UNIX PROCESSES

- A Process is a program under execution in a UNIX or POSIX system.
- **main FUNCTION**
- A C program starts execution with a function called main
- The prototype for the main function is
- **Int main(int argc, char *argv[]);**
- where argc is the number of command-line arguments, and argv is an array of pointers to the arguments.
- When a C program is executed by the Kernel by one of the exec functions,a special start-up routine is called before the main function is called.
- The executable program file specifies this routine as the starting address for the program.This is set up by the Link Editor When it is Invoked by the C compiler.

PROCESS TERMINATION

- There are eight ways for a process to terminate.
- **Normal termination occurs in five ways:**
- **1. Return from main**
- **2. Calling exit.**
- **3. Calling _exit or _Exit**
- **4. Return of the last thread from its start routine.**
- **5. Calling pthread_exit from the last thread.**
- **Abnormal Termination Occurs in Three Ways.**
- **1.Calling abort 2. Receipt of a signal. 3. Response of the last thread to cancellation request.**

EXIT Functions

- Three Functions terminate a Program normally.
- _exit and _Exit which return to the Kernel immediately, and exit ,which performs certain cleanup processing and then returns to the Kernel.
- #include<stdlib.h>
- void exit(int status);
- void _Exit(int status);
- #include<unistd.h>
- Void _exit(int status)
- All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value.
- The exit(0) is same as return(0) from the main function.

_exit and exit :EXPLICIT PROCESS TERMINATION

- Most Processes that is started by Unix Kernel will eventually die.When a process dies ,the kernel closes all open files and frees all memory associated with the process(Like the address space and user area).
- Whether or not it also frees the process table entry(the proc structure) depends on whether the parent has waited for the child's death.Before we get into what exactly the parent wait's for ,Lets understand that a process can terminate in any of the ways.

_exit and exit :EXPLICIT PROCESS TERMINATION

- By falling through to the end of the program.This happens when you don't include an explicit exit or return call in main.The process then invokes an implicit return.
- By an explicit return statement in the main function.
- By the exit function or _exit system call anywhere in the program.
- On receipt of a signal which may terminate the process.
- We examine the exit library function and _exit system call.We have used exit in every C program used in this text. It's time to see what exactly exit does and how it differs from _exit.
- void _exit(int status);
- void exit(int status);

atexit Functions

- With ISO C , a process can register upto 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.
- **#include <stdlib.h>**
- **int atexit(void (*func)(void));**
- returns: 0 if OK, nonzero on error
- This declaration says that we pass the address of a function as the argument to atexit.
- When this function is called, it is not passed any arguments and is not expected to return a value.

Atexit Functions

- The exit function calls these functions in reverse order of their registration.
- Each function is called as many times as it was registered.
-

