Process Environment ,Process Control and Process Relationships

Dr. Dipali Meher
mailtomeher@gmail.com

Agenda

Source: The Design of Univ Operating System- Maurice Bach

Introduction

Process states

Context of a process

Process creation

Process Termination

Process Control Block-Process ID,Obtaining the process ID and Parent Process Id, changing the size of the process

The Shell-Shell prompt, Shell Types,

Running New Processes- vfork, execl.atexit,wait,waitpidwaited functions

Environment List

Agenda

Source: The Design of Univ Operating System- Maurice Bach

Memory Layout of a C Program

Setjump,longjump

Getrlimit and setrlimit functions

Rules for changing the resource limits

System functions-

Launching and waiting new process

Race conditions

Changing user IDs and Group IDs

Daemons

Users and group

Process Scheduling

Agenda

Source: The Design of Univ Operating System- Maurice Bach

classification of Process(I/O bound, Processor Bouns)

Yielding the processes

Threads

Process Priorities

Processor Affinity

Orphan Process

Introduction

Source: The Design of Univ Operating System- Maurice Bach

Definition: The process can be defined as any one of the following:

i) A program in execution(most commonly used)

ii) An Asynchronous Activity

iii) The entity to which processors are assigned

iv) The dispatchable unit

The difference between program and process can be described as a process a processes an active entity where a program is a passive entity.

Source: The Design of Univ Operating System- Maurice Bach

Being passive a program is only a part of a process. On the other hand a process includes:

i) Current value of a program counter(PC)

ii) Contents of the processors registers

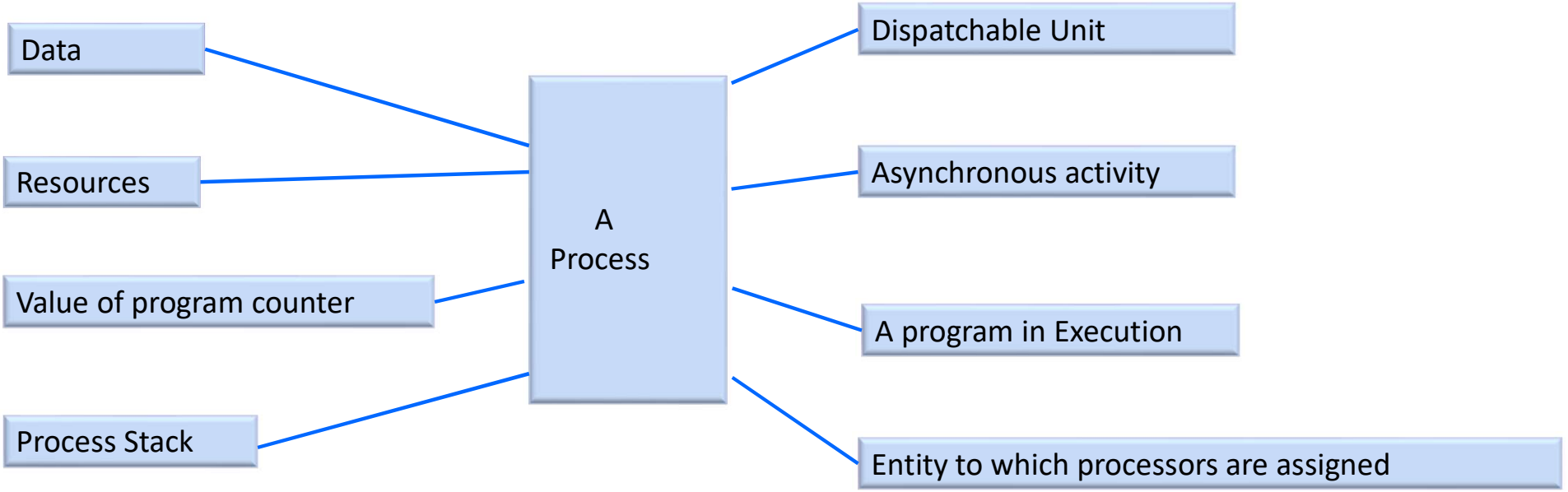
iii) Value of variables

iv) The process stack(SP) which typically contains temporary data such as subroutine parameter, return address and temporary variables

v) A data section that contains global variables

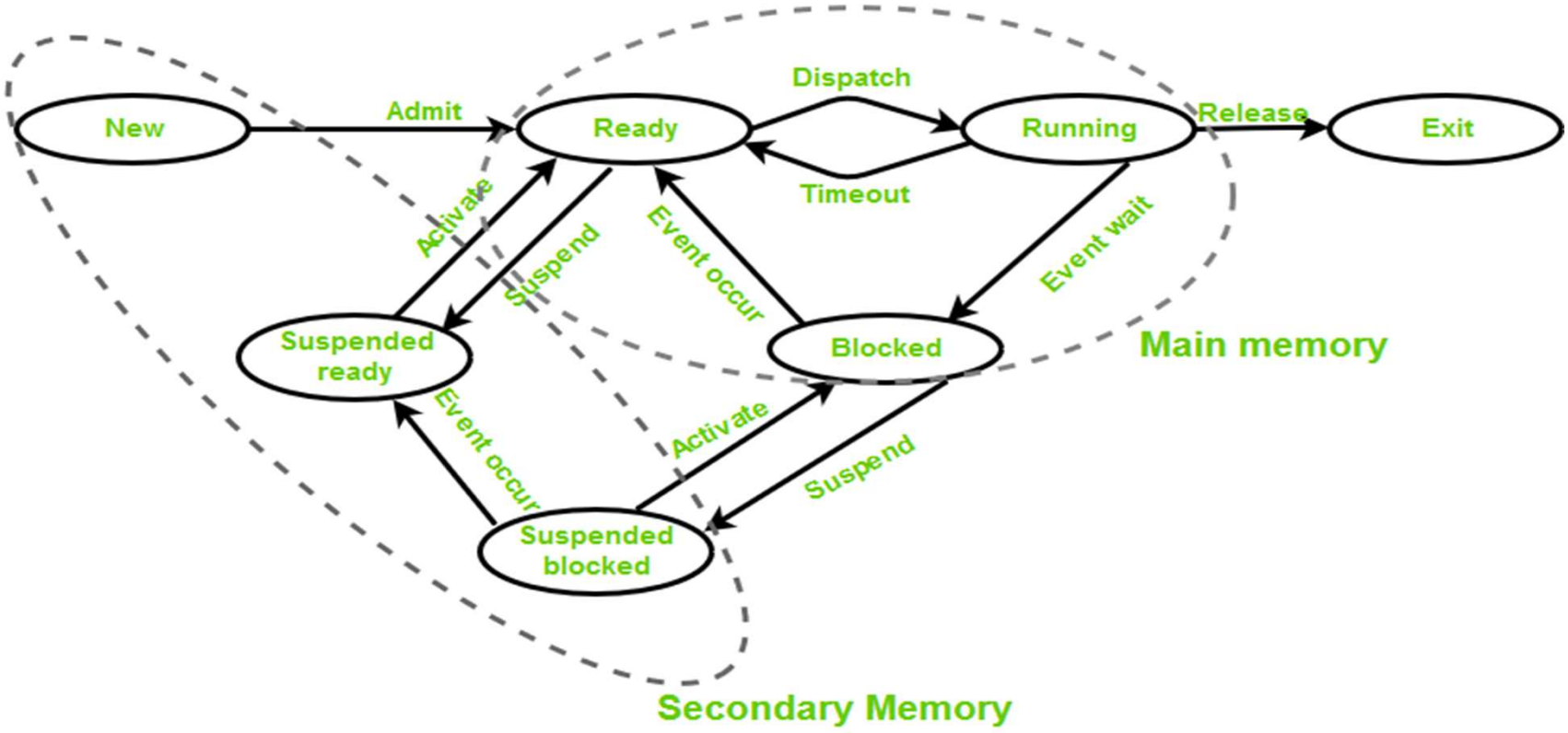
Source: The Design of Univ Operating System- Maurice Bach

