
        printf("what is the value of 3 * 5?\n");
        scanf("%d", &x);
        if(x == 15) {
            printf("Right\n");
        } else {
            printf("Wrong\n");
        }
```

```
        wait(NULL);
    }
}
```

Process arguments and environment

Command line arguments can be passed to the main function via 2 function parameters - argc and argv.

argc is the count of arguments
while argv is an array of character pointers, where each such string is a cli argument. argv[0] is the name of the program, and argv is null-terminated.

Each process has an environment list, this is a list of strings of the form “name:value”, where name is the identifier of the environment variable. Thus an environment variable is composed of a name and a value. The environment consists of several such name, value pairs. When a new process is created it inherits a copy of its parent’s environment. Thus environment variables can serve as a primitive mechanism of interprocess communication. Use the environ array to access the environment variables.

The argv and environ arrays, as well as the strings they initially point to, reside in a single contiguous area of memory just above the process stack.

Limited Direct Execution

We need to virtualise the CPU in an efficient manner (performance) while retaining control over the system.

Consideration 1: Restricted Operations -
During its course of execution the process might need to perform some restricted operations such as I/O or gaining additional system resources like CPU or memory. To achieve this a straightforward approach would be to give the process complete control over the system, and allow it to do whatever it wants, however this would prevent the OS from having control over the system.

The approach taken by OS is to introduce a new processor mode called the User Mode. Code running in the User mode is restricted, and cannot perform actions like I/O. If a process running in User Mode tries to perform a restricted action, an exception will be raised and the OS will likely kill the process.

On the contrary, Kernel Mode is the mode in which the OS or the Kernel runs in. The code running in this mode can perform whatever actions it wants, including restricted actions, or any privileged operations.

User and Kernel Mode can also be viewed with respect to access to hardware resources. In user mode the processes do not have full access to hardware resources. However in Kernel mode the processes have full access to all the hardware resources of the system.

The User mode and Kernel Mode are two different CPU modes.

How can a process in User Mode perform Privileged operations like I / O or reading from the disk

To achieve this we use system calls. System calls allow the Kernel to carefully expose some pieces of functionality to user programs, like I/O, creating / destroying processes, etc.

How are system calls executed: To perform a system call, the Program must execute a special **'Trap'** instruction. The Trap instruction jumps into the kernel (OS Code) and raises the privilege level to kernel mode. The operating system can now perform whatever privileged operations are needed, thus doing the required job for the calling process. When finished, the OS executes a special '**return-from-trap'** instruction, which returns into the calling user program and resets the privilege level to User mode.

How does the Trap instruction know which code to run inside the OS?
This is done by using a Trap Table. The Kernel sets up the Trap Table at the boot time. When the machine is booting it does so in the privileged (kernel) mode. One of the first things which the OS does is to tell the hardware which code to run when certain exceptional events occur - like, Hard Disk Interrupt or keyboard interrupt, or when a program performs a System Call. The OS informs the hardware the location of these Trap Handlers, and hardware will remember this information until the next reboot. Hence the hardware knows what code to jump to when system calls (or other exceptional events) occur.

Notes about System Call / OS Protection

To specify the exact system call, the user code must use a system call number, this needs to be placed in a register. The OS while handling the system call inside the trap handler, examines the system call number, checks if it is valid and if it is then the OS executes the code corresponding to it. This is a level of indirection as the user code cannot specify an exact address to jump to in the OS code, instead it requests a particular service using numbers.

Summary of LDE (Limited Direct Execution) Protocol

LDE can be summarized in 2 broad phases:

1. In the first phase, the Kernel initializes the trap table at the boot time, and the CPU remembers its location for subsequent use.
2. In the second phase, before running the process. The Kernel sets up a few things (allocating a node on the process list, allocating memory) before using a 'return-from-trap' instruction to start the execution of the process, this switches the CPU into User Mode, and it begins running the process. When the process wants to initiate a system call, it traps back into the OS which handles the request and once again returns the control to the process via 'return-from-trap'. When the process completes its work, it returns from main(), and the exit() system call is called which traps back into the OS. At this point the OS does some clean up.

Consideration 2: Switching Between Processes -

How does the OS switch between processes? When a process is running on the CPU, then by definition the OS is not running, so how can the OS regain control of the CPU, so that it can switch between processes.

There are 2 ways to achieve this:

Cooperative Approach

In this approach the OS trusts the processes, and expects them to behave reasonably. Processes that have been running for too long, periodically give up the CPU, so that the OS can decide whether to run some other task.

How can a process give up CPU?
As it turns out most processes transfer control of the CPU to the OS quite frequently by making system calls, e.g. opening a file or creating a new process. Systems which use the cooperative approach, also include an explicit 'yield' system call, which does nothing except transferring control of the CPU to the OS.

Processes also transfer control to the OS, when they do something illegal, for example division by zero, or try to access some memory it doesn't have access to. In such cases the hardware generates a trap to the OS. The OS thus regains control of the CPU, and will likely terminate the offending process.

Non-Cooperative Approach

How can the OS regain control of the CPU even if the processes are not being cooperative? How does the OS prevent a rogue process from taking over the system?

The approach taken by the OS is to use a timer interrupt. A timer device can be programmed to raise an interrupt every few milliseconds. When an interrupt is raised, the currently running process is halted and a pre-configured interrupt handler in the OS runs. At this point the OS has regained control over the CPU and can decide either to continue running the current process or start a different one.

The OS informs the hardware at boot time, which code to run if a timer interrupt occurs. Also at the boot time the OS must start the timer (a privileged operation).

When an interrupt occurs, the hardware needs to save sufficient information (or state) regarding the process that was running when the interrupt occurred. This is done to ensure that a subsequent 'return-from-trap' instruction will be able to resume the program correctly. For achieving this the hardware saves various registers onto the Kernel Stack, which can be restored by the 'return-from-trap' instruction.

The processor will push the program counter (PC), flags and a few other registers onto a per-process kernel stack. The return-from-trap instruction will pop these values off the stack and resume the execution of the process.

Context Switch

When the OS regains the control of the CPU, a decision has to be made whether to continue executing the current process or switch to a different one. If the decision made is to switch to a different process then the OS needs to execute a low level piece of code, which is called Context Switch.

Here is what Context Switch involves

1. Saving the context of the currently running process. The OS executes some low-level assembly code to save the general purpose registers, PC (onto the Kernel Stack) and the ***kernel stack pointer*** of the currently running process.
2. Restores the context of the soon-to-be executing process: OS will restore these general purpose registers, and PC for the soon-to-be-executing process from the kernel stack, and will switch to the kernel stack of this process.
3. By doing the above steps, the OS ensures that when the 'return-from-trap' instruction is executed, instead of returning to the old process, the system will resume the execution of another one.
4. When the OS finally executes the 'return-from-trap' instruction, the soon-to-be-running process becomes the currently running process and the Context Switch is complete.

Context Switch: Details and Terms

During a Context switch there are actually 2 types of register saves which take place:

1. When the timer interrupt occurs, the user registers of the running process are saved to the Kernel stack of the process by the hardware.
2. When the OS decides to switch from process A to process B, the kernel registers of the process are saved to the process structure (or PCB) of the process by the OS.

What are Kernel registers?

Kernel registers refers to a set of registers that are accessible and usable by the Operating System, however a user program (user-mode) cannot access

them. Kernel registers are physically present on the CPU chip, unlike in virtual memory locations.
The kernel registers are used by the OS, and hence can only be accessed in kernel mode, however they are particular to a process. Example of kernel registers: base and bounds registers, kernel registers are used by the OS to enforce privilege levels.

(https://cs.stackexchange.com/questions/96550/whats-the-difference-between-user-registers-and-kernel-registers)

Intricacies of calling it Context Switch: The main point here is that as part of the context switch code (A -> B switch), the OS will update the kernel stack pointer to use the Kernel Stack of B (instead of using Kernel Stack of A). The OS enters a call to the switch code in the context of one process (the one which was interrupted) and returns in the context of another process (soon-to-be-running process).

Context Switch, Example

Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode). In the timer interrupt handler, the OS decides to switch from running Process A to Process B. At that point, it calls the switch() routine, which carefully saves the kernel registers (into the process structure of A), restores the registers of Process B (from its process structure entry), and then switches contexts, specifically by changing the kernel stack pointer to use B's kernel stack (and not A's). Finally, the OS executes the return-from-trap instruction, which restores B's user registers (from its kernel stack) and then process B starts running.

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

Scheduling

We have seen the low level mechanisms like context switch, which are used to run a process, however which process should be run? This is decided by the Scheduler. The Scheduler is a part of the OS that uses high level policies to determine which process should be run at a given moment in time.

Scheduling Metrics: Metrics are used to compare different scheduling policies.

1. Turnaround Time = T(job completion) - T(arrival)
2. Response Time = T(time first scheduled) - T(arrival)

Scheduling Policies

FIFO (First In First Out Scheduling)

Also called First Come First Served Scheduling (FCFS).

- The job which arrives first is run first.
- The jobs will be run till completion, i.e. Non-Preemptive Scheduling.
- Suffers from convoy effect, if a longer job is queued up ahead of shorter jobs. Then the shorter length jobs will need to wait for the longer job to complete, hence increasing the Turnaround Time.
- Easiest to implement but not optimal.

SJF (Shortest Job First Scheduling)

- It runs the shortest job first, then the next shortest and so on.
- ***If all jobs arrive at the same time, then SJF is in fact an optimal scheduling algorithm.***
- Suffers from Convoy effect if jobs arrive at different times (instead of all the jobs arriving together), for example at t = 0, job A arrives, which needs 100 seconds to run, and at t = 10, jobs B and C arrive each needing 10 ms. In this case since at t = 0, no job

other than A is available, hence SJF will pick it and run it to till completion (i.e till t = 100 seconds), processes B and C (shorter jobs) will need to keep waiting for the longer job to complete, hence again increasing turnaround time.

- Non-Preemptive Scheduling, the jobs will run till completion, and no other job can be scheduled during this duration.

STCF / PSJF (Shortest Time to Completion First / Preemptive SJF)

- Adds preemption to SJF
- Any time a new job enters the system, STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.
- Preemptive Scheduler: If the scheduler finds a job with less time left than the currently executing job, then it will stop the one currently running and schedule the other job instead.
- ***STCF is an optimal scheduling algorithm if all jobs only use the CPU. (i.e. do not perform any I/O).***

A note on response time:
Response Time measures interactivity. SJF / STCF optimize turnaround time, however they perform quite badly if the metric considered is Response Time or Interactivity.

This is because to decrease average turnaround time, some jobs might be deferred execution, hence increasing the response time.
Response time measures interactivity and fairness.

T(response) = T(first scheduled) - T(arrival)

RR (Round Robin Scheduling)

Basic Idea: Instead of running a job to completion, we only run the job for a ‘time slice’ (also called scheduling quantum) and then switch to the next job in the run queue. We do so until all the jobs are finished.

Note: Length of the time slice must be a multiple of the timer interrupt period.
RR is an excellent scheduler if Response Time is the only metric we care about. However it performs poorly with regards to the Turnaround time metric. RR is one of the worst policies if our metric is turnaround time.

The length of time slice is critical in RR, the shorter it is the smaller the response time, hence a more interactive system. However if the time slice is too short, then suddenly the cost of context switching will dominate the overall performance. Thus, we need to decide the length of the time slice such that it is long enough to amortize the cost of context switching and not so long that the system becomes unresponsive (high response time).

There is a tradeoff between response time and turnaround time, if you value fairness then response time is lowered however turnaround time increases. On the other hand if you are willing to be unfair then turnaround time decreases but response time increases.

Cost of Context Switching: The cost of context switch arises from the cost of OS saving and restoring some registers as well as a performance cost. As the programs when running, build up some state in CPU caches and TLB, however when the scheduler switches to a new job, this state is flushed out and the newly running job needs to build up its own state, which has a noticeable performance cost.

Incorporation of I/O into scheduling Algorithms

When a process is performing I/O it is in a blocked state and is not using the CPU during the I/O. Thus the scheduler should schedule another job to the CPU during this time.
A scheduler can incorporate I/O by treating each CPU burst of the job as a separate job.

Doing so allows for overlap with the CPU being used by one process while waiting for the I/O of another process to complete. Overlap improves system utilization.
Thus, the scheduler makes sure that interactive processes get run frequently. While these interactive jobs are performing I/O other CPU intensive jobs run, improving processor utilization.

Multilevel Feedback Queue Scheduling
MLFQ uses the following rules for scheduling:

• Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
• Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
• Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
• Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
• Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

Memory Management

Index:

Logical and Physical Address space

Logical Addresses:

- The logical addresses are generated by the CPU during the program execution.
- This is the address seen by a process and is relative to the process's address space.
- User programs can never access real physical addresses, i.e. user programs only deal with logical (or virtual addresses).
- All the addresses generated / referenced by user-level programmes are virtual addresses.
- The set of all logical addresses generated by a program is called a logical address space.

Physical Addresses:

- Physical Address is the location in the actual physical memory where the data is stored
- A user program cannot reference physical memory addresses
- The physical addresses are the addresses seen by the memory unit.
- The set of all physical addresses corresponding to the logical addresses generated by a program is called a physical address space.

For a base value R, if logical addresses lie in the range (0 to max) then the physical addresses will be in the range (R + 0, R + max).

R is the Relocation Register.
In addition we use a limit (bounds) register to prevent a process from accessing memory outside its logical address space. Each logical address should be less than the limit register (bounds).

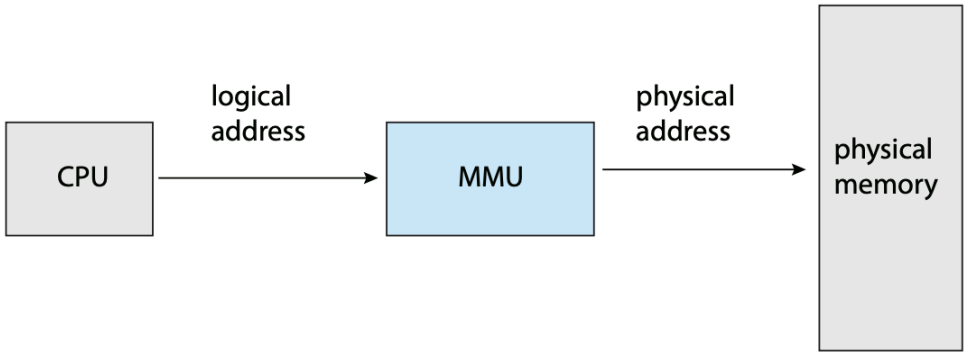
Values of Relocation and Limit Register

Relocation or the base Register is the value we add to the Logical Address to get the corresponding Physical Address.

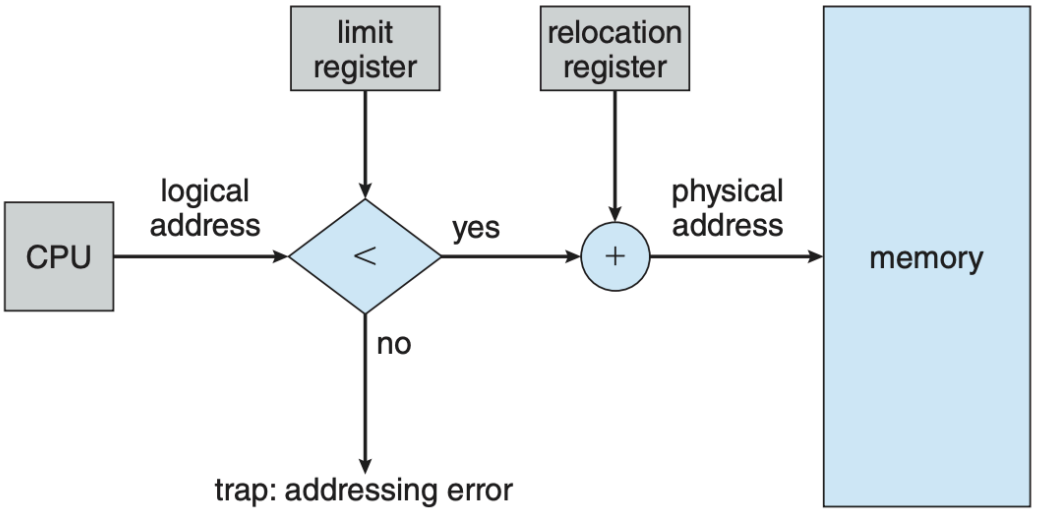
Physical Address = Logical Address + Base / Relocation (register)

Limit (Bounds) register stores the size of the logical address space.

The run time mapping from Logical to Physical Addresses is done by a hardware device called the Memory Management Unit (MMU).



Memory Management Unit



Address Translation

Virtualizing the Memory Introduction

Multiprogramming and Time Sharing

In a multiprogramming OS multiple processes are kept in memory ready to run at any given moment, and the OS switches between them when one of them decides to perform an I/O for example. Doing so increases the CPU utilization and improves efficiency.