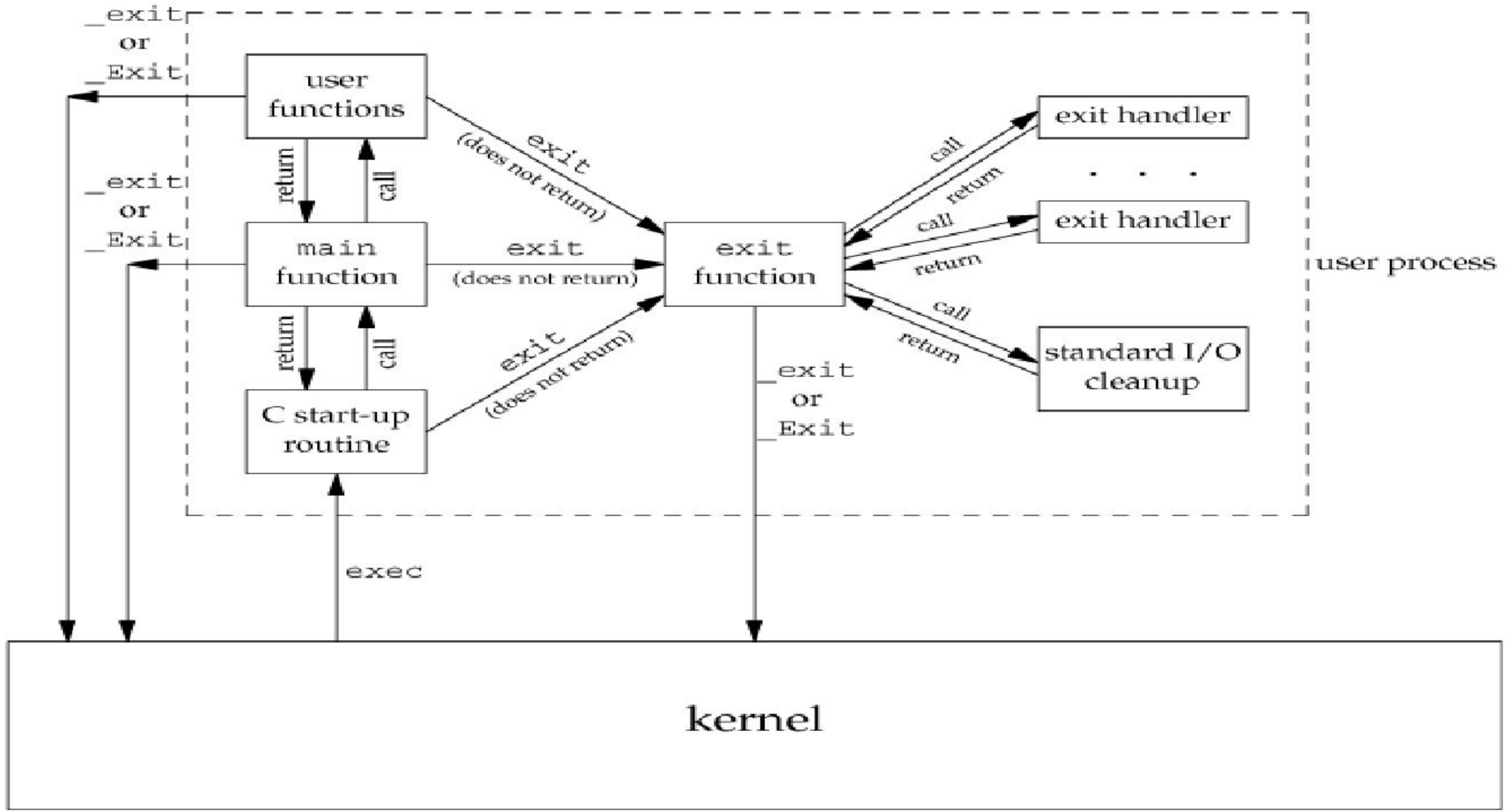
```
static void my_exit1(void);
static void my_exit2(void);
 int main(void)
{
 if (atexit(my_exit2) != 0)
     perror("can't register my_exit2");

 if (atexit(my_exit1) != 0)
     perror("can't register my_exit1");

 printf("main is done\n");
 return(0);
}
```

```
static void my_exit1(void)
{
printf("first exit handler\n");
}

static void my_exit2(void)
{
printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
main is done
first exit handler
second exit handler
```