A process is the unit of work in a system



Process States

Source: The Design of Univ Operating System- Maurice Bach



Process States

Source: The Design of Univ Operating System- Maurice Bach

The state consists of everything to resume the process execution if it is somehow put aside temporarily. The states consists of at least following;

i) Code for the program
ii) Programs static data
iii) Programs dynamic data
iv) Programs procedure call stack
v) Operating systems resource in use
vi) Contents of program counter
vii) Contents of program status

Process States

Source: The Design of Univ Operating System- Maurice Bach

•**New (Create) –** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.

•**Ready –** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.

•**Run –** The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.

•**Blocked or wait –** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.

•**Terminated or completed** – Process is killed as well as PCB is deleted.

•**Suspend ready –** Process that was initially in the ready state but was swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler is said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.

•**Suspend wait or suspend blocked –** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.

Context of a Process

Source: The Design of Univ Operating System- Maurice Bach

The context of a process is stored in the **Process Control Block** (PCB) and contains the process register information, process state, and memory information. The dispatcher is responsible for context switching. It saves the context of the old process and gives control of the CPU to the process chosen by the short-term scheduler.

After normal initiation of the process instance:

i) When a process instance runs normally through to completion, the process state changes from running to finished

ii) If a fault reaches the process boundary then process is put into failing state

iii) The process stays in the failing state while the fault handler runs

iv) After this, the process instance is put into failed state.

Saving Context of a Process

Source: The Design of Univ Operating System- Maurice Bach

The context of a process consists of:

- Contents of its (user) address space, called as *user level context*
- Contents of hardware registers, called as *register context*
- Kernel data structures that relate to the process, called as *system context*

Switching among process involves saving the context of the process into its PCB and placing it on some queue, depending on the cause of the context switch.

The process of saving the context of one process and loading the context of another process is known as **Context Switching**.

Source: The Design of Univ Operating System- Maurice Bach

Components of register context

The register context consists of following components:

i) Program Counter
ii) Processor status register
iii) Stack pointer
iv) General purpose register

Source: The Design of Univ Operating System- Maurice Bach

Process Creation

Due to following 4 principal event that led to process creation:

i) System initialization

ii) A user request to create a new process

iii) Initialization of a batch job

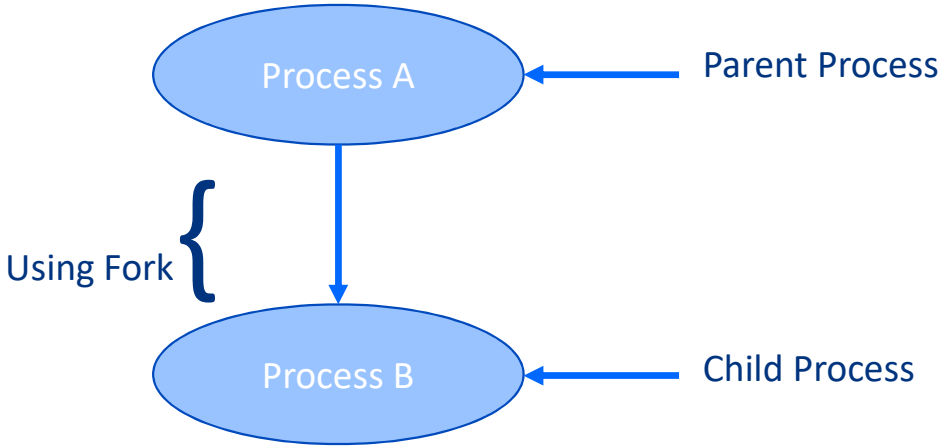
iv) Execution of a process creation system calls by a running process.

Daemon

Source: The Design of Univ Operating System- Maurice Bach

Foreground processes interact with user while background processes that stay in background sleeping but suddenly become active. The background process is also called a daemon.

A Process may create new process by ‘fork’ system call.



Only one process is needed to create child process

Source: The Design of Univ Operating System- Maurice Bach

After ‘fork’ the two processes(parent and child) processes have the same memory image the same environment string and the same open files and resource limits.

Following are the reasons for creating a process:

i) User logs on

ii) User starts a program

iii) Operating system creates process to provide service

iv) Some programme start another process

Source: The Design of Univ Operating System- Maurice Bach

While implementing parent and child process two questions should be answered

i) Should the child be a duplicate of the parent or a new process?

ii) Should the parent wait for the child to complete or not?

Unix ‘fork’ and ‘exec’ allow any permutation of the above options

Source: The Design of Univ Operating System- Maurice Bach

Process Termination

A process is terminated when it finishes executing its last statement, this returning its resources to the same and its process control block is erased.