Multi Tasking or Time Sharing OS were developed as the notion of interactivity became important. Time Sharing can be achieved by running one process for a short while, then stopping it and saving its state, then loading some other process's state and running it for a while and so on.

However, how do we save / load the state? As we have seen previously we can save and restore the registers relatively fast, but how do we save the physical memory of the process? One way is to save the data to some kind of disk, however saving the entire contents of the memory to the disk is non-performant (too slow), thus the approach taken is to leave the processes in memory while switching between them.

Having multiple programs reside in the memory concurrently, makes protection an important issue as we don't want a process to read or write to another process's memory.

The Address Space

The Address space is an abstraction of physical memory, it is the running program's view of the memory in the system.

The Address space contains all of the memory state of the running program.

1. It includes the code of the program.
2. When the program is running it uses a stack to keep track of where it is in the function call chain, as well as to allocate local variables and store function parameters and return addresses.
3. The heap is used for dynamically allocated memory (user managed memory), it is allocated by using malloc() in C.

The code, stack and heap are all stored in the address space of the program.

Code is static, i.e. it occupies a fixed amount of space. However stack and heap are dynamic and may grow or shrink in size.

The heap and stack grow in opposite directions, as each of them wishes to grow hence putting them at the opposite ends of the address space makes sense.

Address space is just an abstraction of the physical memory, it is the running program's view of memory in the system.

The OS needs to build an abstraction of a private, potentially large address space for each running process, even though in reality they are sharing the same physical memory. This is called Virtualizing the memory.

Goals of Virtual Memory

Transparency: The running programs shouldn't be aware of the fact that the memory is virtualized, rather the program behaves as if it has its own private physical memory.

Efficiency: Virtualization of the memory should be efficiently done, both with respect to time and space.

Protection: The OS needs to protect processes from one another, as well as protecting the OS itself. No process should be able to read / write to another process's memory, i.e. processes should be isolated from each other, unaware of other processes' memory.

The OS with hardware help translates these virtual memory addresses to Physical Memory Addresses.

Contiguous Memory allocation

Introduction

How does the OS manage the physical memory (main)?

In contiguous Memory allocation each process is stored in a single contiguous block of memory.

Types of contiguous Memory allocation:

1. Fixed (or Static) Partitioning: The memory is divided into partitions of equal or different sizes, and a partition can only contain a single process
   Limitations:
   a. Internal Fragmentation: Since each partition can store only a single process, there might be some space left inside a partition if the size of the process is less than that of the partition. This leads to wastage of memory and is called internal Fragmentation.
   b. External Fragmentation: This kind of fragmentation happens when there is enough total memory to accommodate a process, but it can't be accommodated as the memory is not contiguous.
   c. Process Size Limitation: Any process with size greater than the size of the largest partition cannot be loaded into Memory.
   d. Low degree of Multiprogramming: Degree of Multiprogramming is low as at any point the number of processes in memory is limited by the number of partitions.

2. Dynamic (Variable) Partitioning: Initially the RAM is empty and partitions are made during the run time according to the needs of the process.

   Advantages:

   a. The size of the partition will be equal to the size of the process.
   b. Since the size of the partition is the same as the size of the process hence there is no internal fragmentation.
   c. No limit on the size of the process, but still the size of the process needs to be smaller than the RAM.
   d. Better degree of multiprogramming.

   Limitations:

   Still suffers from External Fragmentation.

Base and Bounds (Dynamic Relocation) Technique

Address Translation

Address Translation (or hardware-based address translation, because MMU is a hardware device) refers to the translation of Virtual Memory addresses into Physical Memory addresses by the hardware. Thus on each memory reference, an address translation is performed by the hardware, to redirect to the actual locations in Physical memory where the data is stored.

From the program's perspective it's address space starts at 0 and goes up to a certain size. However to virtualize the memory, the OS wants to place the process somewhere else in physical memory not necessarily at address 0. This is called relocation, and we need to achieve it in a transparent manner i.e. the process should not be aware about it. We need to provide the illusion of an address space starting at 0 to the process running, even though in reality the address space is located at some other physical address.

Techniques for Address Translation

1. Base and Bounds (Dynamic Relocation)
2. Segmentation

Base and Bounds (Dynamic Relocation)

With the Base and Bounds technique we need 2 hardware registers, one is called base register while the other one is called bounds register (or a limit register).
The base and bounds register pair allow us to place an Address space anywhere in the physical memory, and ensure that the process cannot access any memory outside of its address space.

From the program's perspective its Address space starts at address zero. When the program starts executing the OS decides where in physical memory the program should be loaded, and sets the base register to this value. For example if the OS decides to load the process at address 32KB then the base register will be set to this value.

Whenever the process running makes any memory reference, it is translated by the processor (hardware) to physical memory address as follows:

$$\text{Physical Address} = \text{Virtual Address} + \text{Base}$$

The bounds register is to ensure that the memory accesses are within the address space of the process.
The bounds register can be set to:

1. Size of the address space of the process, compare virtual address with the bounds.
2. Physical memory address of the end of the address space, compare physical address (post adding Base) with the bounds.

Base and Bounds registers are hardware structures which are kept on the chip, one pair per CPU.

The part of the processor which deals with Address translation is called the Memory management Unit (MMU).

Free Space Management

Free list: The Free list is a data structure which is used to track ranges of the physical memory which are currently not in use, for finding an empty slot in the physical memory (to fit the Address space) the OS will search the free list.

External Fragmentation and Compaction
External Fragmentation is said to occur when there is enough total memory remaining to accommodate a process, however the process cannot be accommodated as the free memory is not available in a contiguous manner.

To solve the problem of External Fragmentation, Compaction is used. Compaction is the technique of collecting all the free memory fragments into one large contiguous block of free memory; this memory can be used to accommodate other processes.

Compaction is achieved by moving all the processes to one end of the memory and moving all the available free space to the other end of the memory. In this way we can get a large contiguous chunk of free memory.

However, Compaction is an expensive operation, as copying all processes to new addresses will take up a fair amount of processor time. Thus it is an overhead as while doing compaction the CPU will not be able to perform other tasks.

Note: Compaction can only be done when relocation is dynamic and done at execution time, by this what we mean is that the mapping between the Logical address and the physical address should not be

static, and the OS should be able to freely move the process in the physical memory during run time, in effect changing the value of R.

Finding the correct slot (hole) from the free list.
As mentioned before when a process arrives, the OS will search the free list to find a hole of appropriate size, i.e. holes with size >= size of process. However, how do we choose which slot (or hole) to use? There could be many strategies to solve this problem, however the common ones are:

1. First Fit
2. Next Fit
3. Best Fit
4. Worst Fit

First Fit: Allocate the first hole that is big enough. Searching can start either from the beginning or the end of the free list. Search stops as soon as we find a free hole that is big enough.

Next Fit: Similar to First Fit however the search for slot (hole) always starts from the last allocated hole.

Best Fit: Allocate the smallest hole that is big enough. We must search the entire list (unless it is sorted), hence this strategy is slower. It produces the smallest leftover hole (i.e. internal fragmentation is less). Since this strategy might create many small holes in the memory it can cause major External Fragmentation.

Worst Fit: Allocate the largest hole that is big enough. We must search the entire list (unless it is sorted), hence this strategy is slower. It produces the largest leftover hole. Since the leftover hole is quite large it can accommodate other processes in the future.

Internal Fragmentation: Unused memory that is internal to a partition.

User-Level Memory Allocation Libraries

Here we discuss the functioning of the User-Level Memory Allocation Library. The memory allocation library is used for dynamic memory allocation by programs.

For example, in C we can use malloc to acquire memory. This memory is allocated in the heap segment.
To free the acquired memory we use the ‘free’ system call.

When a request for memory allocation comes in, the memory allocation library will search in the free list to find a slot of appropriate size, i.e. one with size >= Requested memory.

For example when we call malloc(1000), then the library will search in the free list to find a slot with size >= 1000, and return a void pointer pointing to this address to the calling program.

Free list Management:
Splitting: After finding a chunk of free memory that can satisfy the request, the allocator will split the chunk into 2 parts, the first chunk will be returned to the caller, while the second chunk will remain on the free list.

Coalescing: Whenever a chunk of memory is freed, the allocator will coalesce the free space. For example if the free list contains the following entries:

Start: 0
Len: 10

Start: 20
Len: 10

Suppose process “A”, which has currently acquired memory in the range 10 to 19 decided to free the allocation, then the free list would look like:

Start: 10
Len: 10

Start: 0
Len: 10

Start: 20
Len: 10

Now, there is 30 bytes of empty space, however if a process comes in and requests for 20 bytes, the request would actually fail. This is because the allocator will search for a suitable sized slot in the free list. However it won’t be able to find any. This is because there is no chunk of 20 bytes or more in the free list.

However in reality we can see there is 30 bytes of contiguous space available.
To fix this, whenever a process frees a chunk of memory, the allocator will coalesce the free space (consider it like merge overlapping intervals).

Hence the free list post coalescing would be

Start: 0
Len: 30

And the request for allocation of 20 bytes by the new process can be satisfied.

Free list management is used inside the heap as well as by the OS for ‘base and bounds’ and ‘segmentation’.

How does Free know how much memory to free?
Corresponding to every allocation in the heap a header is also stored just before the starting address of the allocation. This header stores the allocation size and a Magic number (for integrity testing).

Important Note: Memory allocation inside the heap suffers from external fragmentation, however Compaction of free space is not possible inside the heap, this is because when a program requests for memory, the library will return a pointer to the starting address of the allocation. Now, of course if we perform compaction inside the heap, then some other piece of memory (or free space) could occupy that address which was returned by the memory allocation library to the program (which probably would have been stored by the program, for future references). Of Course if compaction happens behind the scenes, and the program tries to access the original address then that will create major problems.
In summary once memory is allocated to a client inside the heap, it cannot be relocated to another location in the memory.

Non-Contiguous Memory Allocation

Segmentation

Segmentation is a non-contiguous memory allocation technique. It is an extension of the base and bounds method.
First, we see the downfalls of the base and bounds technique

1. Internal Fragmentation: Base and Bounds Method suffers from internal fragmentation, i.e. there is unused memory inside the address space. Now even though the process is not using the space, it is still taking up physical memory. This is because the address space will reside at some address in the physical memory. Thus the base and bounds method leads to a wastage of physical memory. (Base and bounds doesn't suffer from external fragmentation, as it is dynamic relocation hence compaction can be performed).
2. It is not possible to run a program if its entire address space doesn't fit into the memory.

To solve these problems, we use the technique of Segmentation.

Segment - A segment is a contiguous portion of the address space of a particular length. For example in a typical address space, the segments would be 'Code', 'Heap' and 'Stack'

The idea of Segmentation
The idea behind Segmentation is 'instead of having one base and bounds pair per CPU, have one base and bounds pair per logical segment of the address space.'

i.e. have a base and bounds pair for each of the segments of the address space, in the example above we'll have 3 sets of base and bounds register pairs, one for 'Code' segment, one for 'Heap' and one for 'Stack'.

What Segmentation allows the OS to do is place each one of the segments in different parts of the physical memory, independently.

Using Segmentation we can avoid wasting the physical memory, as we are no longer filling the physical memory with unused virtual address space. In other words we do not need to allocate physical memory corresponding to the unused space in the Address space of the process (this unused space lies b/w heap and the stack).

With segmentation we can accommodate processes having large address spaces with large amounts of unused space / memory (this kind of an address space is called a sparse address space) into the physical memory (or main memory).

The Bounds register will store the size of the segment (instead of storing the size of the address space, as it was in the case of the 'base and bounds' method).

If a process performs a memory access to an illegal address on a segmented machine, we get a Segmentation Fault, in other words if a process tries to access memory which is out of bounds for a segment, the hardware will detect this illegal access and will trap into the OS. The Operating system will likely terminate the offending process. This is known as a Segmentation Fault.

Which segment are we referring to

How do we tell the hardware which segment the process is referring to so that the hardware can use the base and bounds pair of that segment for Address Translation.

Method - I
Explicit

In the explicit approach we chop up the logical address space, and use the first few bits of the virtual address to identify the segment, the remaining bits are the offset into the segment.

For example if we have 3 segments, then we can use the first 2 bits to identify the segment while the remaining bits will act as the offset into the segment.

The hardware will compare the offset with the bounds (Remember bounds will store the size of the segment here) to check if the memory access is legal. If it is, it will add the base of the segment to the offset (i.e. translate virtual address to physical address) and fetch the data from that address in physical memory.

Limitations with Explicit Approach

1. Waste of addresses: As we saw above we have 3 segments but have used 2 bits (2*2 = 4 segments) for identifying the segment, as a result we waste an entire segment of addresses.
2. Limiting Segment Size: An issue with using the top few bits for identifying segment is that it limits the use of the virtual address space.
   Each segment is limited to a maximum size. For example if we have a 14 bit virtual address space and we use first 2 bits for identifying the segment, then the maximum size of each segment

will be 4KB (14 bits in total, 2 used for segment hence 12 remaining, 12 bits can be filled in 2^12 ways, i.e. 2^12 addresses = 4096 addresses, and 4096 = 4KB), and it is not possible for a program to grow a segment.

Method - II
Implicit

Hardware determines the segment itself, by noticing how the address was formed. If for example the address was generated from the program counter then the address lies in the code segment. If the address is based off of the stack pointer, then the address lies in the stack segment. Any other address must lie in the heap segment.

Stack Segment
Stack segment grows backwards (i.e. towards lower addresses). The hardware needs to know this fact in order to correctly translate virtual address to physical address.
To achieve this the hardware, in addition to storing the base and bounds pair per segment should also store a bit for each segment indicating if the segment grows in positive direction (i.e. towards higher addresses), the value of this bit:
1: Segment grows in positive direction
0: Segment grows backwards or in negative direction

Thus, the translation of virtual address to physical address for stack segment also happens differently. As mentioned before the first few bits are used to identify the segment, and the remaining bits act as the offset into the segment. However this is the positive offset and for the stack segment we need to find the negative offset. To find the negative offset we subtract the segment size from the positive offset.

i.e. Negative Offset = Positive Offset - Segment Size
Then we add the base to the Negative offset to get the physical address.

Bounds Check:
abs(Negative Offset) <= bounds register.
As mentioned before, the bounds register will store the size of the segment.