UNIX PROCESSES

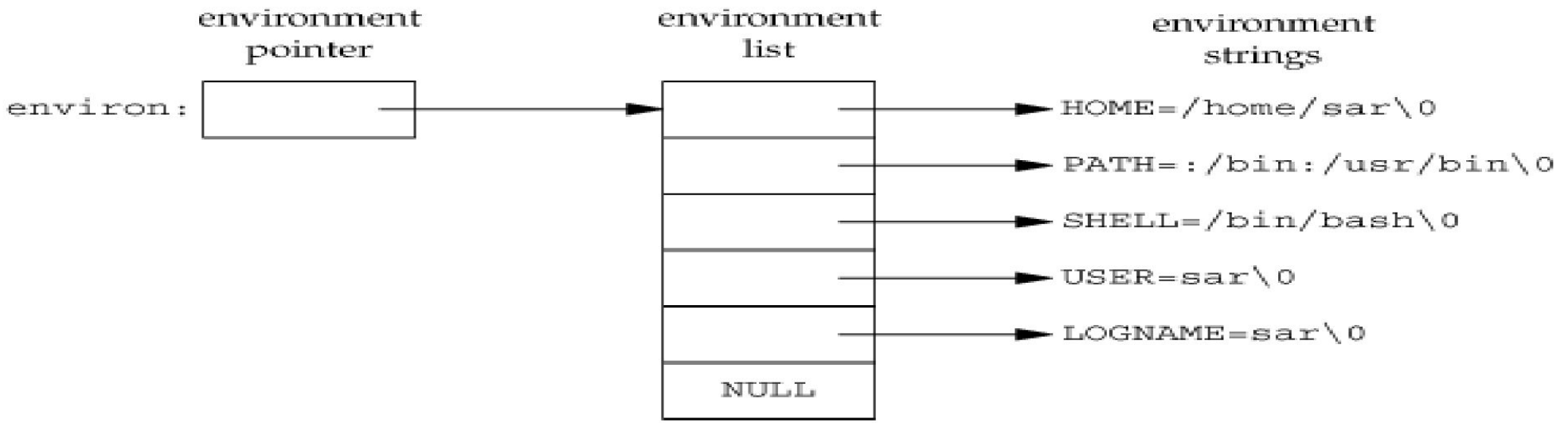


UNIX PROCESSES: COMMAND LINE ARGUMENTS

- When a program is executed, the process that does the exec can pass command-line arguments to the new program. Example: Echo all command-line arguments to standard output.
- **int main(int argc, char *argv[])**
- **{**
- **inti;**
- **for (i= 0; i< argc;i++)/* echo all command-line args*/**
- **printf("argv[%d]: %s\n", i, argv[i]);**
- **exit(0);**
- **}**