The new process terminated the existing process usually due to following reasons:

i) Normal Exit: Most process terminate because they have completed their job

ii) Error Exit: When a process discovers a fatal error

iii) Fatal Error: An error caused because of a bug in program

iv) Killed by another process: A process executes s system call by telling the operating system to terminate some other process (KILL)

Process Control Block

Source: The Design of Univ Operating System- Maurice Bach

A process in an operating system is represented by a data structure known as process control block or process descriptor. The PCB contains important information about the specific process including:

i) The current state of process

ii) Unique identification of a process in order to track the correct information

iii) A Pointer to parent process

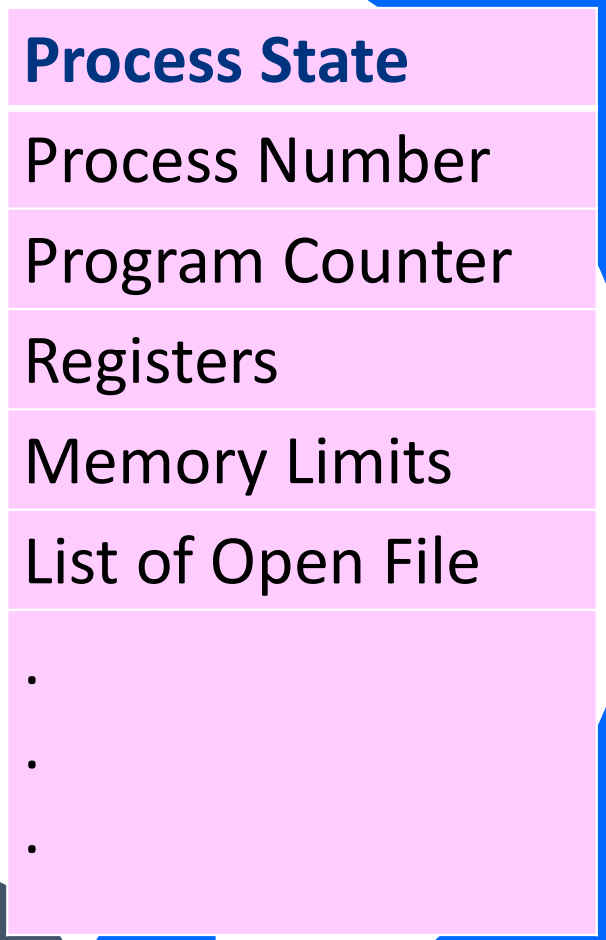
iv) A pointer to child process

v) The priority of Process

vi) A register save area

vii) Pointers to locate memory of processes

viii) The processor it is running on



Source: The Design of Univ Operating System- Maurice Bach

Obtaining the process ID and Parent Process ID

Each process is represented by unique identifier known as Process ID.

The PID is guaranteed to be unique at any single point of time.

There are various types of processes such as:

i) Idle Process

ii) Init process

By default Kernel imposes a maximum process ID value 32768.. Kernel allocated process Id to a process in linear fashion.

Source: The Design of Univ Operating System- Maurice Bach

Obtaining the process ID and Parent Process ID

The process ID can be obtained using getpid(). The getpid() system call returns the process ID of the invoking process.

```
#include<sys/types.h>
#include<unistd.h>
pid_t getpid(void);
```

Source: The Design of Univ Operating System- Maurice Bach

Changing the size of the process

A process may increase or decrease the size of its data region by using the brk system call.

Syntax

brk(ends);

Where ends is the value of the highest virtual address of data region of the process .

User can also call

oldendds=sbrk(increment)

If the process is calling brk to free previously allocated space, the kernel releases thememory;if the process accesses virtual addresses in pages that it has released, it incurs a memory fault

Source: The Design of Univ Operating System- Maurice Bach

The Shell

The shell provides you with an interface to the UNIX system, It gathers input from you and executes program based on that input. When a program finished executing, it displays that programs output.

A shell is an environment in which we run out commands, programs and shell scripts. There are different flavors of shells like operating systems. Each flavor of shell has its own set of recognized commands and functions.

Source: The Design of Univ Operating System- Maurice Bach

Shell Prompt

The prompt $ which is called command prompt is issued by the shell. The shell reads your input after you press enter. It determines the command you want executed by looking at the first word of your input.

A word is unbroken set of characters, spaces and tabs and separate words.

Shell Types

In UNIX there are two major types of shells.

i) The Bourne shell, If you are using a bourne type shell, the default prompt is the $ character.

ii) The c shell . If you are using Ctype shell, the default prompt is the % character.

Source: The Design of Univ Operating System- Maurice Bach

Subcategories of Bourne Shell

- Bourne Shell(sh)
- Korn Shell(ksh)
- Bourne Again shell(bash)
- POSIX shell(sh)
- The original UNIX shell was written in the mid-1970s by Stephen R, Bourne while he was at & T Bell Labs in New Jersey.
- The Boune shell was the first t or on UNIX systems thus it is referred to as “the shell”

Source: The Design of Univ Operating System- Maurice Bach

Running a New Process

The act of creation of new process is called forking and is executed by fork() system call.

The new process created by fork() is called as child process. This function called once but it will return twice. (As return values of child process is 0 and Parent process id Process ID of new child process)

The child process is a copy of the parent process. Child process gets a copy of the parents data space, heap and stack.

Source: The Design of Univ Operating System- Maurice Bach

How to handle descriptors of processes?

i) The parent process waits for a child to complete(parent will not require any descriptors). The child process will update offset of shared descriptor.

ii) After fork the parent process closes the descriptors that it does not need and the child does the same thing. This way neither interfaces with others open descriptors

vfork function

Source: The Design of Univ Operating System- Maurice Bach

Vfork() is also system call which is used to create new process. New process created by vfork() system call is called child process and process that invoked vfork() system call is called parent process. Code of child process is same as code of its parent process. Child process suspends execution of parent process until child process completes its execution as both processes share the same address space.