Thus the important point to note, for code and heap segment we can directly use the offset into the segment (for bounds check and address translation), but for stack segment we need to convert this offset into Negative offset, and then perform bounds check and address translation.

Sharing Segments

It is possible to share the same physical segment between multiple processes. For example: it is possible for multiple processes to be using the same code segment in the physical memory.
To support sharing, the hardware will need to store additional information for each segment, i.e. what actions are permitted on that segment. This is done by using Protection bits, which are a few bits indicating whether or not a program can read or write to a segment, or whether it can execute the code within the segment.
The hardware checks will need to be modified as in addition to bounds check we also need to check if the specified operation is permitted on the segment or not.

Thus far we have only had 3 segments in the address space, however it is possible for the address space to have many segments (called fine-grained segmentation). To support many segments we need to use a Segment Table, and require additional hardware support.
When a context switch occurs the OS must save and restore the segment registers.

Paging

Paging is a non-contiguous memory allocation technique which allows the address space of the process to be stored in a non-contiguous manner in the physical memory.
Paging involves dividing the physical memory into fixed-sized blocks called frames, and dividing the address space of the process into blocks of the same size called pages.

To allocate memory for the process its pages are loaded into the available frames. Each frame can contain only one page.

Do note a frame can also be called as Physical Frame and Page can be called Virtual Memory Page.

Page Table
Page Table is a per-process data structure that stores which page is mapped to which frame.
=> The Page Table contains the base address of each frame in the physical memory.
=> The Page Table is stored in main memory, and a pointer to the page table is stored in the PCB (Process Control Block) of the respective process, as the page table is a per-process data structure.

=> *A Register called the Page Table Base Register (PTBR)* is present in the system which points to the current page table. At the time of context switching the OS only needs to change the value of this register to change the page table, i.e. switch to using the page table of the process it has decided to run.

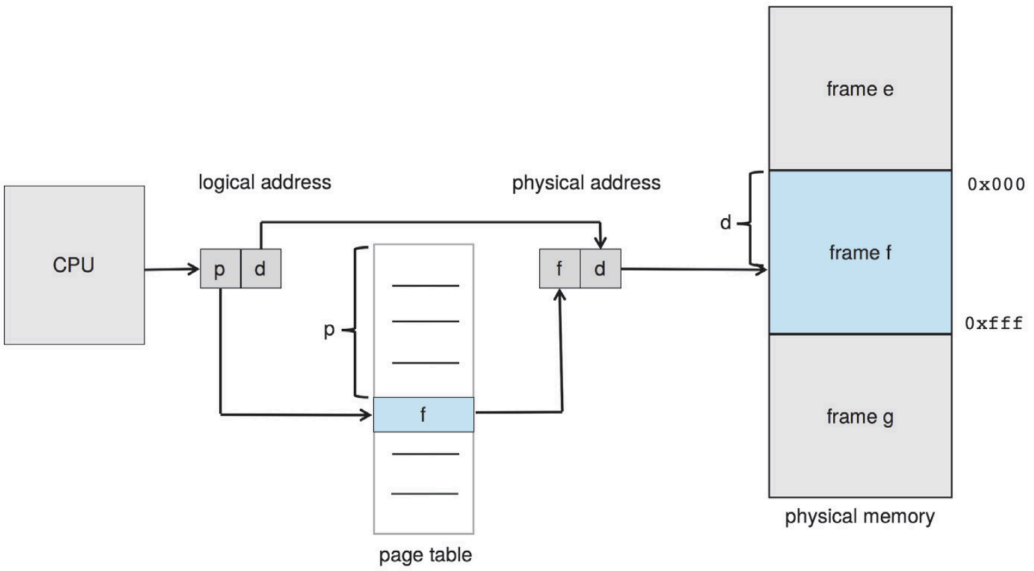
Address Translation in Paging

With Paging the virtual address (or the logical address, i.e. the one generated by the CPU) is divided into 2 parts:
page number (p) and page offset (d)

The page number indicates which page we are referring to, and the page offset indicates which byte of the page we are interested in.

To translate the Virtual Address to Physical address, follow below steps:

1. Extract Page Number (p), and use it as an index into the page Table.
2. Extract the corresponding frame number (f) from the page table.
3. Replace the page number (p) with the frame number (f) in the logical address, the address thus obtained is the physical address.



(Paging)

Page Size

The page size is defined by the hardware, and thus depends on the processor architecture. A common value for page size is 4KB (The page size is a power of 2).

~ % getconf PAGESIZE
4096

Page Size = Frame Size.

Notes on Fragmentation

Paging does **not** suffer from external fragmentation, this is because the physical memory is essentially an array of **fixed sized** slots, where each slot is a frame. A frame can be given to any process that needs one.

However Paging does suffer from internal fragmentation, in particular the last frame allocated to the process might not be completely full, i.e. have some space left, thus unused memory.

In general if the size of the address space is not divisible by the page size then we will have internal fragmentation.

For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 – 1,086 = 962 bytes.

In the worst case, a process contains n pages + 1 byte. It will be allocated n + 1 frames, and thus the last frame will be almost completely unused.

The Frame Table

Frame Table is a system wide data structure used by the OS to track the frames of physical memory, like how many frames are there in total, how many frames are allocated and how many frames are available etc. The frame table has one entry for each frame, indicating whether the frame is allocated or free, and if allocated then to which process.

Why is Paging Slow?

We discussed above that the page table is stored in main memory. But this is not the only option.

The Page Table can be implemented by using a set of dedicated high speed **hardware** registers (fast registers), this makes translation very efficient as the page table is stored in hardware registers. However this approach increases the context switch time, as each of the registers must be saved and restored during a context switch.

The use of fast registers to implement Page Table is appropriate only if the page table is small. Hence the approach taken is to keep the Page Table in the main memory, this allows for faster context switches.

Disadvantages of having the Page Table in the main memory

As mentioned above, even though having Page Table in main memory makes context switches faster, it increases memory access time as well. This is because with Paging, Address Translation requires 2 memory accesses, first we need to access the page Table to get the frame ‘f’ corresponding to the page ‘p’ (first memory access), next we combine f (frame number) with d (page offset) to get the physical Address. Now using this address we access the physical memory (second memory access). The 2 memory accesses required for address translation make Paging slower, a solution to this problem is to use a TLB (Translation Look-aside Buffer).

Paging with TLB (Translation Look-aside Buffer)
TLB is a special, small, fast-lookup hardware cache. TLB is associative, high speed memory. Each entry in TLB consists of 2 parts, a key and a value. When presented with an item, the TLB will search the keys to find the item, if found it’ll return the corresponding value.

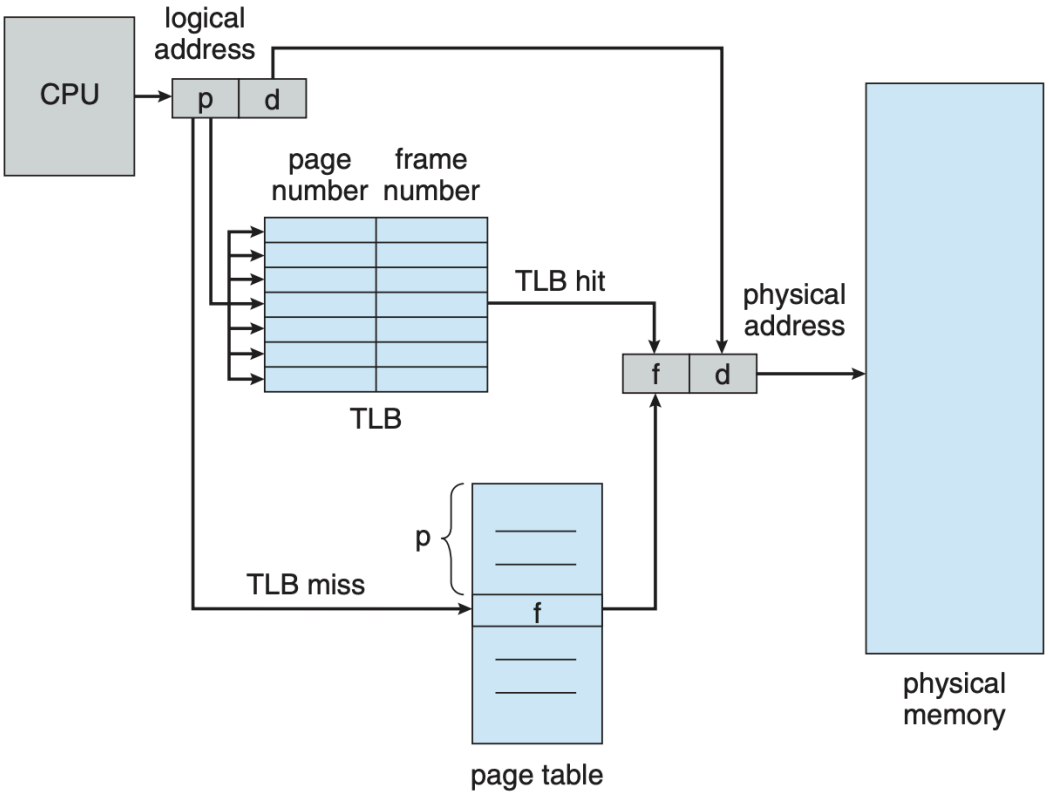
TLB is used for address translation in the following way:
Whenever a logical address needs to be translated to physical address, the MMU (Memory management unit), will first check if the page number is present in the TLB. The TLB will quickly search and return the corresponding frame number; this frame number can be combined with the page offset to get the physical address.

Point to note, searching with TLB is very quick, it is expensive hardware. (The reason is that the TLB lookups are part of the instruction pipeline). Ofcouse since it is hardware based, thus lookups are already much faster than performing lookups on main memory.

What happens if the page number is not present in the TLB. A point to note is that TLB is quite small, i.e. it cannot have a large number of entries, generally a TLB can store b/w 32 - 1024 entries.

Now if the specified page number is not present in the TLB, then a memory reference to the page table must be made to find the corresponding frame number. Additionally we add the page number - frame number mapping to the TLB (Cache Aside). So that if we query for this page number again in future, it'll be quickly found in the TLB.

If the specified key is present in the TLB, it's called a TLB Hit.
Else, if the key is not present in TLB, it's a TLB miss.



(Paging with TLB)

What happens if the TLB is full, and we try to add an entry?

In such a situation the TLB will need to evict one of the older entries to make room for the new one, which entry it chooses depends on the Cache Eviction algorithm, for example LRU or LFU or any other.
If we want, we can wire down some entries to the TLB, i.e. these entries will never be removed from the TLB.

Context Switching with TLB
How does the OS perform a context switch if we are using Paging with a TLB.

One approach is to flush out the TLB completely whenever we decide to switch to a different process, if we don't flush out the TLB then there might be older entries leftover by the previous processes, these entries are invalid with respect to the new process.

For example consider that process P0 leaves an entry mapping its page1 to the physical frame 5 on the TLB. Now when P1 comes in and tries to perform an address translation involving it's page number 1, then the MMU will look into the TLB and get a TLB hit and return the frame number 5, However the frame 5 is allocated to process P0 not to P1, hence P1 should not be able to access it, this is a security concern.

An alternative approach is for the TLB to store some additional info for each entry. This piece of information called Address-Space Identifier (ASID) uniquely identifies the process to which the frame is currently allocated to. This is used for protection purposes, so that no process can reference a frame allocated to any other process. With an ASID in place the TLB will perform lookups differently, instead of just searching for the page number, the TLB will now search for the pair {page number, ASID}.

Having an ASID enables the TLB to store entries for multiple processes at the same time. Hence at the time of context switch we don't need to flush out the TLB, we can use it as is, without any security or integrity concerns.

What is stored inside a Page Table Entry?
The page table is a data structure that stores which page number is mapped to which frame number. The Page Table however, can contain additional info as well. The page tables discussed thus far are called linear page tables, since they are essentially an array and the page number serves as the index.

In such a page table, the PTE (page table entry) contains just the frame number.
Here we discuss some other important information stored in a page table entry.

1. Valid Bit: The valid bit is used to indicate whether a particular translation is valid or not.
   For example a process contains the code and heap segments on one end and the stack on another, there might be some unused space in between. In paging since we divide the address space into fixed sized blocks (pages), hence there will be some unused pages (Since there is some empty space, it indicates the process is not using up the entire address space, and the pages which fall into this region of unused space are the unused pages). Obviously we should not be allocating physical frames for these pages, since that would be a waste of physical memory. To do this we mark all the unused pages as invalid (i.e. set the valid bit to 0 in the page table entries for these pages). Hence if a program tries to access

the frame corresponding to such a page, an exception will be raised which will trap into the OS.

2. Protection Bits: The protection bits indicate what actions are permitted on a page, for example read-write, read-only, execute etc. If a program tries to perform some other action on the page then a trap to the OS will be generated.

3. Dirty Bit: Used to indicate if the page has been modified.

4. Present Bit: The present bit indicates whether this page is present in physical memory or on the disk. In all the examples discussed so far the pages were always present in the main memory inside the frames. However it is possible for the pages to not be present in the main memory.

Virtual Memory

Introduction

There could be processes with address space larger than the entire Physical memory, how can we load a process into the main memory?
In a multiprogramming environment we load multiple processes in memory which are ready to run at any point, however there might not be enough space in physical memory to fit the address space of every process.
Thus in such scenarios it might not be possible to completely load a process into main memory, then how could it be executed?

To solve this problem we use the technique of Virtual Memory.

Virtual Memory is a technique that allows for the execution of processes that are not completely in main memory, it does so by treating a part of the secondary storage (disk) as main memory.

The major advantage of Virtual Memory is that programs larger than the physical memory can be executed.

Essentially Virtual Memory helps in creating the illusion of a large virtual address space.

A Debrief:

The instructions must be in physical memory to get executed. This is a necessary requirement. The first approach to meet this requirement is to load the entire Address space of the process in physical memory (be it in a contiguous or non-contiguous manner). However, this approach

limits the size of the program to the size of the physical memory, because a program larger than physical memory cannot be loaded into the physical memory, hence such a program will never be run.

However there are often situations when not the entire program is needed. For example, a program might contain some error handlers for unusual errors, which will almost never occur in practice. Thus this code is almost never executed.
Certain Options and Features provided by a program might be very rarely used.
In the worst case, A Program could contain some dead code, i.e. code which will never get run but is included as part of the program.

Thus in such situations the entire program might not be needed, even in cases where the entire program is needed it might not all be needed at the same time.

Considering the above factors if we can execute a program that is only partially in memory then that would be beneficial.

Benefits of Virtual Memory:

1. Virtual memory makes it possible for programs larger than physical memory to get executed, i.e. the program size is no longer limited to the physical memory size.
2. If instead of loading the entire process to memory, we load only some parts of it as and when needed, then physical memory will be saved which can be used to accommodate other processes, hence more processes can be fitted into the main memory, increasing the CPU utilization and degree of multiprogramming.

Demand Paging

Demand Paging is a popular method of Virtual Memory Management. With normal Paging, all the pages of the process need to be loaded to the main memory.
With demand paging, pages are loaded only when they are needed (on-demand), thus if a page is never accessed, it'll never be loaded to the physical memory.

Swap Space
The OS needs to reserve some space on the disk, for moving pages in and out (swap in, swap out). This space is known as swap space. The OS can read and write to the swap space.
Additionally the OS needs to remember the disk address of each swapped out page.

Present Bit
How do we know if a page is present in the physical memory or on the disk?

We use the preset bit for this purpose, if the present bit is 1 it means that the page is in the physical memory, however if it's 0 then the page is somewhere on the disk.
The present bit is stored in the Page Table Entry (PTE) corresponding to the page number.

Where does the present bit fit into the flow?
When a program performs a memory reference, the hardware will translate this virtual address to physical address. To do this it'll have to find the physical frame number this page number is mapped to. Now the following conditions are possible.

TLB contains an entry for the page number

then:

The corresponding frame number is retrieved and it is combined with the page offset to get the physical address.

Entry not found in TLB

then:

Perform a lookup in the page table, and examine the PTE corresponding to this page number.
If valid = 1, and present = 1, then return the frame number
If valid = 0, raise an Exception (Illegal Memory Access)
If valid = 1 and present = 0, the translation is valid but the page is in secondary storage, the hardware will raise a Page Fault exception which will trap into the OS, and the OS will load the page into the memory (swap in). Further it'll update the corresponding PTE, setting the present bit to 1 as well as updating the PFN (physical frame number) field.

Page Fault

A Page Fault is generated when the hardware tries to access a page which is not in memory. When a Page fault occurs, the hardware traps into the OS and the Page Fault Handler inside the OS runs.

The Page Fault handler like the other exception handlers is a piece of code inside the OS. When a Page Fault occurs, the hardware will raise a Page Fault exception and this handler will get triggered.

What will the handler do?

1. Since the page is present on the disk (i.e. swapped out), thus the OS will need to swap in the page. However, the OS needs to know the address of the location on the Disk where the page is stored. How can the OS remember this address? A common

approach is to save this information in the PTE corresponding to the page number when it is being swapped out. Thus the OS can simply lookup the Page Table Entry, and find the address.
2. Once the address is determined, the OS will issue an I/O request to the disk to fetch the page into memory.
3. Once disk I/O completes, and the page is in physical memory the OS will update the PTE of the page, setting the present bit to 1, and updating the PFN field.
4. When the above steps are complete, the hardware will retry the instruction.

Point of consideration: While the disk I/O is in flight, the process which originally made the memory reference will be in the blocked state, since disk I/O is an expensive operation (time consuming), the OS will schedule another process while the first process is blocked. This overlap of executing one process while waiting for another to complete I/O improves CPU utilization.

Page Eviction

What happens if the OS needs to swap in a page from the disk, but the memory is already full, i.e there are no free (unallocated) frames available, inside which the page can be stored. In such a scenario when there are no free frames available (or physical memory is full), the OS would need to take action and evict one of the existing pages from the main memory. Which page the OS will choose depends on the Page Replacement Policy being used (for example LRU, LFU etc.).
Once the page has been evicted, there will be space in memory to accommodate the incoming page. Specifically there would be a free frame available, which can accommodate the incoming page.

Address Translation and Physical memory access in Paging (control flow)

The next 2 code blocks essentially represent the entire control flow, associated with address translation. This includes performing a TLB lookup, making sure the action is permitted (by checking the protection bits), ensuring the translation is valid (valid bit) and finally swapping in a page if it's not in physical memory (i.e. present bit is 0).