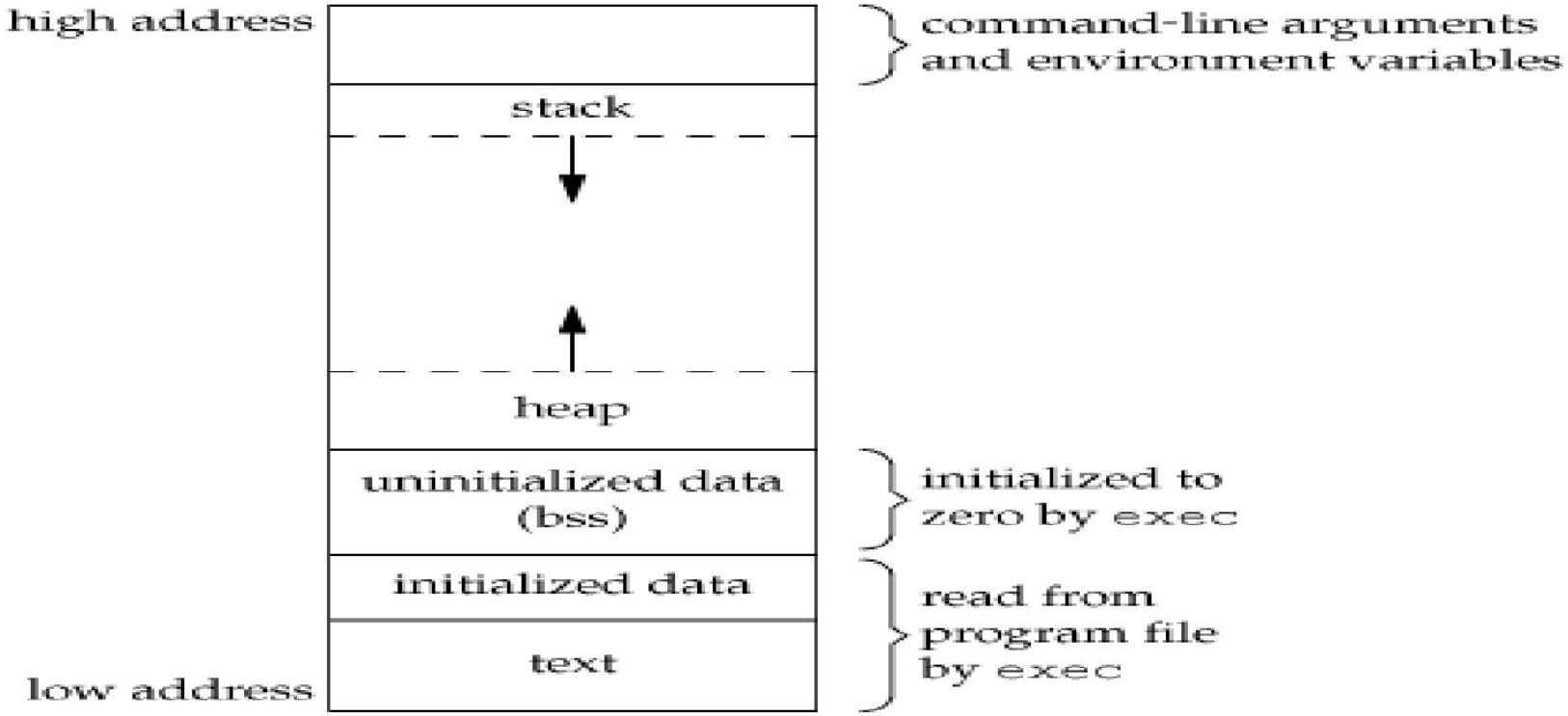
ENVIRONMENT LIST

- Each program is also passed an environment list.
- Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string.
- The address of the array of pointers is contained in the global variable environ: **extern char **environ;**
- Generally any environmental variable is of the form: ***name=value.***
-



Unix Processes

- **Heap:**
- Where dynamic memory allocation usually takes place.
- Historically,theheap has been located between the uninitialized data and the stack.



UNIX PROCESSES

- Historically, a C program has been composed of the following pieces:
- **Text segment**:
- The machine instructions that the CPU executes.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs,such as text editors,the C compiler, the shells, and so on.
- Also,the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

UNIX PROCESSES

- **Initialized data segment:**
- Usually called simply the data segment,containing variables that are specifically initialized in the program.
- For example,the C declaration.
- **Int maxcount=99;**
- Appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment:**
- Often called the"bss“ segment, named after an ancient assembler operator that stood for"block started by symbol.“
- Data in this segment is initialized by the kernel to arithmetic 0 or nullpointers before the program starts executing.

Unix Processes

- **Stack Segment.**
- Where automatic variables are stored,along with information that is saved each time a function is called.
- Each time a function is called,the address of where to return to and certain information about the caller's environment,such as some of the machine registers,are saved on the stack.
- The newly called function then allocates room on the stack for its automatic and temporary variables.
- This is how recursive functions in C can work.
- Each time a recursive function calls itself,a new stackframe is used,so one set of variables doesn't interfere with the variables from another instance of the function

UNIX KERNEL SUPPORT FOR PROCESSES

- ***fork*** and ***exec*** are commonly used together to spawn a sub-process to execute a different program.
- The advantages of this method are:
- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

UNIX KERNEL SUPPORT FOR PROCESSES

- **#include <unistd.h>**
- **pid_t getpid(void); Returns: process ID of calling process**
- **pid_t getppid(void); Returns: parent process ID of calling process**
- **uid_t getuid(void); Returns: real user ID of calling process**
- **uid_t geteuid(void); Returns: effective user ID of calling process**
- **gid_t getgid(void); Returns: real group ID of calling process**
- **gid_t getegid(void); Returns: effective group ID of calling process**

Fork() Function

- An existing process can create a new one by calling the fork function.
- **#include <unistd.h>**
- **pid_t fork(void);**
- Returns: 0 in child, process ID of child in parent, 1 on error.
- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0,where as the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allow a process to obtain the process IDs of its children.

Fork() Function

.)The reason fork returns value 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent.

- (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; The parent and the child do not share these portions of memory
- The parent and the child share the text segment .

Fork() Function

- The two main reasons for fork to fail are
- If too many processes are already in the system,which usually means that something else is wrong,or
- If the total number of processes for this real user ID exceeds the system's limit.
- There are two uses for fork:
- When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time.This is common for network servers,the parent waits for a service request from a client. When the request arrives,the parent calls fork and lets the child handle the request.The parent goes back to waiting for the next service request to arrive.

Fork() Function

.) When a process wants to execute a different program.This is common for shells.In this case,thechild does an exec right after it returns from the fork.

Eg Program.

```
Main()
{
fork();
 printf(“Hello World\n”);
}
```

- Some Operating Systems combine the operations from step2 (a fork followed by an exec) Into a single operation called spawn.