Difference Between fork and vfork

Source: The Design of Univ Operating System- Maurice Bach

FORK()	VFORK()
In fork() system call, child and parent process have separate memory space.	While in vfork() system call, child and parent process share same address space.
The child process and parent process gets executed simultaneously.	Once child process is executed then parent process starts its execution.
The fork() system call uses copy-on-write as an alternative.	While vfork() system call does not use copy-on-write.
Child process does not suspend parent process execution in fork() system call.	Child process suspends parent process execution in vfork() system call.
Page of one process is not affected by page of other process.	Page of one process is affected by page of other process.
fork() system call is more used.	vfork() system call is less used.
There is wastage of address space.	There is no wastage of address space.
If child process alters page in address space, it is invisible to parent process.	If child process alters page in address space, it is visible to parent process.

Exec system calls family

Source: The Design of Univ Operating System- Maurice Bach
Source: The Design of Univ Operating System- Maurice Bach

- execl(),execlp(),execle(),execv(),execvp()
- exit()
- atexit()
- wait()
- waitpid()

exec system call

The fork() returns the PID of the child process. If the value is non-zero, then it is parent process's id, and if this is 0, then this is child process's id.

The exec() system call is used to replace the current process image with the new process image. It loads the program into the current space and runs it from the entry point.

So, the main difference between fork() and exec() is that fork starts new process which is a copy of the main process. the exec() replaces the current process image with new one, Both parent and child processes are executed simultaneously.

execl() system call

Source: The Design of Univ Operating System- Maurice Bach

#include<unistd.h>

int execl(const char *path, const char *arg,….);

A successful execl() call changes not only the address space and process image but certain other attributes of the process:

i) Any pending signals are lost

ii) Any memory locks are dropped

iii) Most process statistics are reset

iv) Most thread attributes are returned to the default values

v) Anything that exists safely in user space, including features of c library, such as atexit() behavior is dropped.

vi) Any signals that the process is catching are returned to their default behavior.

Exec family functions

Source: The Design of Univ Operating System- Maurice Bach

p: the full path of user for a given file
e: new environments supplied for the new process

#include<unistd.h>

i) Int execlp(const cha r* file,const char * arg..)

ii) Int execle(const char *path, const char * arg,…,char * const envp[]);

iii) Int execv(const char * path, char *cont argv[]);

iv) Int execvp(const char *file,char * const argv[]);

v) Int execve(const char * filename, char * const argv[],char * const envp[]);

Exit functions

Source: The Design of Univ Operating System- Maurice Bach

A process can be terminated by 5 ways

1) Executing a return from the main() function. This is equivalent to calling exit
2) Calling the exit function
3) Executing a return from the start routine of the last thread in the process
4) Calling the exit or exit() function.

Source: The Design of Univ Operating System- Maurice Bach

Abnormal termination of a process

1) Calling abort: this is a special case of the next item. It generate the SIGABRT signal
2) When the process receives certain signals
3) The last thread responds to a cancellation request.

atexit() function

Source: The Design of Univ Operating System- Maurice Bach

This function to be called for normal process termination

#include<stdlib.h>

int atexit(void(*function)(void)).

This function registers the given function to be called at normal process termination either via exit or via return the programs main function. Functions so registered are called in the reverse order of their registration; no arguments are passed.

atexit function returns value 0 if successful otherwise returns non zero value.

Functions registered using atexit() are not called if a process terminated abnormally because the delivery of a signal

Wait and waitpid function

Source: The Design of Univ Operating System- Maurice Bach

wait(): When the process terminated either normally or abnormally the kernel notifies the parent by sending the SIGCHLS signal to parent. A process that calls wait or waitpid can

I) Block if all of its children are still running

II) Return immediately with the termination status of a child

III) Return immediately with an error, if it does not have any child process.

Difference between wait and waitpid

Source: The Design of Univ Operating System- Maurice Bach

wait	waitpid
Can block the caller until a child process terminated	It has an option which prevents child process from blocking
It waits for the child process termination otherwise it blocks the caller.	The wait pid function does not wait for the child that terminated first

pid_t wait(int *_status_);
pid_t waitpid(pid_t _pid_, int *_status_, int options**);**

Source: The Design of Univ Operating System- Maurice Bach

The **wait**() system call suspends execution of the current process until one of its children terminates. The **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

Source: The Design of Univ Operating System- Maurice Bach
Source: The Design of Univ Operating System- Maurice Bach

The call *wait(&status)* is equivalent to:

```
waitpid(-1, &status, 0);
```

wait3() and wait4() functions

pid_t wait3(int *status, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);

wait3() waits of any child, while **wait4**() can be used to select a specific child, or children, on which to wait.

These functions are having additions argument that allows the kernel to return a summery of resources used by the terminated process and all its child process

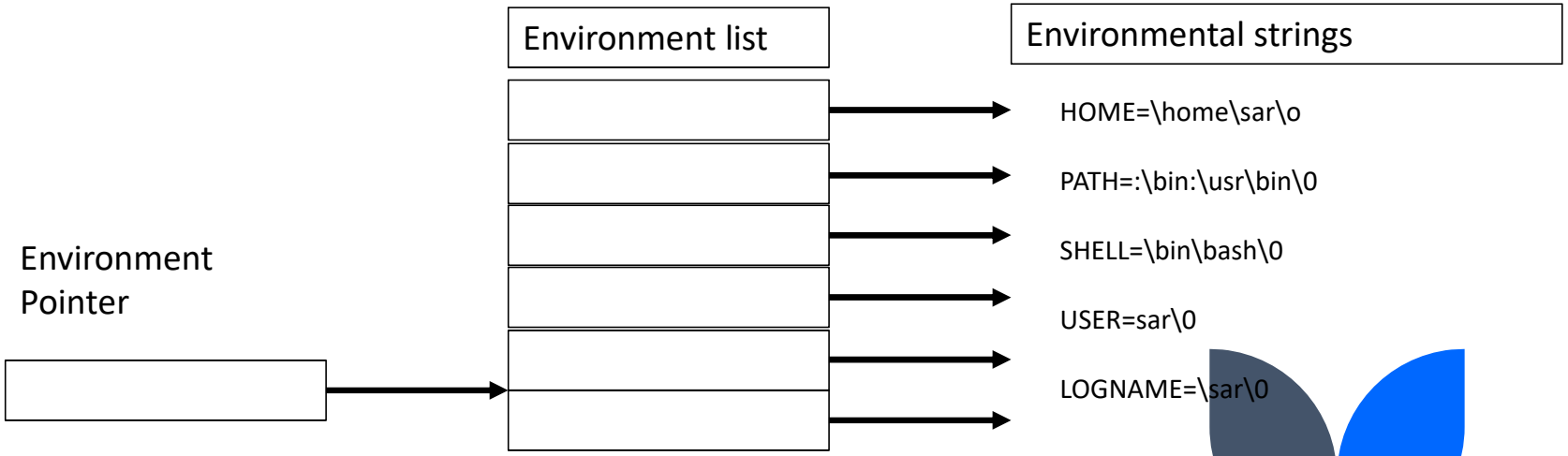
Environment List

Source: The Design of Univ Operating System- Maurice Bach

Each program is having an environment list which contain an array of character pointers, with each pointer containing the address of a null terminated c character string.