```
p, d = extract_page_number_and_offset()

tlb_entry = tlb.search(p)
if (tlb_entry != null) {
      if (can_access(tlb_entry.protection_bits) == true) {
            f = tlb_entry.f
            physical_addr = compute_physical_address(f, d)
            register = access_memory(physical_addr)
      } else {
            raise Exception(PROTECTION_FAULT)
      }
} else {
      // Entry not found in TLB, lookup in the page table
      pte = PTBR[p] // Memory Access
      if (pte.valid == false) {
            raise Exception("Illegal Memory Access")
      } else {
            // Translation is valid
            if (can_access(pte.protection_bits) == false) {
                  raise Exception(PROTECTION_FAULT)
            } else {
                  if(pte.present_bit == 1) {
                        f = pte.f
                        physical_addr = compute_physical_address(f, d)
                        tlb.insert({p, {f, pte.protection_bits}})
                        register = access_memory(physical_addr)
                  } else {
                        raise Exception(PAGE_FAULT)
                  }
            }
      }
}
```

The next diagram shows the control flow of the OS page fault handler, which is triggered when the hardware raises the PAGE_FAULT exception.

```
pte = PTBR[p]

free_frame = find_free_frame()
if(free_frame == -1) {
      // no free frame, evict an existing page from memory
      free_frame = evict_page_from_memory()
}

buf = read_page_from_disk(pte.disk_address)
assign_page_to_frame(buf, free_frame)