Fork() Function

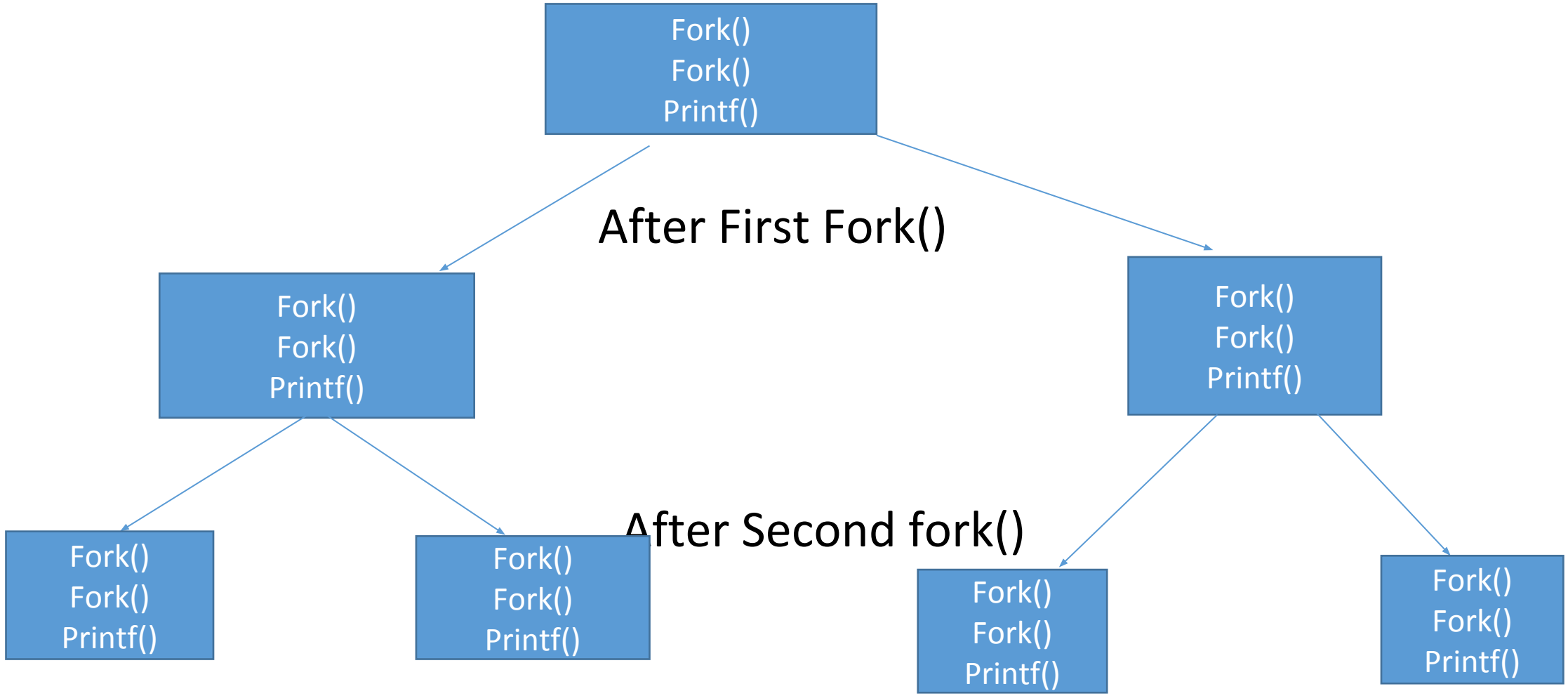
- The Statement Hello World will be displayed twice on the screen.The fork() creates a child that is duplicate of the parent process.
- The parent process in this case being the program listed above.Since now there are two identical processes in memory,the Hello World is printed twice.
- The Child process begins from fork() .All the statements after the call to fork() will be executed twice.Once by the parent process and once by the child process.But had there been any statement before the fork() they would have been only executed by the parent process.

Fork() Function

- What would happen if we had two calls to the fork() function,one below the other, in the same program.
- main()
- {
- fork();
- fork();
- printf(“Hello World\n”); }
- Here instead of 2 processes 3 processes will be created.This gives a total of 4 processes in memory:the parent its 2 children and one grandchild.
- The first call to fork() creates one child process.Now there are two processes.

  Both processes begin executing from the second call to the fork(),thus giving a total of four processes.

Fork() Function



Orphan Processes

- Normally after a fork() the time slice is given to the child process.Now suppose the parent process were to terminate before the child process,what would happen?.
- For one,we would have a fatherless child.In this case it will be adopted by init process(Process Dipatcher pID 1).
- After a Fork a parent process may choose to suspend its execution until its child process terminates by calling wait or waitpid system call, or it may continue execution Independently of its child process.
- A process can execute a different program by calling the exec system call.If the call succeeds the Kernel will replace the Process existing text,data and stack segment with a new set that represents the new program to be executed.
-

Vfork() Function

.)The function vfork has the same calling sequence and same return values as fork.