Environment list consists of name=value

The environment pointer is an array of pointer which points to the environment strings.



Source: The Design of Univ Operating System- Maurice Bach

Memory Layout of a CProgram

C program has been composed of following pieces

i) Text segment:

ii) Data segment:

iii) Uninitialized data segment:

iv) Stack:

v) Heap:

Source: The Design of Univ Operating System- Maurice Bach

Memory Layout of a C Program

C program has been composed of following pieces

i) Text segment: The machine instructions that the CPU executes. The text segment is sharable so that only a single copy needs to be in memory for frequently executed programs such as text editors, c compilers the c shells

Memory Layout of a C Program

Source: The Design of Univ Operating System- Maurice Bach

C program has been composed of following pieces

i) Data segment: It contains variables that are specifically initialized in the program

ii) Uninitialized data segment: It is also called the BSS statement(block started by symbol). Data in this segment is initialized by the kernel to arithmetic 0 or null pointer before the program starts executing.

iii) Stack: automatic variables are stored, along with the information and saves each time a function is called.

iv) Heap: It deals with dynamic memory allocation

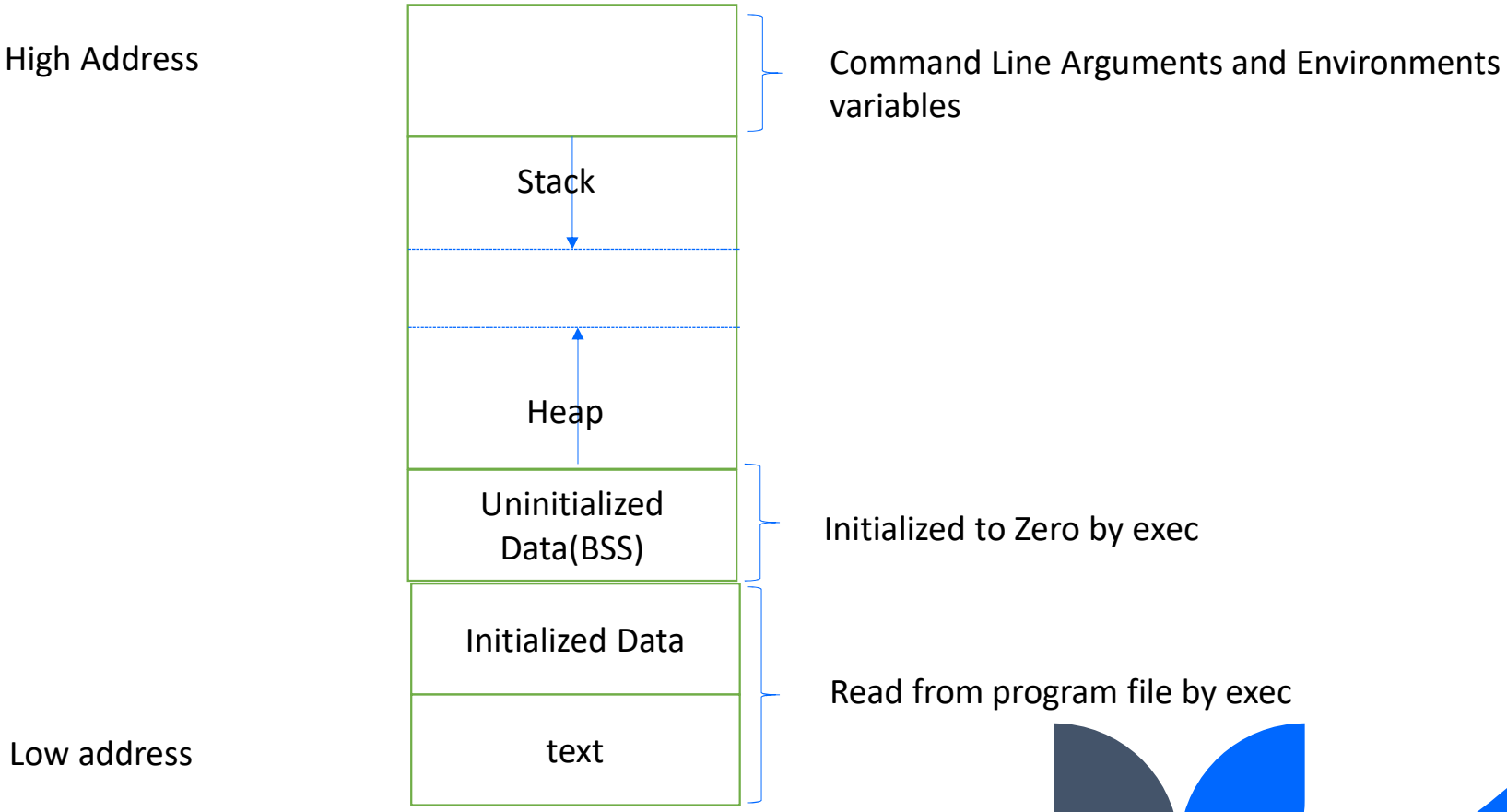
Important Terms

Source: The Design of Univ Operating System- Maurice Bach

Shared Libraries: This library removes common library routines from the executable file and maintain one single copy of routine in memory so that all processes can reference one library. So, this reduces the size of executable file. Library functions can be replaced with new versions without having to relink edit every program that uses the library.

Environment Variables: The UNIX kernel never looks at these strings. Interpretation of environment variable sis depends on various applications. The shells use numerous environments variables.