pte.f = free_frame
pte.present = 1

RetryInstruction()
```

The 2 code flows combined give the complete control flow of memory access.

The first code flow shows what the hardware does during address translation (Hardware control flow), and the second one shows what the OS does when a page fault occurs (Software Control Flow).

Pure Demand Paging

Pure Demand Paging is a type of demand paging where we start the execution of the process without loading any of its pages to memory. When the OS sets the Program Counter (Instruction Pointer) to the first instruction of the process, a page fault will occur, and OS will load the page into memory, after the page is brought in the process will continue to execute, generating page faults as necessary until all the pages it

needs for execution are in the memory. After this point it won't generate any more page faults.

Page Replacement Algorithms

When a page fault is generated the page fault handler in the OS runs, which will swap in the needed page from the disk. However if the physical memory is already full, i.e. no free frames available then the OS will need to evict (or remove or deallocate) one of the existing pages from memory. How is this page chosen? That's decided by Page Replacement Algorithms.

FIFO (First In First Out) Page Replacement Algorithm - The FIFO algorithm records for each page, the time when the page was brought into memory. When a page has to be replaced the oldest page is chosen (i.e. the one which was brought into the memory earliest). Do note it is not mandatory to record the time when the page was brought into the memory, instead we can create a FIFO queue to hold all the pages in the memory. When we need to replace a page, then the page at the head of the queue is chosen and when a page is brought into the memory it is inserted at the tail of the queue.

The Performance of the FIFO Page Replacement algorithm is not always good. For example consider the 2 cases:

1. The page replaced may be part of an initialization module, which was used a long time ago and is no longer needed. (Good Replacement Candidate)
2. On the other hand the page could contain a heavily used variable that was initialized early and is in constant use. (Will cause Page Faults).

What happens in case of a page hit?
When the page is already in main memory then in the case of FIFO page replacement algorithm no change takes place, i.e. the queue retains the same FIFO order as it did before. Don't confuse it with LRU, where in the case of a page hit the page is moved to the top of the stack.

FIFO Page Replacement algorithm suffers from Belady's Anomaly, for some reference strings, page faults might increase as the number of allocated frames increases. For other algorithms page faults always decrease as the number of frames increases.

Optimal Page Replacement Algorithm - The idea behind the Optimal Page Replacement algorithm is that: replace the page that will not be used for the longest period of time in the future.

This algorithm produces the minimum number of page faults possible, and doesn't suffer from Belady's Anomaly.

However it is difficult to implement as the OS requires future knowledge of the reference string (i.e. which pages will be referenced in the future), which is basically impossible. (Similar to SJF).

Optimal Page Replacement Algorithm is used in comparison studies, serving as a benchmark.

Least Recently Used Page Replacement Algorithm (LRU) - Replace the page that has not been used for the longest period of time. LRU algorithm records for each page the time it was last used. When a page has to be evicted, LRU chooses the page that has not been used for the longest period of time.

Approaches to implement LRU

1. Track Time of Last used
    a. In this approach we add a new field to the Page Table entry tracking the time the page was last used.
    b. When we need to replace a page, we replace the one which has not been used for the longest period of time, i.e. the page with the smallest value for this time field.

2. Stack implementation
    a. Keep a stack of page numbers
    b. Whenever a page is referenced it is removed from the stack (if it exists) and put on the top.
    c. As a result the more recently accessed pages will be near the top of the stack.
    d. The least recently used page will be at the bottom of the stack.
    e. Since we need to remove pages from the middle of the stack, hence a better approach would be to use a doubly Linked List, with a head and tail pointer.

The LRU Page Replacement Algorithm is an approximation of the Optimal Page Replacement Algorithm, i.e we use the recent past as an approximation of the near future.

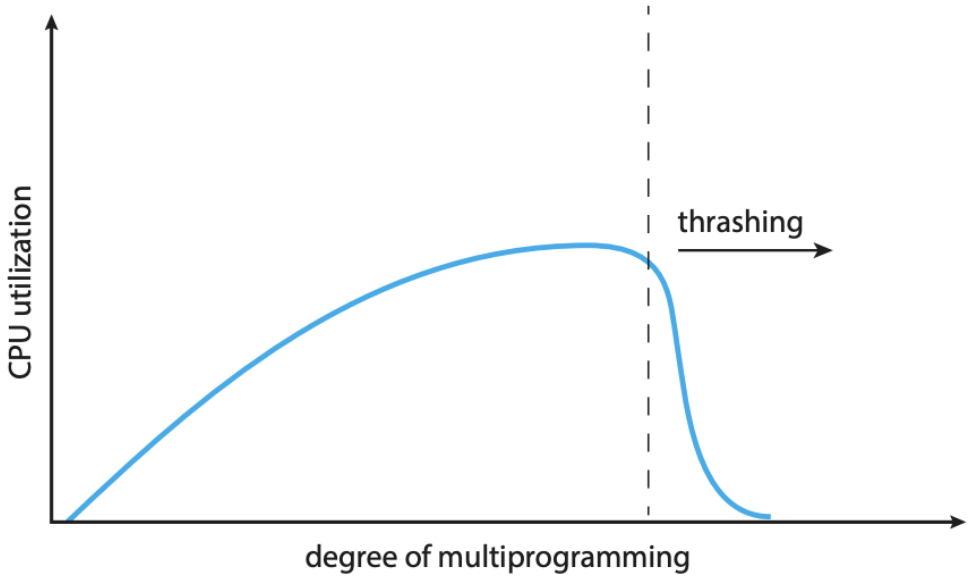
Other Page Replacement Algorithms: LFU, MFU.

Thrashing

When a process is running not all of its pages are loaded in the physical memory, as we saw before pages can be swapped into the memory from the backing store (disc). However if a process is allocated very few frames, or there is not much room left in the physical memory, then the process will quickly page fault. When the page fault occurs the OS will load in the page from the disk, however since the number of frames available is small, hence they will very quickly fill up, once that happens the OS will need to evict some pages from the memory, which might still be in active use (This is because the number of frames available is so small, a Page Replacement Algorithm like LRU ends up evicting a page which is still in active use, i.e. one which has been used very recently and one which might be needed in the very near future). Such a situation is known as Thrashing.

In the case of Thrashing the system spends an excessive amount of time swapping pages in and out rather than executing processes, hence CPU Utilisation is hampered. In other words the system is said to be thrashing when it spends more time servicing page faults than executing processes.

Thrashing hampers CPU Utilisation and degree of multiprogramming.



Here as can be seen in the graph the CPU utilization increases with degree of multiprogramming which is expected as there are more processes available in memory to run at any point in time. However after a certain point any increase in degree of multiprogramming actually decreases the CPU utilization as thrashing has ensued.

Techniques to handle Thrashing

1. Working Set Model - The Working Set model is based on the Locality model of process execution. According to the locality model as the process executes it moves from one locality to another locality. A process generally consists of several different localities. A locality is basically a set of pages that are actively being used together.

   For example consider the following sets of pages
   A = {1, 3, 5}, B = {2, 4, 6, 8, 10}

Here A and B are 2 different localities, remember each locality is defined by the set of pages that are actively being used together.

The basic principle behind the Working set model is that if we allocate enough frames for a process to accommodate its current locality, then it will only page fault when it moves to some other locality, however if the frames allocated are lesser than the size of the current locality then the process is bound to thrash.

To determine the current locality and henceforth it's size the Working set model uses a working-set window, which tracks the k most recent page references (all references not just unique ones), this window known simply as the working set serves as an approximation of the locality. The choice of k is critical.

2. Page Fault Frequency - In this strategy we aim to control and regulate page fault rate. In case of Thrashing the Page fault rate is very high.
   When the page fault rate is very high we know the process needs more frames. Conversely if the page fault rate is too low then we know the process has too many frames.
   We define an upper and lower bound for the page Fault Rate. If the actual page fault rate exceeds the upper limit, then we allocate an additional frame for the process. Instead, if the actual page fault rate falls below the lower limit, then we remove a frame from the process. Thus we can regulate the page Fault rate ensuring that it stays in the range [LB, UB], addition of frames will decrease the page fault rate, conversely if the page fault rate is below the LB then we can deallocate some frames, as the process has too many frames. The frames hence freed can be used by other processes.

Using this strategy we can prevent thrashing, by imposing a strict upper bound on the page fault rate.

Memory Layout of a C Program

In C, during the course of program execution the memory is organized into the following sections:

1. Text Segment
2. Data Segment
3. BSS
4. Heap
5. Stack

Text Segment:
The Text segment stores the executable instructions, i.e. the code of the program.

Data Segment
The Data segment is also called the initialized data segment. This segment stores the global variables and static variables which have been initialized by the programmeer.

Ex, static int x = 10
global int y = 25

Will be stored in the data segment.

BSS (Block Started by Symbol)
The BSS, is also called the uninitialized data segment. This segment stores the static variables and global variables which have either not been initialized by the programmer or have been initialized to zero. Data in this segment is initialized to 0 by default, before the program starts executing.

Ex static int x;
global int y;

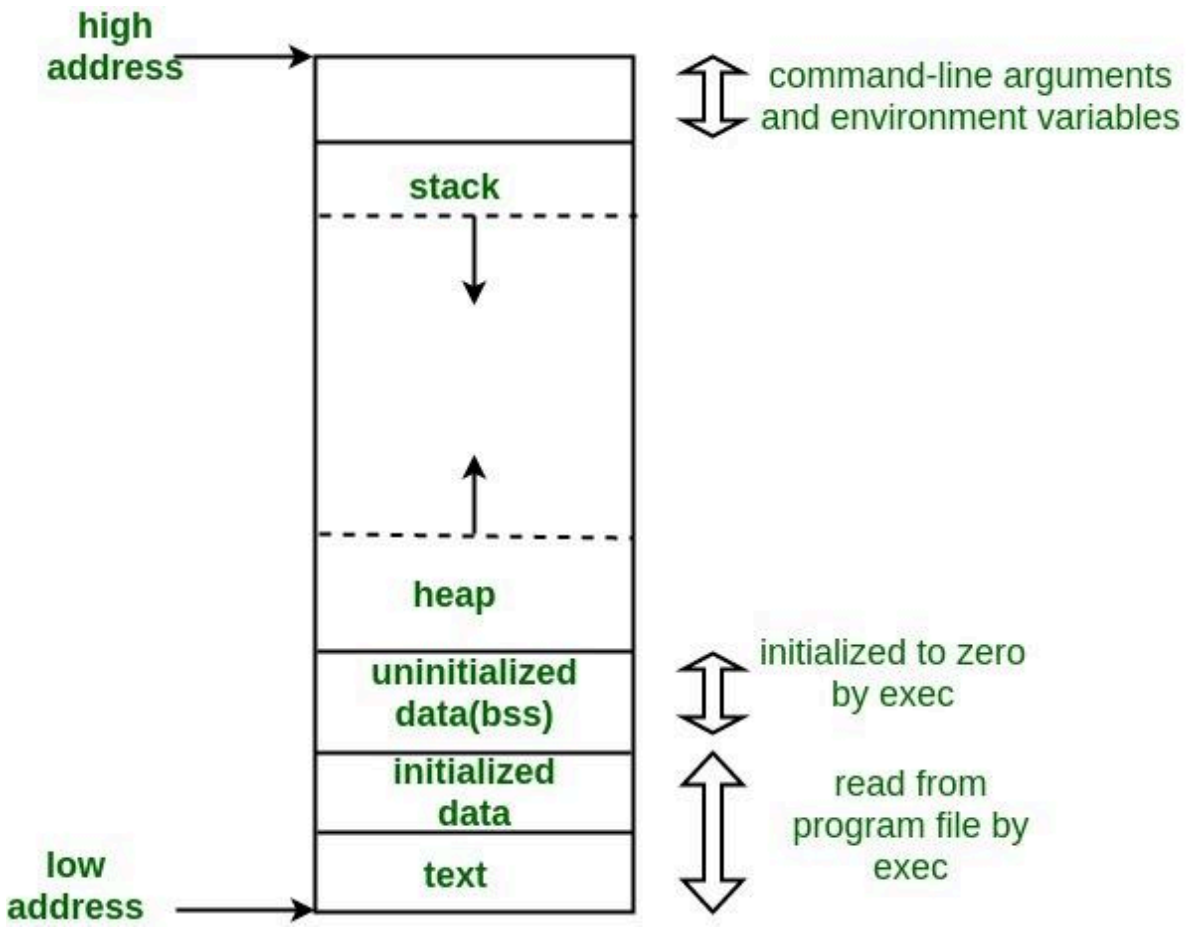
Will be stored in the BSS segment.

Stack

Stack area stores the program stack, which is a LIFO structure. The program stack keeps track of function execution. Every time a function is called, a stack frame corresponding to it will be pushed to the stack. The stack frame will at the minimum contain the return address (ie address of the caller). In addition, the stack also stores automatic variables. One stack frame doesn't interfere with the other.
Do note stack memory is also called an automatic storage class.

Heap

Heap is the segment where dynamic memory allocation takes place.
Heap is managed by calls to malloc / realloc / free.



To see the memory layout of a C program
$ gcc memory-layout.c -o memory-layout
$ size memory-layout

Important Note about heap / stack memory allocation

When we allocate something dynamically using malloc, there are actually TWO pieces of data being stored. The dynamic memory is allocated on the heap, and the pointer itself is allocated on the stack. So in this code:
int* j = malloc(sizeof(int));
This is allocating space on the heap for an integer. It's also allocating space on the stack for a pointer (j). The variable j's value is the address returned by malloc.

The pointer variable itself would reside on the stack. The memory that the pointer points to would reside on the heap.

Pointer variable itself is stored on the stack, it's the memory which is dynamically allocated via malloc, calloc that is actually stored on the heap.

Struct Memory allocation

structs can be allocated memory both on stack as well as heap.
To allocate memory for the struct on stack, declare it as a normal / non-pointer value

Q. Why is the process created via fork, called an almost exact copy of the parent process?
The child process created via fork, copies the memory of the parent process, getting a copy of the parent's data segments, heap and stack. In addition, the child process uses the same CPU registers, Program Counter, and the same open files as the parent process. In simplified terms: The child process receives a copy of the parent's virtual memory space, including the program counter, registers, and open files. This means the child process starts executing from the same point where the parent left off.
However after the child process is created, both the child and parent process will run in their own separate memory spaces, thus not affecting each other.

Concurrency

Index:

Threads

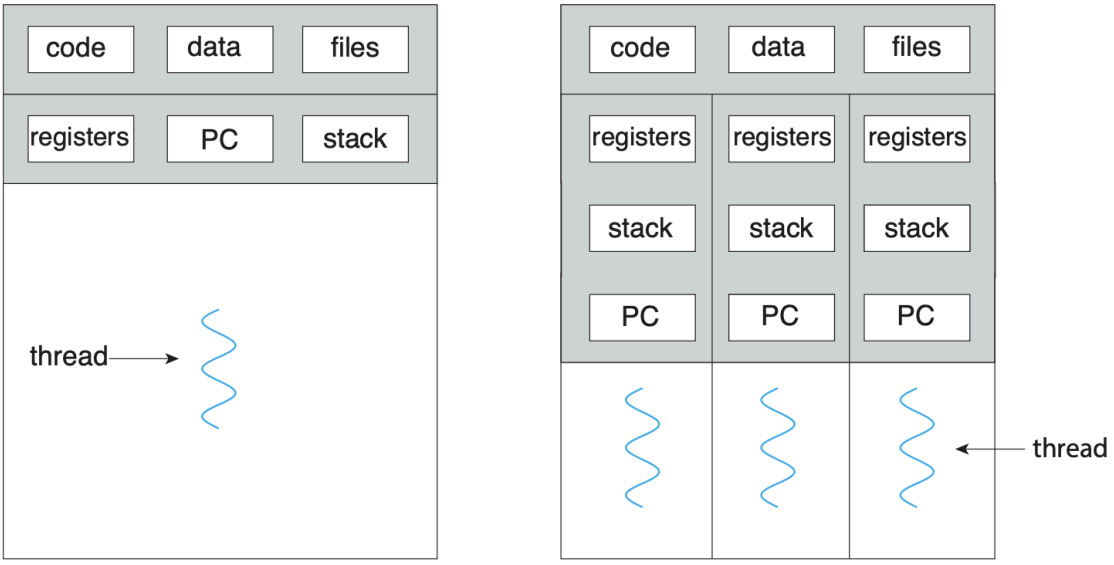
A multithreaded program has multiple points of execution, i.e. multiple Program Counters (PC). In contrast a single-threaded program has a single point of execution, i.e. single PC. A thread is basically a single sequence stream within the process.

Each Thread is very much like a separate process, except for one difference, The threads share the same address space and thus can access the same data.

Each thread has its own private set of registers, including Program Counter (which tracks where the program is fetching instructions from).

If we have two threads running on a single processor and we want to switch from running T1 to T2, then we need to perform a context switch. When the context switch happens the register state of T1 is saved and the state of T2 needs to be restored. Similar to PCB we have a TCB (Thread Control Block) to store the state of each thread of a process. However since the threads share the same address space, hence when we are performing a context switch we don't need to change the address space (by switching the page table for example), hence context switching is faster with threads.

Each thread has its own stack segment, and will store its stack-allocated variables, function parameters and return addresses in its own stack (called thread-local storage).



single-threaded process

multithreaded process

As can be seen above the threads share the same Code and data segment. Also in addition they share the same heap segment as well (not mentioned above), along with other resources like open files. On the other hand each thread has its own stack segment (thread-local storage), PC and its own private registers. Since each thread has its own stack memory hence each thread will have its own private stack pointer as well (which points to the top of the stack), hence stack pointer is a per-thread structure (Btw, stack pointer is a register so it is already covered when we say that each thread has its own private registers).

Thread Use Cases

1. Parallelism: Imagine we have a program which operates on very large arrays, for example incrementing each element in the array by x. If we are performing this operation on a single-processor system, then we need to perform the operations as is. However if we have multiple processors we can speed up the program considerably, by performing a portion of work on each processor. The task of transforming a single threaded program into a

program that utilizes multiple CPUs for work, is called Parallelization, this is achieved by using a thread per CPU. Parallelization allows programs to run much faster on modern hardware.

2. Avoid Blocking Program Progress due to slow I/O: When a process issues an I/O it needs to wait for the I/O to complete, however instead of waiting the program could do some other useful work in the time being, this can be achieved by having multiple threads. While one thread is blocked waiting for I/O, the CPU scheduler can switch to another thread. Thus threading enables the overlap of I/O with other activities within a single program.

Threading is used by applications like Database management Systems, Web Servers etc.

When we create a new thread it may start running right away, alternatively it may be put in a “ready” state (The Scheduler will transition it to “running” later).

Accessing Shared Data - Threads share the same address space, In addition to thread specific data like PC, private registers and Stack there is also common global shared data which can be accessed by all the threads. This could include variables, files etc.

In a single-threaded program since there is only one thread, hence it can freely access whatever data it wants, however in a multi-threaded program where multiple threads are accessing the shared data we can face problems with concurrent access, i.e. if multiple threads try to

access some shared resources simultaneously, it could lead to data inconsistencies, as we will see in the next example.

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>

static volatile int counter = 0;

void * thread_routine(void * arg) {
     printf("%s\n", (char *) arg);

     for(int i = 0; i < 1e7; i++) {
          counter++;
     }

     return NULL;
}

int main(int argc, char const *argv[])
{
     pthread_t t1, t2;
     printf("main:begin counter = %d\n", counter);

     pthread_create(&t1, NULL, thread_routine, "A");
     pthread_create(&t2, NULL, thread_routine, "B");

     pthread_join(t1, NULL);
     pthread_join(t2, NULL);

     printf("main:end counter = %d\n", counter);
```

```
    return 0;
}
```

(Code - A)

In this code 2 threads access the shared variable counter, The start Routine of each thread is the function ‘thread_routine’, which adds 1 to the counter 10 million times (1e7), since both threads will execute this piece of code we expect the output to be (1 x 10 mil) x 2 = 20 million, i.e. 2e7.

However the actual output produced is very different, refer 2 runs of this program.

First Run
% gcc threading.c
% ./a.out
main:begin counter = 0
A
B
main:end counter = 10423395

Another run
% gcc threading.c
% ./a.out
main:begin counter = 0
A
B
main:end counter = 11069214

To understand why this is happening refer to section 26.4 of the book.

The output of this program is not deterministic, i.e. we cannot say in advance what output it will produce. The output depends on timing and on how the threads are scheduled by the Scheduler.

Definitions

Critical Section - The critical section is a piece of code that accesses the shared resources (for example global variables, or common files). The critical section code must not be executed concurrently by multiple threads, instead only one thread at a time should execute the code in the critical section.

Race Condition - A Race Condition is said to occur when multiple threads execute within the critical section at the same time, a race condition can lead to the shared resources being updated incorrectly, additionally it makes the result non-deterministic, i.e. we do not know what the output will be, and it might be different across runs.

Mutual Exclusion - The property of Mutual Exclusion guarantees that if one thread is executing within the critical section, then the other threads will be prevented from entering the critical section.

Solutions to the Critical Section Problem

Locks

We can use locking mechanisms to solve the Critical Section Problem. To use locks we first need to define a lock variable which will track the state of the lock at any instant in time. The state of the lock could be either available (or unlocked or free), or acquired (locked or held). To enter the critical section, a thread must acquire the lock.

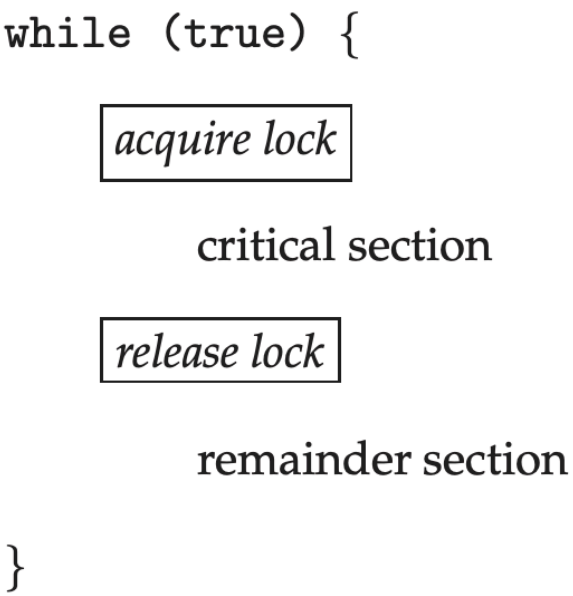
Lock States
When the lock is in the available state it means none of the threads have acquired the lock and hence none of the threads are in the critical section.
When the lock is in the acquired state, it means exactly one thread holds the lock, and is executing in the critical section.

Control Flow:
If a thread wishes to enter the critical section it will try to acquire the lock by calling the acquire() routine.
If no other thread holds the lock (i.e. it is free) then the requesting thread will acquire the lock and enter its critical section. While it is in the critical section, the lock will be in the acquired state and hence no other thread will be able to acquire it. Once the thread in the critical section has done its work it'll call the release() routine to free the lock, hence the state of the lock will change to available again and other threads will be able to acquire it.

However if some other thread currently holds the lock, then the call to lock() routine will not return, As a result if a thread attempts to acquire an unavailable lock it will be blocked until the lock is released. Hence if

a thread currently has the lock then no other threads will be allowed to acquire it and hence can't enter the critical section.

Hence the flow will be as follows:
Acquire Lock
Critical Section
Release Lock



Pseudo Code for acquire() and release()

```
acquire() {
     while(!available) {
          // busy wait
     }
     available = false
}

release() {
     available = true
}
```

Calls to acquire() and release() must be performed atomically.

Thus using locks is a solution to the critical section problem. With a locking mechanism in place only one thread can enter the critical section at a time. It helps to achieve mutual exclusion i.e. if a thread is executing in the critical section then the other threads will be prevented from entering the critical section.

Disadvantages of Locks

The disadvantage with this implementation of locks is busy waiting, i.e. if a thread tries to acquire a lock which is not available, then the thread will loop continuously in the call to acquire(), this is a waste of CPU cycles that some other process might have used productively. Thus Busy waiting leads to a waste of resources.

Pthread Locks

POSIX Library uses the word mutex for locks (as it is used to provide mutual exclusion).