- The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- The vfork function creates the new process,just like fork,without copying the address space of the parent into the child,as the child won't reference that address space;the child simply calls exec(or exit)right after the vfork.
- Instead,while the child is running and until it calls either exec or exit,the child runs in the address space of the parent.

Vfork() Function

.)This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.

- Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit.When the child calls either of these functions,the parent process resumes.
- [This can lead to Deadlock if the child depends on further actions of the parent before calling either of these two functions.]
-

Vfork Function

vfork FUNCTION

Example of vfork function

```
int glob = 6;  /* external variable in initialized data */
int main(void)
{
int  var=88;          /* automatic variable on the stack */
pid_t      pid;
printf("before vfork\n");
if ((pid = vfork()) < 0)
    perror("vfork error");
else if (pid == 0)   /* child */
{
   glob++;                  /* modify parent's variables */
   var++;
   _exit(0);          /* child terminates */
 }
```

```
/* Parent continues here.*/
printf("pid = %d\n", getpid());
printf("glob = %d, var = %d\n", glob, var);

exit(0);
}
```

Output:

```
$ ./a.out
before vfork
pid = 29039
 glob = 7, var = 89
```

Vfork() Function

- Here the incrementing of variables done by the child, changes the value in the parent .Since the child run's in the address space of the parent,this does'nt surprise us.
- Notice in the program we call _exit instead of exit.As we described in section 8.5 , _exit does not perform any flushing of standard I/O buffers.
- If we call exit Instead,the output is different.
- $a.out
- Before vfork
- Here the output from the parent's printf has disappeared!.What's happening here is that the child calls _exit which flushes and closes all the standard I/O streams.
- This Include stdout

Vfork() Function

- Even though this is done by the child ,Its done in the parent address space,so all standard I/O File objects that are modified in the parent.
-

Wait() AND waitpid() FUNCTIONS

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- When a process calls wait or waitpid it can

  .)Block(If all of its children are still running).

  .) Return Immediately with the termination status of a child(If a child has terminated and is waiting for its termination status to be fetched.) Or
- Return Immediately with an error (If it doesn't have any child processes).
- When a process terminates either normally or abnormally ,the parent is notified by the kernel sending the parent the SIGCHLD signal.
- If the process is calling wait because it received SIGCHLD signal we expect wait to return immediately.

Wait and Waitpid Functions

- #include<sys/types.h>
- #include<sys/wait.h>
- Pid_t wait(int *statloc);
- Pid_t waitpid(pid_t pid, int *statloc, int options);
- Both Return process Id If Ok ,0 or -1 on error.
- Wait can block the caller until a child process terminates while waitpid has no option that control which process it waits for.
- If a child has already terminated and it is a zombie ,wait returns immediately with that child’s status.Otherwise it blocks the caller until a child terminates.

Wait and WaitPid Functions

- If the caller blocks and has multiple children ,wait returns when one terminates.We can always tell which child terminated because the Process ID is returned by the function.
- For both Functions the arguments statloc is a pointer to an integer .If this argument is not a null pointer ;the termination status of the terminated process is returned and stored in the location pointed by the argument.
- if we don't care about the termination status we just pass a Null pointer as this argument.
- Traditionally the Integer status that is returned by these two functions that has been defined by the Implementation with certain bits indicating the exit status (for a normal return), other bits indicating the significant number
- (for an abnormal return) one bit to indicate If a core file was generated and so on.

Wait and WaitPid Functions

- Posix.1 specifies that the termination status is to be looked at using various macros that are defined in <sys /wait.h>.
- There are three mutually exclusive macros that tell us how the process terminated and they all begin with WIF.
- Based on which of these macros is true ,other macro's are used to obtain the exit status ,signal Number and the like.These are shown in table below.

Waiting To pick up Child Status

- After a process has forked a child more often than not ,It is followed by a call to exec .The Two processes run independnently .So what does the parent do while the child process is executing.It can do two things
- Wait to gather the child process exit status.
- Continue execution without waiting for the child (and pick up the exit status later,if at all).
- The First is the normal shell behaviour when we enter the command.The shell exhibits its non waiting role when it runs a job in the background.
- When there's atleast one child process running,wait blocks till a child dies.