Typical memory arrangement of c program



setjump and longjump functions

Source: The Design of Univ Operating System- Maurice Bach

These functions are used for exception handling. They basically allows the user to discard a bunch of stack frames and try to create fault tolerance boundaries but not in every function.

Setjump saves the contents of the registers

Longjump can restore them later

Longjump returns the state of the program when setjump was called.

Source: The Design of Univ Operating System- Maurice Bach

Th estate of the program depends completely on the contents of its memory(i.e. code, global. Heap and stack) and the contents of its registers The contents of registers include:

i) The stack Pointer

ii) Frame pointer(FP)

iii) Program Counter(PC)

iv) Process State Register(PS)

Setjump and longjump

Source: The Design of Univ Operating System- Maurice Bach

setjump save the contents of the registers

Longjump restores contents of register later

longjump returns the state of the program when setjump was called.

#include<setjmp.h>

Int setjmp(jmp_buf env)

longjmp(jmp_buf env, int val);

Setjump and longjump functions

Source: The Design of Univ Operating System- Maurice Bach

"Setjump" and "Longjump" are defined in setjmp.h, a header file in C standard library.

- **setjump(jmp_buf buf)** : uses buf to remember the current position and returns 0.
- **longjump(jmp_buf buf, i)** : Go back to the place buf is pointing to and return i.

Features of setjmp function

Source: The Design of Univ Operating System- Maurice Bach

- Stores information about current stack an frame pointers
- Typically stored in static data areas structure
- Avoid passing pointers through all children
- Children generate exception intermediate functions are irrelevant

Features of longjmp function

Source: The Design of Univ Operating System- Maurice Bach

- Restores the stack pointer
- Does not restore the contents of th elocka stack frame
- Doe not know about other resources(e.g dynamically allocated memory)
- Supports several GNU tool(-gcc and –gdb)

The setjump and longjump fuctions try to recover from a variety of low level errors

getrlimit and setrlimit functions

Source: The Design of Univ Operating System- Maurice Bach

Every process has a set of resource limits some of which can be queries and changes by following functions:

```
#include<sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr)
int setrlimit(int resource, const struct rlimit *rlpter)
struct rlimit
{
Rlimit rlim_cur;
Rlimit rlim_max;
};
```

Rules for changing resource limits

Source: The Design of Univ Operating System- Maurice Bach

1) A process can change into soft limit to a value less than or equal to its hard limit
2) A process can lower its hard limit to a value greater then po equal to its soft limit
3) Only a super user process can raise a hard limit

(hardlimit maximum value of rlim_cur)

System Functions

Source: The Design of Univ Operating System- Maurice Bach

These functions helps to execute a command string from within a program. The advantage using a system function, instead of using fork and exec directly, instead of that system function handles all the required error handling as well and signal handling.

Race Conditions

Source: The Design of Univ Operating System- Maurice Bach

A Race condition occurs when multiple processes are trying to perform some operations on shared data and the final outcome depends on the order in which the processes run.

The form function provides the ground for race conditions.

A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes.

Race Conditions

Source: The Design of Univ Operating System- Maurice Bach

We cannot predict parent or child process runs first.

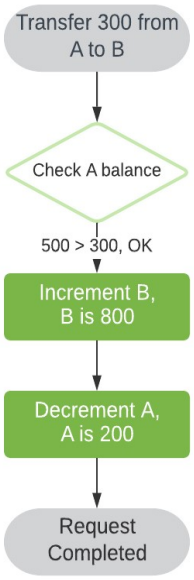
Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
static void charatatime(char *);
int main() {
 int pid;
if((pid=fork())<0) printf("fork error\n");
else if(pid==0) charatatime("output from child\n");
else charatatime("output from parent\n"); _exit(0);
}
static void charatatime(char *str) \
{ char *ptr; int c;
setbuf(stdout,NULL);
for(ptr=str;(c=*ptr++)!=0;)
putc(c,stdout);
}
```

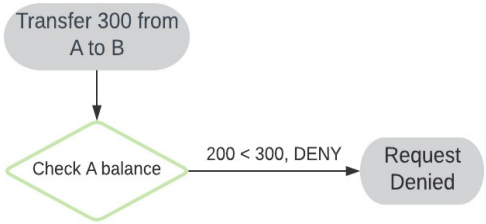
Race Conditions

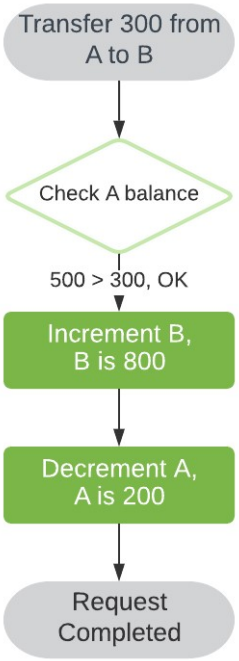
Source: The Design of Univ Operating System- Maurice Bach

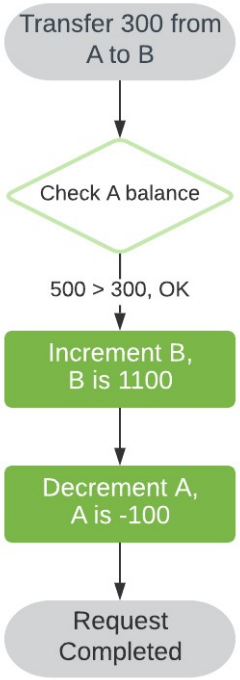
To avoid race conditions or pulling of nay process to execute some form of signaling is required between multiple processes. Various form of Interprocess Communication(OPS) can also be used for this purpose.



Let's say we have accounts A and B, each having a balance of 500, and we perform two attempts to transfer 300 from A to B:







However, if these two attempts are kicked off simultaneously, in different processes or threads, we may observe some undesired behavior:

Given unpredictable thread scheduling, the order of specific steps is arbitrary. We've encountered a race condition due to the interleaving of our execution flows.

Race Conditions

Source: The Design of Univ Operating System- Maurice Bach

To avoid race conditions, any operation on a shared resource – that is, on a resource that can be shared between threads – must be executed atomically. One way to achieve atomicity is by using *critical sections* — mutually exclusive parts of the program. Another approach is to use *atomic operations* to take advantage of the hardware's ability to ensure indivisibility.

Source: The Design of Univ Operating System- Maurice Bach

Changing User IDs and Group Ids

Processes are associated with users and groups. Processes run under the appropriate users and groups. Privileges and access control are based on user and group ids. When our programs need additional privileges or a need to gain access to resources that they currently are not allowed to access, they need to change their user or group id that has the appropriate privilege or access. Similarly when programs need to lower their privileges or prevent access to certain resources they do so by changing either user ID or group ID to an ID without privilege or ability access to the resource.

Changing User IDs and Group Ids

Source: The Design of Univ Operating System- Maurice Bach

Set UID function is used to set real user ID

#include<unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);

Daemons

Source: The Design of Univ Operating System- Maurice Bach

A daemon is a process that runs in the background, not connecting to any controlling terminal. Daemons are normally started at boot time, run as root or some other special user and handle system level tasks.

A daemon has two requirements

1) It must run as a child of init
2) It must not be connected to a terminal.

A program performs following steps to become daemon

Source: The Design of Univ Operating System- Maurice Bach

i) Call fork(): This creates a new process which will become the daemon

ii) Call exit(): This ensures that the original parent is satisfied and that its child terminated. The daemons parent is no longer running sand that the daemon is not a process group leader.

iii) Call setsid(): Giving the daemon a new process group and session both of which have it as leader.

iv) Change the working directory to the root directory via chdir(). This is done because the inherited working directory can be any where on the file system.

v) Close file descriptors

vi) Open file descriptors0,1,2(0: std in,1: std out 2: std err)and redirect them to '/dev/null'

Source: The Design of Univ Operating System- Maurice Bach

A program performs following steps to become daemon

i) Call fork(): This creates a new process which will become the daemon

ii) Call exit(): This ensures that the original parent is satisfied and that its child terminated. The daemons parent is no longer running sand that the daemon is not a process group leader.

iii) Call setsid(): Giving the daemon a new process group and session both of which have it as leader.

iv) Change the working directory to the root directory via chdir(). This is done because the inherited working directory can be any where on the file system.

v) Close file descriptors

vi) Open file descriptors0,1,2(0: std in,1: std out 2: std err)and redirect them to '/dev/null'

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
int main(int argc, char* argv[])
{ FILE *fp= NULL;
pid_t process_id = 0;
pid_t sid = 0; // Create child process
process_id = fork();
// Indication of fork() failure
if (process_id < 0)
{ printf("fork failed!\n");
// Return failure in exit status exit(1); }
// PARENT PROCESS. Need to kill it.
if (process_id > 0)
{ printf("process_id of child process %d \n", process_id);
// return success in exit status
exit(0);
}
//unmask the file mode
umask(0);
//set new session
sid = setsid();
```