Create lock variable
pthread_mutex_t lock;

Acquire Lock
pthread_mutex_lock(&lock)
// Lock acquired enter critical section
// Critical Section
Release Lock
pthread_mutex_unlock(&lock)

We can use pthread locks to fix the race condition in (Code - A)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>

static volatile int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void * thread_routine(void * arg) {
    printf("%s\n", (char *) arg);

    pthread_mutex_lock(&lock);
    for(int i = 0; i < 1e7; i++) {
        counter++;
    }
```

```
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t t1, t2;
    printf("main:begin counter = %d\n", counter);

    pthread_create(&t1, NULL, thread_routine, "A");
    pthread_create(&t2, NULL, thread_routine, "B");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("main:end counter = %d\n", counter);
    return 0;
}
```

Output:

% gcc test.c

% ./a.out

main:begin counter = 0

A

B

main:end counter = 20000000

Evaluating locks:

Locks are evaluated against the following parameters:

1. Mutual Exclusion: This is the basic job of the lock to ensure no other thread can enter the critical section if a thread is executing within it.
2. Fairness - Each Thread contending for the lock should get a fair shot at acquiring the lock once it is free.
3. Bounded Waiting - No thread should starve waiting to acquire the lock, there exists a limit on the number of times other threads are allowed to enter the critical section after a thread has made a request to enter the critical section.
4. Performance

Software based Solutions

Approach 1: Using loads and stores, Can we use a single flag variable to solve the problem of race condition?

In this approach we use a variable flag to indicate if a thread is executing in the critical section.
i.e. flag = 1, A thread is executing in the critical section
flag = 0, the critical section is not being executed by any of the threads.

When a thread wants to execute the code in the critical section, it will check the value of the flag. If flag == 0, then no thread is currently executing the critical section code, hence the requesting thread can enter the critical section, it does so by changing the flag to 1, and enters the critical section. When it has completed its work, the thread will clear the flag (setting it back to 0) so that other threads can enter the critical section.

However if a thread tries to enter the critical section, when a thread is already running in it (i.e. flag == 1) then the thread will spin wait (busy wait), i.e. it'll loop continuously until the critical section is empty.

However, this approach is not valid, as it does not solve the problem of Mutual Exclusion, this is due to a lack of atomicity in the operations.

Suppose we have 2 threads A and B, both wishing to enter the critical section.
Assume the following order of execution

Thread: A, flag = 0
while(flag == 1)

Context Switch to B

Thread B, flag = 0
while(flag == 1)
flag = 1
Context Switch

Thread A, flag = 1
Continue from PC: flag = 1

Thus both threads A and B set the value of the flag as 1, and will both enter the critical section. Hence this is not a valid approach.

Approach 2: Enhanced single flag variable approach.

Instead of a boolean flag we can have an integer flag: turn.
The variable turn indicates whose turn it is to enter the critical section.

Assume we have 2 threads T0, T1, initially turn is set to 0.

```
turn = 0 // Shared flag initialization
```

Execution Logic

```
while(turn != i) {
      // some other thread in the critical section, busy wait.
}
// Enter Critical Section
// Done execution in Critical Section
turn = 1 - i (allowing other thread to run)
```

However there are problems with this approach related to Progress and Starvation. Since turn is initially set to 0. Hence thread 0 will always be

the first thread to enter the Critical Section, however what happens if thread 1 wants to enter the critical section before thread 0 does?
In such a scenario thread 1 will need to wait, and loop continuously in the spin lock, until thread 0 executes the critical section and sets the turn variable to 1 in the remainder section.

Hence this violates the principle of Progress, as even though the critical section is empty we cannot let thread 1 access the critical section. It's also possible for whatever reason that thread 0 never executes the critical section code, in this case thread 1 will keep waiting to enter the critical section and will eventually starve.

Approach 3: Peterson's Algorithm

Peterson's algorithm is restricted to 2 threads, T[0] and T[1], we define 2 variables:

bool flag[2], flag[i] == true, indicates that T[i] is ready to enter the critical section.

int turn, The variable turn indicates whose turn it is to enter the critical section, for example turn == i, indicates it is T[i]'s turn to enter the critical section.

How does Peterson's Algorithm work:

To enter the critical section, Thread Ti first sets flag[i] to be true and then sets turn to the value 1 - i, thereby asserting that if the other thread wishes to enter the critical section, it can do so.

If both threads try to enter the critical section at the same time, turn will be set to both i and j at roughly the same time. However, Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two threads is allowed to enter the critical section first.

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Where j = 1 - i

Hardware based Solutions

We can make use of some atomic instructions provided by the processor to achieve concurrency control.

Test and Set Instruction (Atomic Exchange):
=> We pass a memory address and a new value to the Test and Set Instruction.
=> It returns the existing value located at the provided memory address.
=> Simultaneously, it updates the value at the memory address to the new value.

```
int TestAndSet(int *ptr, int new_value) {
    int old = *ptr;
    *ptr = new_value;
    return old;
}
```

Obviously these group of instructions are executed atomically.
How can it be used as a lock:

```
void lock() {
        while(TestAndSet(&lock->status, 1) == 1) {
                // spin wait
        }
}

void unlock() {
        lock->status = 0;
}
```

Initially the lock variable is 0, when a thread tries to acquire the lock it'll call the TestAndSet instruction updating the lock to 1, and getting the old value at that memory location as the return value (0). Since Return value is not 1, it is allowed to enter the critical section.

If another thread tries to acquire lock now, it'll get a return value from TestAndSet of 1, hence it'll need to spin wait.

Compare And Swap Instruction:

Compare And Swap like Test And Test is a hardware primitive, we pass a memory address to the compare and swap instruction, along with two values an expected value and a new value.

The CompareAndSwap instruction compares the value at the specified address with the expected value, if they are the same it goes ahead and updates the value at the specified address to the new value, if however they differ then CompareAndSwap does nothing. In either case it returns the original value at the location.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if(original == expected) {
        *ptr = new;
    }
    return original;
}
```

We can create a spin-lock from Compare And Swap in a similar manner as TestAndSet

```
void lock() {
        while(TestAndSet(&lock->status, 0, 1) == 1) {
                // spin wait
        }
}
```

Lock Free Synchronization

Using Hardware instructions like TestAndSet and CompareAndSwap we can build spin-locks like mentioned above. However we can also use these atomic hardware primitives (instructions) to achieve lock free concurrency.

For example suppose we have an integer, which we need to increment in a concurrent manner (i.e. multiple threads are trying to increment it simultaneously). To prevent race conditions one approach would be to make use of locks, i.e. acquire the lock, make the update, release the lock. However we can perform this update via lock free synchronization as well, for example by using CompareAndSwap.

```
void atomicIncrement(int *ptr, int delta) {
        do {
                int value = *ptr;
        } while(compareAndSwap(ptr, value, value + delta) == 0);
```

Where a return value of 1 indicates CompareAndSwap succeeded, while a return value of 0 indicates failure, if the CompareAndSwap couldn't be executed successfully then we keep trying until it does.

Evaluating Spin Locks

Parameter	Evaluation

Correctness	Spin Locks provide mutual exclusion, i.e. only one thread is allowed to enter the critical section at a time. Hence they are correct.
Fairness	Spin Locks do not guarantee fairness; a thread spinning, waiting to acquire the spin lock, may spin forever under contention.
Performance	Spin Locks wastes CPU cycles. If a thread executed within the critical section is preempted, now the threads which are chosen to run will not be able to acquire the lock (since it is already held by one of the threads), and hence will need to spin wait, as a result each of the threads will spin for the duration of the time slice before being preempted, resulting in a waste of CPU cycles.

Alternatives to spinning:
If the lock is in the acquired state, and a thread makes a request to acquire the lock then instead of putting the thread into spin waiting (or busy waiting) we can:

1. Put the thread to sleep
2. Instead of spinning, the thread can explicitly give up the CPU by using the yield system call. Yield will move the thread from running to ready state.

Condition Variables

In certain situations a thread needs to check if some condition is true, before it can continue executing. For example, a parent thread waiting for the child thread to complete before it can carry on its execution. In this case the condition it is checking would be “has the child thread completed its execution”.

How can we achieve this?
A straightforward solution could be to use a shared variable, say a variable called done.
done = 0 child thread hasn’t finished
done = 1, child thread has completed execution hence the parent can run.

The parent thread can wait for the child thread to complete by putting itself in a while loop until the ‘done’ variable changes to 1.

i.e.
while(done == 0) // busy waiting

When the child thread is done executing, it’ll update done to 1 and the parent thread will fall out of the while loop and continue execution.

While this approach could work the disadvantage is obvious - busy waiting, as the parent is continually looping in the while loop it’ll waste cpu cycles hence it is inefficient. A better strategy would be to put the waiting thread (i.e. the parent thread in this context) to sleep (move it to a blocked state), once the child thread has finished execution it can send some kind of pulse to wake up the parent thread. This is what we can achieve by using Condition Variables.

Condition Variables is a Thread Synchronization tool, and it helps in running multiple threads in a deterministic manner. For example in a general context if we have three threads T1, T2, T3, then we cannot say anything about their order of execution, however with Condition Variables we can force them to run in a particular order. To wait for a condition to become true, a thread can make use of Condition Variable.

The Condition variable provides 2 operations:

1. wait() - When a thread wants to wait for a condition to become true, it'll call the wait() routine. The wait routine will put the thread to sleep.
2. signal() - The signal() call is executed by a thread when it has changed some condition, the signal() call will wake up all the sleeping threads waiting on this condition. For example when the child thread is done executing it'll call the signal() routine to wake up the asleep parent thread.

The wait() call takes the condition variable as well as a mutex as input. It does multiple things, first of all it assumes that the lock is held by the thread which is calling wait(), i.e. the mutex is locked. Next, wait() will release this lock and put the calling thread to sleep, when the thread wakes up (in response to signal()) after some time it will reacquire the lock as part of the wait() call.

Basically wait() does 3 things

1. Release the lock, pthread_mutex_unlock
2. Put the thread to sleep until signal
3. Re-acquire the lock, pthread_mutex_lock

This complexity is needed to perform the sleep operation atomically.

Semaphores

Semaphore is a thread synchronization tool, which can be used in multithreaded programs for various purposes:

1. It can act as a mutex, i.e. locking mechanism
2. It can be used by a thread to wait for some condition or some thread.
3. Finally, it can be used to control access to resources with multiple but limited instances.

A semaphore at its core is an integer variable S, which apart from initialization can only be accessed by 2 operations - wait() and signal().

The semaphore is initialized to the number of instances of the resource available. A thread which wants to acquire an instance of the resource will call the **wait()** operation on the semaphore, and decrease the count.

=> If the resultant count is >= 0 then there are non-assigned instances of the resource available, hence the thread will acquire an instance.

=> However if the resultant count is < 0, then it means all the instances are in use and there are no instances, hence as a result the thread which made the request will need to wait for an instance to become available.

How will the thread wait ?

1. The thread can enter busy-waiting or spin-waiting where it'll spin continuously until an instance is available.

2. Or, it can be put to sleep, when an instance is available we can wake up the thread and bring it to action again, since an instance is available the thread can acquire it and go on with its work.

When a thread which was able to acquire an instance of the resource is done using it, it will release it via a call to the signal() operation. The signal() operation will increase the value of S by 1 (indicating an additional instance is available now). If we use the approach of busy waiting in wait(), then incrementing S alone will be enough, however if we want to use the other approach wherein the requesting thread is suspended, then we'll need some additional mechanisms to wake up the sleeping threads.

wait() and signal() operations must be executed atomically, i.e. if one thread is modifying the semaphore variable, then no other thread can simultaneously modify it. In addition in the case of wait() operation, the testing of the integer value S i.e. (S <= 0), should also be performed atomically. Do remember if an instruction is atomic then it means when a CPU interrupt occurs at that point either the instruction has executed completely or not executed at all, there is no intermediate state. In other words the instruction must be executed in one CPU cycle.

Structure of the semaphore:

The semaphore at its core is still an integer variable, however the basic implementation suffers from busy waiting and hence to resolve this we implement the Semaphore in a different manner.

Instead of an integer variable, we will define the semaphore as a struct, containing 2 attributes: value (int) and process_list (Linked List of PCBs).

```
struct semaphore {
     int value;
     struct pcb* list;
}
```

This implementation of semaphore allows us to redefine wait() and signal().

wait() - When a thread wants to acquire an instance of the resource it'll call the wait() operation and it will decrement the value of the semaphore, if the value is >= 0 it knows there are free instances of the resource available hence it'll acquire one. However if the value is < 0, then there are no free instances available so the thread will be put to sleep (in other words it'll be suspended, i.e. moved to the blocked state). The suspended thread will be put into a waiting queue associated with the semaphore (the linked list of tcb nodes). As the thread is blocked, the control is transferred to the OS and it'll schedule another thread.

How do we wake up a sleeping thread?
signal() - When a thread no longer requires a resource, it'll call signal(), the signal() operation will increment the value of semaphore by 1. Next, signal() will check the waiting queue associated with the semaphore, if it is non-empty, then signal() will pick one of the threads from the queue, remove it from the queue, and finally wake it up by using the wakeup() system call, this will transition the thread from the blocked state to ready state, and it'll be put in the ready queue.

Implementation of wait() operation

```
wait(semaphore* S) {
     S->value--;
     if(S->value < 0) {
          Add thread to S->list
          sleep();
     }
}
```

Implementation of signal() operation

```
signal(semaphore *S) {
     S->value++;
     if(S->value <= 0) {
          Remove thread T from S->list
          wakeup(T);
     }
}
```

Summary of semaphore implementation: Now the semaphore is defined as a struct containing 2 attributes - a value, which is the value of the semaphore and a list of threads waiting on that semaphore.
If a thread needs to wait on a semaphore, it is put to sleep (instead of busy waiting) and it is put in the list of threads waiting on that semaphore. When signal() operation is called, it'll remove one thread from the list of waiting threads and wake it up.

The semaphore struct will contain a pointer to the head of this linked list (consider it as a linked list of pcb's), if we want to ensure bounded waiting then we can implement this list strictly as a FIFO queue, and the semaphore variable can store pointers to both head and tail of the list.

If the semaphore variable's value is negative, then it means some threads are waiting on the semaphore, and the magnitude of this negative value indicates how many such threads are there.

Types of Semaphores
Binary Semaphores - The value of a binary semaphore can be either 0 or 1.
Counting Semaphores - The value of a counting semaphore, can range over an unrestricted domain.

Both types of semaphores serve different purposes, Binary Semaphores can be used to achieve mutual exclusion, as the Binary Semaphores behave similarly to mutex locks.

Counting semaphores are used to control access to a resource having multiple but limited instances. In this case the semaphore is initialized to the number of instances of the resource.

Additionally semaphores can also be used to solve the problem of Thread Ordering, or running the threads in a deterministic manner.

Deadlock

In a multiprogramming system we have multiple processes competing for a finite number of resources. For a process to execute it needs to acquire all the resources it needs, which involves the request - use - release cycle, i.e. the process makes a request for the resources, if it is allocated to the process then the process can continue execution else it enters a wait or blocked state waiting for the resource to become available.

Deadlock is said to occur when for example threadA is holding the lock LA and waiting to acquire lock LB, unfortunately threadB which holds LB is waiting for LA to be released.

Example:

ThreadA:
pthread_mutex_lock(&LA)
pthread_mutex_lock(&LB)

ThreadB
pthread_mutex_lock(&LB)
pthread_mutex_lock(&LA)

If the above code runs then a deadlock might be produced, not necessarily though as it depends on the CPU scheduler.

Conditions for Deadlock;
There are four conditions which must all hold for a deadlock to occur.

=> Mutual Exclusion: Threads have exclusive control over the resources they acquire, i.e. if a thread has acquired a resource then no other thread can acquire or use that resource. Example thread acquiring a lock.

=> Hold-and-Wait: Threads hold resources allocated to them while waiting to acquire additional resources. Example: a thread which holds lock1 waiting to acquire lock2.

=> No Preemption: Resources cannot be forcibly removed from the thread that has acquired them, acquired resources can only be released by the thread itself.

=> Circular Wait: There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain.

Methods for handling Deadlocks

a. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
b. Allow the system to enter a deadlocked state, detect it and recover.
c. Ignore the problem altogether and pretend that deadlocks never occur in the system, i.e. deadlock ignorance (this is based on the fact that deadlocks are very rare).

Deadlock Prevention

As mentioned before a deadlock can only occur if all four of the above conditions are met.
So in deadlock prevention we try to make sure that at least one of the conditions is not met.

1. Preventing Circular Wait: In the above example of threadA and threadB, we can easily avoid a deadlock by ensuring that both the threads request for locks in the same order, for example LA followed by LB, in such a scenario there can't be any deadlock. This is an example of total ordering, where LA will always be requested before LB, by enforcing a total ordering on the locking code we can guarantee that no circular wait arises hence no deadlock. Hence, we design the locking code in such a way that circular wait is never induced.
   In large codebases however a total ordering might not be possible as there are many locks, in such a scenario we can use a partial ordering. Both total ordering and partial ordering structure the lock acquisition so as to prevent a deadlock.

2. Preventing Hold and Wait: To prevent the hold and wait requirement, we can ensure that all the resources required by a thread are acquired at once, atomically, there are 2 protocols which can help in preventing hold and wait.
   a. Protocol A: Each thread needs to request and acquire all its resources before execution.
   b. Protocol B: A thread can request resources only when it has none.

3. Preventing No Preemption:
   a. If a process is holding some resources and requests additional resources that cannot be immediately allocated to it, then all the resources that the process is currently holding are preempted. The process will restart only when it has regained its old resources as well as the new ones it requested.

b. If a process requests some resources, we first check if they are available, if they are then we allocate them to the process, if not then we check if these resources are allocated to some other process that is waiting for additional resources, if so, preempt the desired resources from that process and allocate them to the requesting process.

4. Mutual Exclusion: We cannot prevent mutual exclusion as this is a necessary mechanism because some resources are fundamentally non-sharable for example locks, hence we cannot prevent deadlocks by denying the mutual exclusion condition, however Sharable resources like Read-only files can be accessed by multiple threads simultaneously.

Resource Allocation Graph (RAG)
The RAG depicts the relations between the processes and the resources they have requested or acquired.

P -> R, indicates P has requested to acquire R
P <- R, indicates R has been allocated to P

If the Resources allocation graph has no cycle, then there is no deadlock in the system, if there is a cycle then there may be a deadlock.

Deadlock Avoidance
In case of deadlock avoidance the system has some global knowledge of the resources various threads will need during their execution, using this information the threads are scheduled in such a way that no deadlock occurs.

Some terminologies

=> Safe State: A system is in each safe state if it can fulfill the resource requests of each process, in some order and still avoid deadlock. A system is in safe state only if there exists a safe sequence.
=> Safe Sequence: The threads are scheduled in a way that no deadlock occurs, this ordering of threads (for example T1, T3, T5, T4, T2) is known as a safe sequence.
=> Unsafe State: The system cannot guarantee that it'll remain deadlock free if it fulfills the current set of resource requests by the processes. An unsafe state **may** lead to a deadlock.
=> The key idea behind deadlock avoidance is that a request for resources should be fulfilled only if it leaves the system in a safe state.
=> To find the safe sequence the Dijkstra's Banker's algorithm is used.

Deadlock Detection
Case - I: Single instances of each Resources - Convert the Resource Allocation Graph (RAG) to the wait-for graph. If the wait-for graph contains a cycle then there is a deadlock else not. To convert RAG to wait-for graph collapse all the resource nodes.

Case - II: Multiple instances of resources: Banker's Algorithm (just check if a safe sequence exists, if it does not then a deadlock exists).

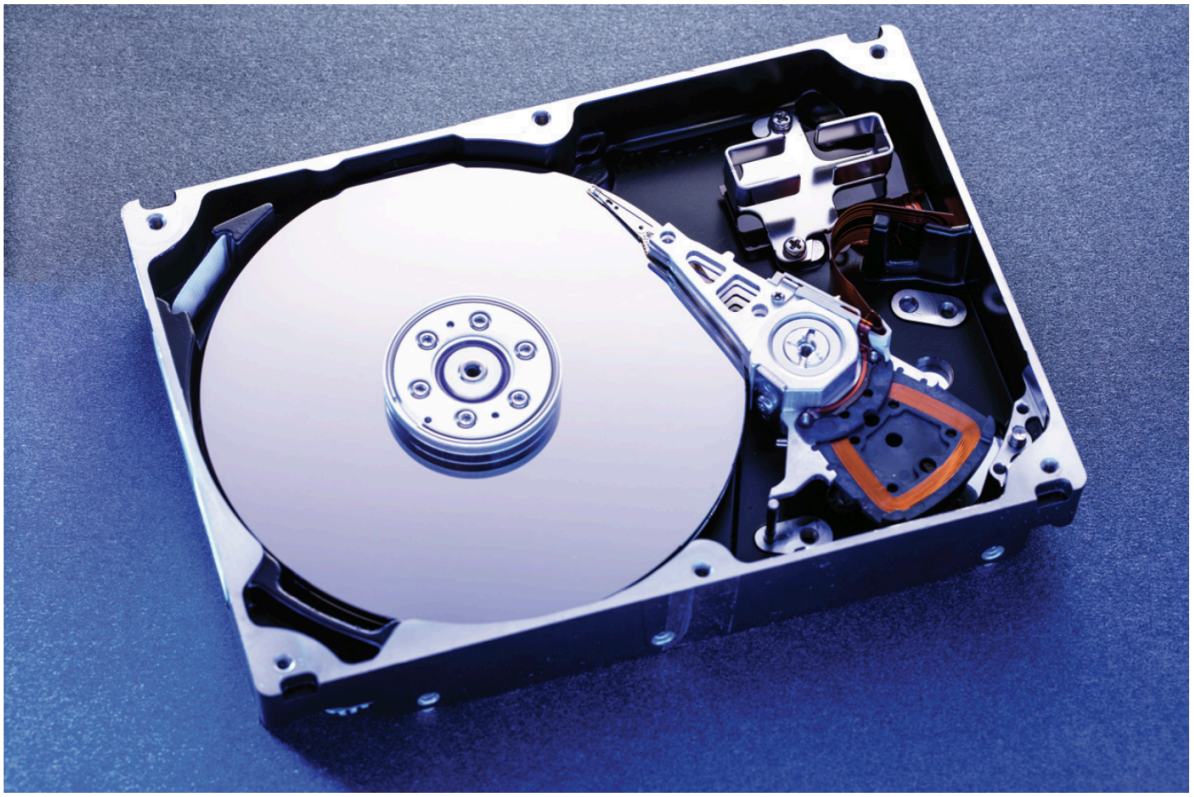
Recovery from deadlock

1. Process Termination
    a. Abort all the deadlocked processes.
    b. Abort one process at a time until the deadlock cycle is eliminated.

2. Resource Preemption
    To eliminate the deadlock we successively preempt resources from some of the processes and give these

resources to other processes until the deadlock cycle is removed.

Additional Topic: Secondary Storage (or Mass Storage) - Hard Disk Drives



The Hard Disk drives have been the main form of persistent data storage in computer systems for decades. Hard Disk Drive (HDD) is an I/O device.

Geometry of a disk Drive

A disk consists of a large number of sectors (also called blocks), each of which can be read or written. Each **Sector** is 512 bytes (traditional design) or 4KB (modern design) in size. The sectors are numbered from 0 to n - 1, where n is the number of sectors on the disk. Hence, we can view the disk as an array of sectors.

Points to note regarding disks:

1. Accessing two blocks near one another within the address space of the disk will be faster than accessing two blocks which are further apart.
2. If we need to access any k blocks, then the access will be the fastest if the blocks are in a contiguous chunk as compared to the blocks being located in random locations.

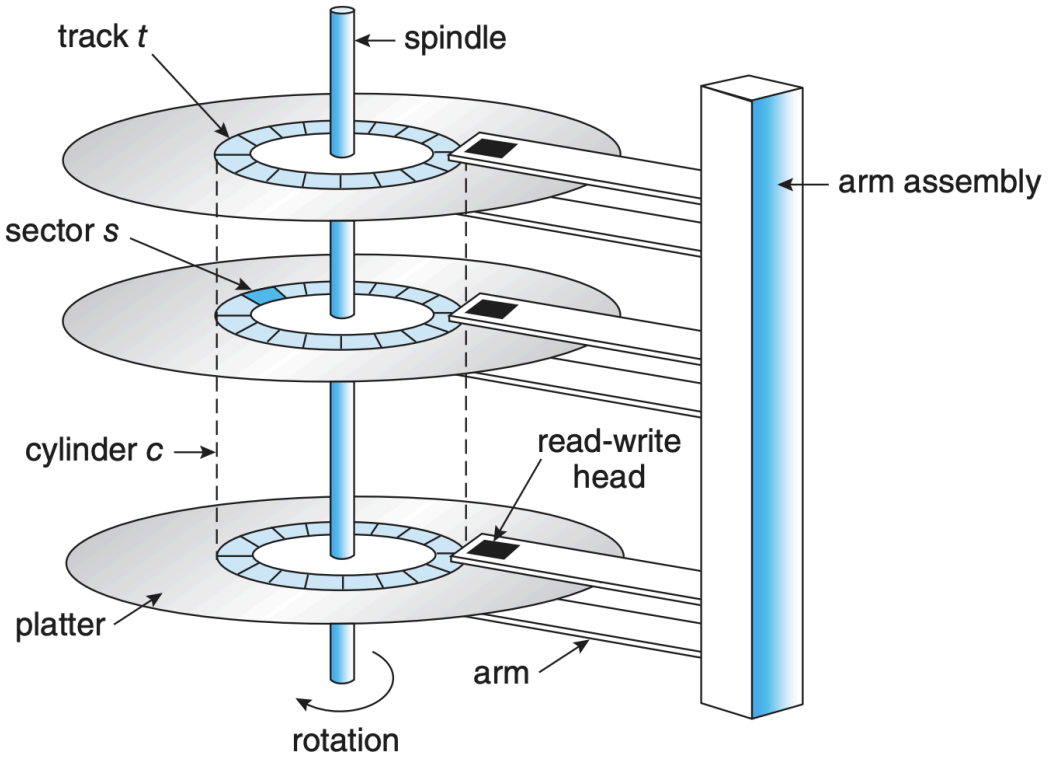
Components of hard disk:

Platters - A Platter is a circular hard surface on which the data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters, each platter has 2 sides or surfaces. The platter is made of some hard material like Aluminium, then coated with a thin magnetic layer.

All the platters are bound together around the **spindle**. The Spindle is connected to a motor that spins the platters around at a fixed rate. This rate of rotation is measured in Rotation per minute (RPM), modern values for RPM are in the range 7,200 - 15,000. Using the RPM value we can determine how long it will take for a single rotation of the platter to complete.

Data is encoded on each surface in concentric circles of sectors, each such concentric circle is called a track. A single surface can contain thousands and thousands of tracks tightly packed together.

To read or write data to a surface, a Disk Head is needed. There is one disk head per surface. The disk head is attached to a disk arm. The Disk arm positions the disk Head over the desired track.



(Visualization of a disk with multiple platters)

Rotational Delay - Once the disk head has been positioned over the desired track, the disk head must wait for the desired sector to rotate under it, this wait / delay is known as Rotational Delay.

Seek Time - The seek time is the delay incurred waiting for the disk head to move to the correct track. As mentioned before a platter contains thousands and thousands of tracks, and if we want to access a sector in a particular track then the disk Head will need to move to that particular track.

Seek times along with Rotational delays are costly disc operations.

Transfer time - Once the disk head has been positioned over the desired sector, the final phase of I/O will take place and the data will be read

from or written to the sector. The time taken to perform the actual read / write is known as the Transfer time.

$$T_{I/O} = T_{Rotational\ Delay} + T_{Seek\ Time} + T_{Transfer\ Time}$$

Disk Scheduling

Unlike Job Scheduling, in the case of Disk Scheduling we can guess how long a disk request will take, so approaches based around SJF will work well.

=> **Shortest Seek Time First (SSTF)**: Fulfill the request with the shortest seek time first, this approach orders the I/O requests by track. The disk head will serve the requests closest to it, in terms of the track at all times. SSTF leads to starvation of requests involving far away tracks.

=> **Elevator / SCAN**: The Elevator / SCAN Algorithm moves back and forth across the disk servicing requests across the tracks. In other words this algorithm sweeps the disk from inner to outer (or outer to inner) tracks and then sweeps it again in the reverse direction, the algorithm services any requests it encounters during its sweep, i.e. it sweeps across the disk servicing requests. If a request comes in for a sector located on a track which has already been serviced in the current sweep, then the incoming request is not handled immediately, rather it is added to a queue and will be taken care of in the following reverse sweep.

=> **Shortest Positioning Time First (SPTF)**: Similar to SSTF, but it also accounts for Rotational Delays.

Files and Directories

A file is simply a linear array of bytes, to which we can read and write data. Each file has an internal low level name (also called inode number).
A directory contains a list of files and other directories, specifically it contains a list of (user readable names, low level names) pairs. For example suppose we have a file with the user readable name “foo” and and the low level name “10”. The directory which contains foo will thus have an entry (“foo”, “10”) for this file. Directories like files too have low level names.

In UNIX systems, the directory hierarchy begins at the root directory (/).
Absolute path - Path of the file wrt the root directory
Relative path - Path of the file wrt the current working directory.

File APIs:

=> Creating and opening files - “open”

*fd = **open(pathname, flags, mode)** opens the file identified by pathname, returning a file descriptor used to refer to the open file in subsequent calls. If the file doesn’t exist, open() may create it, depending on the settings of the flags bitmask argument. The flags argument also specifies whether the file is to be opened for reading, writing, or both. The mode argument specifies the permissions to be placed on the file if it is created by this call. If the open() call is not being used to create a file, this argument is ignored and can be omitted.*

The “open” system call is used to open files, if the file does not exist then open will create it if the O_CREAT flag is passed.

While opening the file we provide some flags to specify the access mode (read only, write only, read/write) and some open file status flags.

Pathname:

Absolute or relative path to the file.

Flags:

Access mode flags:
O_RDOLNY (open the file in read-only mode)
O_WRONLY (open the file in write-only mode)
O_RDWR (open the file in read-write mode)

Open file status flags:

O_APPEND, O_SYNC, O_NONBLOCK,
O_TRUNC (truncate the file to zero bytes, i.e. remove any existing content).

Mode:
The mode argument specifies the permissions to be set on the file.

A call to open returns the file descriptor of the opened file, a file descriptor is just an integer private per process which is used to access a file. A file descriptor is like a file handle or a pointer to an object of type file. Once we have the file descriptor we can perform other operations with the file like reading, writing.
OS manages the file descriptors on a per-process basis.