Source: The Design of Univ Operating System- Maurice Bach

```
if(sid < 0)
{ // Return failure exit(1); }
// Change the current working directory to root.
chdir("/");
// Close stdin. stdout and stderr
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
// Open a log file in write mode. fp = fopen
("Log.txt", "w+"); while (1)
{ //Dont block context switches, let the process sleep
for some time
sleep(1);
fprintf(fp, "Logging info...\n"); fflush(fp);
// Implement and call some function that does core
work for this daemon.
}
fclose(fp);
return (0); }
```

https://www.thegeekstuff.com/2012/02/c-daemon-process/

Source: The Design of Univ Operating System- Maurice Bach

Users

Users can be any one of the following:

i) root: Super user(uid==0)

ii) Daemon: Handle networks

iii) Nobody: Owns no files, used as default ser for unprivileged operations

iv) Web browser can run with this mode

   i) User needs to log in with a password. The encrypted password is stored in /etc/shadow

   ii) User information is stored in/etc/passwd, the place that was used to store passwords

Source: The Design of Univ Operating System- Maurice Bach

Groups

i) It is better to assign permissions to a group of users. i.e. we like to assign permission based on groups

ii) A user has a primary group and this is the one associated to the files the user created.

iii) Any user can be member of multiple groups

iv) Group member information is stored in /etc/group

v) Command groups uid displays the groups that uid belongs to

vi) For systems the use NIS(Network Information Service), originally Yellow Page(YP)

vii) The command ypcat group(can display all the groups and their members)

Source: The Design of Univ Operating System- Maurice Bach

Process Scheduling

i) The process schedule is the component of a kernel that selects which process to run next.

ii) Scheduler is a subsystem of the kernel that divides the finite resource of processor time among a systems process.

iii) Multiple processes can run concurrently

iv) The process which interact with users, read, write files heavily or respond to I/O or network events tend to spend a lot of time blocked while they wait for resources to become available and they are not runnable during those periods.

Source: The Design of Univ Operating System- Maurice Bach

Goals of Process Scheduling

i) Within short time frame linux allows concurrent execution of proceses.

ii) Deals with scheduling: concerned with then to switch and which process to choose

iii) System tick id set to 1ms

Source: The Design of Univ Operating System- Maurice Bach

Scheduling Policy

The scheduling algorithm must fulfill conflicting objectives

i) Fast process response time

ii) Good throughput for background jobs

iii) Avoidance of process saturation

iv) Reconciliation of the needs of low and high priority processes

v) Running process is not terminated when its time slice or quantum expires. The process switches to another

vi) No additional code needs to be inserted in the programs to ensure CPU time scheduling

vii) The scheduling policy is also based on the ranking processes according to their priority

viii) In UNIX process priority is dynamic

Source: The Design of Univ Operating System- Maurice Bach

Classification of Processes

There are three classes of Processes . They are

1) **Batch Processes**: These processes do not need user interaction. Example compilers, database search engines, scientific computations
2) **Realtime Processes**: These processes should never be locked by lower priority processes. E.g. video and sound applications, robot controllers, data collection program from physical sensors
3) **Interactive Processes**: These types of processes interact constantly with users. Eg. Command shells, text editors, graphical applications

Source: The Design of Univ Operating System- Maurice Bach

Process Scheduling system calls

i) getpriority(): Get the maximum static priority of a group of processes

ii) setpriority(): set the priority of a group of processes

iii) nice(): changes the static priority of a conventional process

Every conventional process has its own static priority (value) decided by schedular to rate the process with respect to other processes in the system.

Kernel gives static priority of a process 100: highest priority 139: Lowest priority

A new processes always inherits static priority of a process.