Example: xv6

```
struct proc {
        ...
        struct file *ofile[NOFILE]; // Open files
        ...
};
```

As mentioned above the OS manages the file descriptors at the process level. Each process has a list (or array) of open file descriptors. Each entry in this table points to an entry in the system wide open file table. The open file table contains information for all the open files across the system, like the current file offset, flags (for example: whether the file is readable or writable). In addition, each entry in the open file table points to an entry in the inode table.

For each open file the OS tracks the current file offset, the offset is the location in the file from where the next read or write will take place, basically it is a pointer to a particular byte in the file. OS tracks the current offset for all the files via the system-wide open file table. The file offset is changed implicitly by calls to read and write, however we change it explicitly as well by calling “lseek”.

=> Changing file offset - “lseek”

Use the lseek system call to explicitly change the file offset.

off_t lseek(int fd, off_t offset, int whence);
The first argument is familiar (a file descriptor). The second argument is the offset, which positions the file offset to a particular location within the file. The third argument, called whence for historical reasons, determines exactly how the seek is performed. From the man page:
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current

location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
The file offset is initialised to zero when the file is opened.

=> File Control - "fcntl"
The fcntl system call can be used to retrieve and modify the access mode and the open file status flags of a file.
The fcntl call takes the file descriptor and a command which indicates the action. Depending on the command an additional argument (...) needs to be specified.

Use the F_GETFL command to retrieve the flags.
Use the F_SETFL command to modify the existing flags specified during the open call.

A note on shared file table entries:
In many cases the mapping b/w a file descriptor and an open file table entry is one to one, i.e. each descriptor corresponds to a unique entry in the open file table.

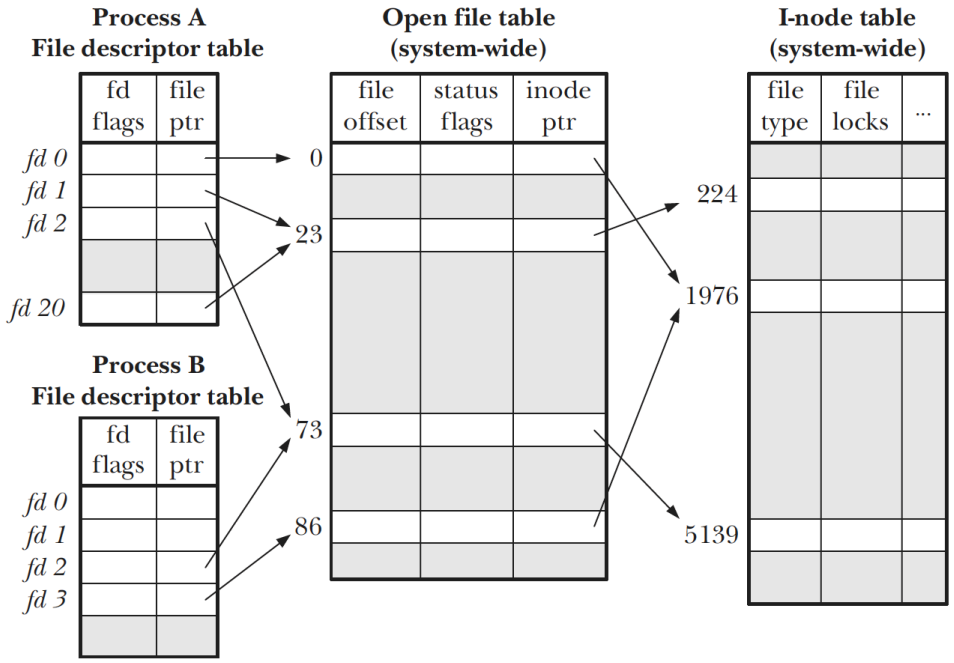


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

However it is possible for an open file table entry to be shared, i.e. more than one descriptor can refer to the same file table entry in other words to the same underlying file.
These descriptors can be within the process or even across multiple processes. So for example it is possible for file descriptor #4 of process A and file descriptor #6 of process B to refer to the same underlying file.

How can the open file table entry be shared?
If we try to open the same file twice within a process or across multiple processes, then that wouldn't result in the file table entry being shared. Instead a separate open file table entry will be created for each instance of the file being opened, and each entry will have its own unique file offset.

When is the open file table entry shared?
When a process calls fork, its child process will inherit the file descriptors of the parent process, i.e. the descriptors in the child and the

parent process will be pointing to the same open file table entry, i.e. to the same underlying file.

We can also achieve this behaviour by using dup and dup2 system calls.
dup(fd) - Create and return a new file descriptor that refers to the same underlying open file as fd. The descriptor returned will be the smallest available.
dup2(oldfd, newfd) - Similar to dup, however we can specify the descriptor we need. If newfd is open, the dup2 will close it first.

File links

Hard links:
Using the link system call we can create a hard link to a file. The **link** system all takes 2 arguments: the old file name and a new file name. The system call links the new file name to the old one and essentially creates another way to refer to the same file.

kanema@KANEMA-M-V93L files % cat file
hello
kanema@KANEMA-M-V93L files % ln file file2
kanema@KANEMA-M-V93L files % cat file2
hello

link creates another name and refers it to the same inode number as the original file.

kanema@KANEMA-M-V93L files % ls -i file
49436618 file
kanema@KANEMA-M-V93L files % ls -i file2
49436618 file2

kanema@KANEMA-M-V93L files %
As can be seen both file and file2 names are referring to the same inode number.

When a file is being created, a couple of things happen. First a structure called inode will be allocated which will track virtually all the information regarding the file (like its size, disk address of the file blocks etc). Second, a human readable name will be linked to the file and the link will be put in a directory (Remember directory is a list of <name, low level name (or inode number)> mappings).

To delete a file, we use the rm command which internally calls the “unlink” system call. The unlink system call removes the link between the human readable name and the inode number of the file. Within the inode structure a reference count or link count is maintained, which tracks how many different file names have been linked to this particular inode. When unlink is called, it decrements the reference count. Once the reference count reaches zero only then the filesystem frees the inode and related data blocks, and thus truly deletes the file.

Symbolic (or Soft) links:
A hard link creates another name that can be used to refer to the file. Even if the original filename is deleted the new name can still be used to access it.
Symbolic links are themselves a file type. To create a symbolic link use the “ln” command with the “-s” flag.

kanema@KANEMA-M-V93L files % ln -s file file2
kanema@KANEMA-M-V93L files %
kanema@KANEMA-M-V93L files % cat file2
hello

kanema@KANEMA-M-V93L files %
kanema@KANEMA-M-V93L files % stat file
16777232 49618195 -rw-r--r-- 1 kanema staff 0 6 "Feb 21 10:02:08 2025" "Feb 21 10:02:06 2025" "Feb 21 10:02:06 2025" "Feb 21 10:02:06 2025" 4096 8 0 file
kanema@KANEMA-M-V93L files %
kanema@KANEMA-M-V93L files % stat file2
16777232 49618210 lrwxr-xr-x 1 kanema staff 0 4 "Feb 21 10:02:10 2025" "Feb 21 10:02:10 2025" "Feb 21 10:02:10 2025" "Feb 21 10:02:10 2025" 4096 0 0 file2
kanema@KANEMA-M-V93L files %

Checking the output of ls -al
kanema@KANEMA-M-V93L files % ls -al

-rw-r--r-- 1 kanema staff 6 Feb 21 10:02 file
lrwxr-xr-x 1 kanema staff 4 Feb 21 10:02 file2 -> file

The first character in the left-most column is a “-” for regular files, a “d” for directories and an “l” for soft links.

As can be seen the size of the symbolic link (file2) is 4 bytes, this is because the data of the link file is the pathname of the linked-to file.

With symbolic links, dangling references are possible if the original file name is deleted. This will cause the soft link to point to a pathname that no longer exists.

Directory APIs:

To create a directory use the mkdir system call. When a directory is created it has 2 entries, one entry (.) that refers to itself (i.e. a link

referencing the directory) and one entry (..) that refers to its parent (i.e. a link referencing the directory’s parent)

Appendix

How does the Operating System boot up?

Step - I: Power ON

When we switch on the PC, the power supply component of the PC will be powered on, this component is responsible for providing power to every hardware component.

Step - II: CPU Loads the BIOS or UEFI.

The CPU gets powered on, it is the most important component in the boot process.

BIOS (Basic Input output system) is a small program stored in the BIOS chip (non-volatile, ROM storage). UEFI (unified extensible firmware interface) is an upgraded version of BIOS.

Step - III: BIOS (or UEFI) run tests and initialize hardware.

=> BIOS loads some settings from a memory area. This memory area is backed by a CMOS battery.

=> Once the settings are loaded, the actual BIOS program is loaded along with the settings.

=> The program will perform POST (power on self test), this is a series of tests performed on each hardware component to check if it is functioning as expected.

Step - IV: BIOS or UEFI hands off to boot device

The boot device or bootloader contains the instructions to load the operating system.

Boot device can be: disk (HDD / SSD), CD or USB device.

The bootloader initializes the operating system. The boot device contains the bootloader instructions, the bootloader is itself a program, which could be stored in:

1. MBR (Master boot record): Stored in the 0th index of the disk
2. EFI (Extensible Firmware interface) - A separate dedicated partition in the disk stores the bootloader.

Once the bootloader is found, the BIOS instructs the CPU to bring the bootloader into memory from this location, and then run this program to load the OS, at this point the job of BIOS is done.

Step - V: bootloader loads the Operating System
Bootloader is a program which initializes the operating system, OS vendors provide their own bootloader programs.

windows: bootmgr.exe
mac: boot.efi
linux: grub

Types of OS

Single Process OS: Only one process executes at a time from the ready queue, and the process will run to completion. Eg. MS DOS 1981

Batch Processing OS: Jobs are submitted to the OS in batches, however within a batch all the jobs are executed in a sequential manner, like in the case of a single process OS. Eg. Atlas.

Multiprogramming OS: In case of multiprogramming OS, multiple processes (code and data) are kept in memory, if the currently executing process performs some blocking operation like I/O, the OS can switch to running another job in the ready queue, hence the CPU doesn't sit idle while the process completes I/O instead it starts running another program, thereby improving the CPU utilization. Eg. The Dijkstra multiprogramming system. Characteristics:

a. Single CPU
b. Context Switching

Multitasking OS: It is an extension of multiprogramming OS, however instead of waiting for a process to perform I/O and then switching to another one, we use the concept of time sharing, whereby each process runs for some time, then it is stopped and the OS switches to run another process, which improves the system responsiveness. To achieve time sharing we use a timer so that each process runs only for a fixed amount of time. Eg. CTSS. Characteristics:

a. Single CPU
b. Context Switching
c. Time Sharing

Multiprocessing OS: Similar to multitasking OS, but there are multiple CPUs. Example: Windows NT. Improves reliability and throughput.

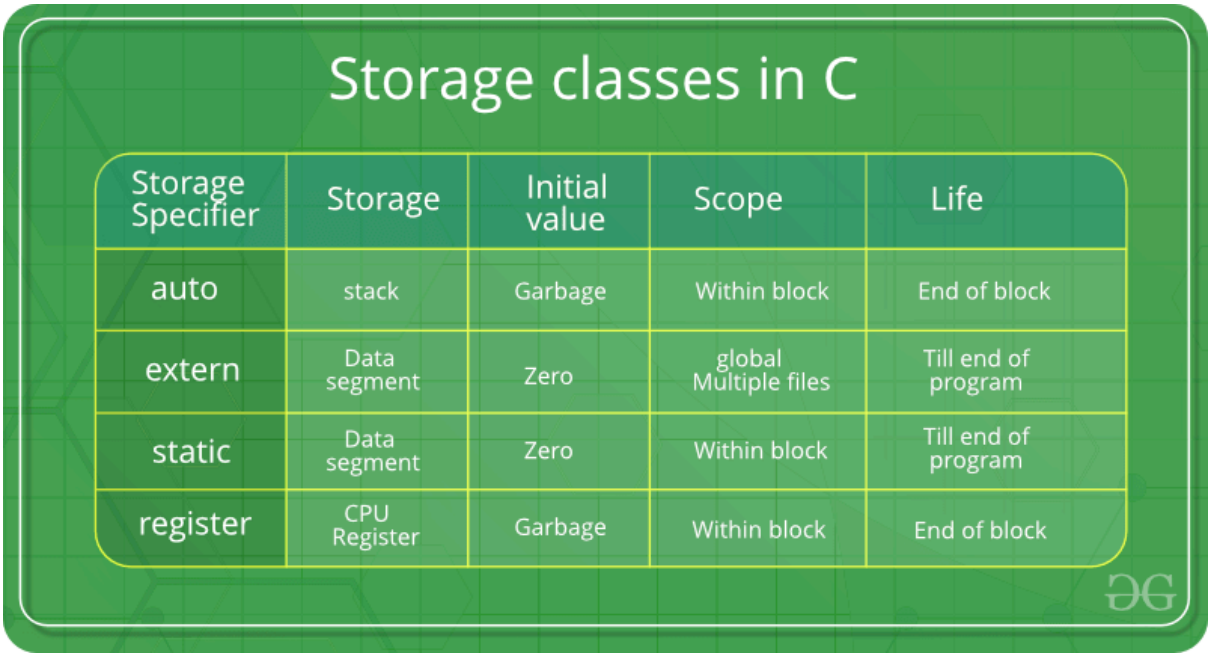
Concurrency vs Parallelism

Concurrency is a condition in which two or more tasks can start, run and complete in overlapping time periods. It doesn't necessarily mean that the tasks will be running at the same instant. For example multitasking on a single core machine. Concurrency is a more generalized form of parallelism.

Parallelism is a condition that arises when two or more tasks literally run at the same time, for example a multicore processor on which two threads are executing simultaneously.

Appendix: C

Storage Classes in C:



Static Modifier - Static variables are initialized only once when the program starts and remain in memory till the program is terminated. => Static variables can only be initialized to constant literals or constant variables (const) => Static variables and functions cannot be accessed outside the file in which they are defined (note: In C functions are global by default).
Auto modifier - Auto variables are defined inside a block, and are destroyed once the block ends.
Extern modifier - Extern keyword is used when we need to share variables across multiple files, i.e. use a variable which has been designed in another file. ‘extern’ keyword declares the variable and doesn't define it again.

extern int var: Declares the variable “var”, but doesn’t define it, i.e. no memory is allocated for “var”. Instead it asks the compiler that the variable “var” is defined somewhere else in the compilation unit and we just want to use, not re-define it. Functions are by default extern. Note: Extern variables need not be global, what extern keyword tells the linker is to go outside of the current scope and the definition of the declared variable will be found.
Register Modifier: We can add the register modifier to frequently accessed variables. The register keyword hints the compiler to store these variables in the register memory, which is the fastest i.e. lowest access time. Do note, we can only hint the compiler to store the variable in register memory, whether the compiler stores the given variable in a register or not is determined by the compiler. Further, the compiler performs some optimizations itself.

Interprocess Communication:

To communicate b/w processes we can use:

- Pipes
- FIFOs (Named Pipes)
- Signals