Differentiate between I/O bound process and Processor bound process

Source: The Design of Univ Operating System- Maurice Bach

CPU Bound Process	I/O Bound Process
CPU Bound means the rate at which process progresses is limited by the speed of the CPU	I/O Bound means the rate at which a process progresses are limited by the speed of the I/O subsystem
CPU bound process need very little I/o but require heavy computation.	IO bound process is the one that spends more of its time doing I/o then it spends on doing computation
They should be given low priority by schedular	They should be given high priority by schedular
Longer CPU bursts	Shorter CPU bursts
Example: Compiler, simulator, Scientific applications	Example Word Processors, Database applications

Source: The Design of Univ Operating System- Maurice Bach

Sched-Yield System call

The sched_ yield( ) system call allows a process to relinquish the CPU voluntarily without being suspended; the process remains in a task_running state, but the scheduler puts it at the end of the runqueue list. In this way, other processes that have the same dynamic priority have a chance to run. (which is same as preemptive multitasking of processes by kernel)

Source: The Design of Univ Operating System- Maurice Bach

Threads

The units of execution within a single process are known as threads. All the processes have at least one thread.

Every thread has its own set of registers, instruction pointer, and processor state.

A process can have large number of threads which perform different tasks but share same address space, dynamic memory, mapped files, object code, list of open files, and other kernel resources.

Source: The Design of Univ Operating System- Maurice Bach

Process Priorities

i) Linux supports process priorities. A process priority allows a process to run by lowering the priority of the other processes.

ii) Runnable processes are scheduled according their priority as highest to lowest.

iii) A priority is called as nice value. This nice value dictates(orders-get changes according to priority) when process runs.

iv) There are two types of processes in linux1) conventional 2) real time

v) Real time processes are having absolute priority.(they have 99 priority levels) RR or FIFO algorithm is used to select processes. Schedular will nor preempt A FIFO process in timer interrupt.

Source: The Design of Univ Operating System- Maurice Bach

Linux algorithm for conventional jobs:

i) Each process is given a nice level when started.

ii) The schedular divides the time to 'epoch' and each process is allowed to use a certain amount of time(time_quantum) during each epoch.

iii) When all ready processes used their time quantum, a new epoch begins. The time quantum of each process is increased by its base time quantum

Base time quantum=(20-nice levels) ticks

1 tick=10.5 ms

iv) Waiting processes may have some quantum left behind. Half of it carries forward to nest epoch. In this way I/O bounded processes are given highest priority than CPU bounded processes

The schedular always schedules the process with the highest "goodness" that is sum of remaining abs base time quantum.

Source: The Design of Univ Operating System- Maurice Bach

nice() system call

i) This system call is used to set to retrieve processes nice value.

#include<unistd.h>

int nice(int n)

Returned value

If successful, nice() return the new nice value minus (NZERO).

If unsuccessful, nice() returns -1 and sets errno to one of the following values:

i) The process is having nice value will be given highest priority to complete it self by kernel. Less nice value will have highest priority and high nice value is having lowest priority.

Source: The Design of Univ Operating System- Maurice Bach

get priority and set priority system call

i) These system calls operates on process, or process groups or user.

ii) A call to getpriority returns the highest priority of any of the specified processes where as a call to set priority s

iii) The getpriority() call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The setpriority() call sets the priorities of all of the specified processes to the specified value.ets the priority of all specified processes.

I/o priorities

Source: The Design of Univ Operating System- Maurice Bach

i) Processes are also having i/o priorities.

ii) i/O schedulers use a processes nice value to determine the I/O priority. For this ionice() system call is used.

ionice -c scheduling_class -n priority_nice_value command

Tag	Description
-c, –class class	Determine the name or number of the scheduling class to utilize; 0 for none, 1 for real-time, 2 for best-exertion, 3 for inactive.
-n, –classdata level	Determine the scheduling of class information. This possibly has an impact if the class acknowledges a contention. For constant and best-exertion, 0-7 are legitimate information (need levels), and 0 speaks to the most noteworthy need level.
-p, –pid PID…	Indicate the cycle IDs of running cycles for which to get or set the scheduling boundaries.

Example:**1.** To set a process, say having PID as 1 to be an idle I/O process.

```
ionice -c 3 -p 1
```

To run ‘bash’ or any other program as a best-effort program.

ionice -c 2 bash

To get Ids of running process

Ionice -u 1

Processor Affinity

Source: The Design of Univ Operating System- Maurice Bach

i) It refers to likehood of a process to be scheduled consistently on the same processor.

ii) **Processor affinity**, or **CPU pinning** or "cache affinity", enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. This is soft affinity.

iii) Hard affinity: Sometime user will force kernel to schedule a process for execution to some other processer.

iv) A process has affinity for processor on which it is currently running.

v) Processor affinity is **the probability of dispatching of a thread to the processor that was previously executing it.**

vi) **it is also known as cache affinity.**

Source: The Design of Univ Operating System- Maurice Bach

Resource limits

i) Each process in the system uses certain amount of different resources like files, CPU time, memory and so on.. Kernel imposes resource limits on processes. That every resource will run some number of processes.

ii) List: The list of resource limits used for context and processes within.

- i) ID: resource limit ID
- ii) Name: human readable identifier used in user space utilities
- iii) ulimit: command line switch for the ulimit utility
- iv) Unit: appropriate unit for limit
- v) Tag: special resource limit code to denote if resources are accounted, enforced.
- vi) Description of capability. flag effects
- vii) Description:

Orphan Process

Source: The Design of Univ Operating System- Maurice Bach

i) An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself.

ii) In UNIX OS any orphaned process will be immediately adopted by the special init system process. This operation is called as re-parenting and occurs automatically.

iii) A process will become orphan unintentionally as parent process may be terminated, crashed.

iv) The process group mechanism will not allow any process to become orphan

v) In case of server client architecture. Client may request anything to server but client may crashes after that (process at client end becomes parent and server end becomes child as it serves) but due to this situation server process become orphan.

Orphan Process

Source: The Design of Univ Operating System- Maurice Bach

These orphaned processes waste server resources and can potentially leave a server starved for resources. The orphan process problem can be resolved by:

i) By killing the orphan(examination)

ii) Expiration: each process is allotted a certain amount of time to finish before being killed

iii) Machines try to locate the parents of nay remote computations at which orphaned processes are killed(reincarnation)

Thank you