  It then returns the pid of the first dying child and places its exit status in stat_loc.

-

Wait and WaitPid Functions

MACRO	MEANING
WIFEXITED(status)	True if status was returned for a child that terminated Normally. In this case we can execute WEXITSTATUS to fetch the lower order 8 bits of the argument that the child passed to exit or _exit.
WIFSIGNALED(status)	True if status was returned for a child that terminated abnormally (By receipt of a signal that it did'nt catch). In this case we can execute WTERMSIG(status) to fetch the signal number
WIFSTOPPED(status)	True if status was returned for a child that is currently stopped.

Wait API

- The Kernel writes a process accounting record to a log file and clears the slot in the process table that was allocated to the child.
- The parent then resumes operation with the statement following exit.
- The variable stat_loc representing the termination status does’nt contain only the exit status but some other things as well-like the process stateand the signal that caused the process to be in that state.
- Separate macro’s are available to obtain the complete Information.
- The exit status itself is stored in the least significant bits of stat_loc.
- We have also achieved a rudimentary form of IPC ;The parent knows the child exit status.

Waitpid :

- Wait suffers from a number of limitations.It blocks till a child dies and ther's nothing that the process invoking it can do till that happens.
- Further if a parent spawns several children ,wait return's the moment one of them among process groups dies;it cant wait for a process with a specific PID to die.
- Finally wait can't handle process groups.
- Overcoming these Limitations is important in process management, and for these reasons the use of waitpid is recommended over wait,waitpid takes three arguments.
- Pid_t waitpid(pid_t pid, int *statloc, int options);
- The behaviour of waitpid is also controlled by the options setting.This can be zero or bit-wise OR of the constants WNOHANG,WUNTRACED ETC.

Waitpid

- The second argument ,stat_loc has the same significance as in wait;the termination status of the process is stored there.
- The options setting enables waitpid to run in non-blocking mode, but there are other functions.The pid can take four types of values.
- If pid is -1,waitpid blocks till a child dies or changes state.
- if pid >0 waitpid waits for a specific process with pid as its PID.
- If pid=0 waitpid waits for any process belonging to the same process group as the process invoking the call.
- If pid < -1 waitpid waits for any process whose PGID is the same as the absolute value of pid.

Waitpid

- If options is 0,waitpid blocks till the child changes state,but the behaviour of the process is also influenced by the way pid is specified.
- For emulating wait(&status) we need to use waitpid(-1,&status,0).The advantage of using waitpid over wait now seem quite obvious.
- Waitpid need not block till a child dies,it can wait for a child with a specific PID to die and it can also handle process groups.

ZOMBIES and ORPHANS

- Though the normal shell behaviour is to wait for a child to die,that is not always practicable.
- With most shells today supporting job control it is usual for a process or process group to be in the background or remain in suspended state.
- However there are two things that can happen if the parent does’nt wait for the child to die.
- The child dies while the parent is still alive.
- The parent dies while the child is still alive.
- When the child dies it return its exit status to the parent,depending on the exit status of the child the parent will collect exit status of the child process through wait api.Until it collects the child is said to be in zombie state.

Exec :The Final step in Process Creation

- Forking is an essential phase of process creation,but more often than not,we don’t stop at fork.
- We want to run a separate program in a forked process.This is done by doing an exec.This operation replaces the entire address space of a process (the text,data and stack) with that of new program.
- The forking mechanism is responsible for creating processes,but it’s exec that actually executes programs on a UNIX system.
- Many of the attributes inherited during a fork don’t change with an exec.
- For instance the previous program’s file descriptors ,the current and root directory ,umask settings and the environment remain the same .
- Since no new process is created ,the PID does’nt change across an exec.

Exec Operation

- Exec overwrites the kernel I/O buffer's ,so make sure that you flush them before using exec.Fork inherits all I/O buffers which creates problems when using printf.
- The exec operation can be performed by six members of a family of one system call and five library function's,which we'll refer to simply as exec or the exec family.
- The entire set can be grouped into two parts ,which we'll call the "execl" set

  and the "execv" set because the function names begin with the string exec followed by either an l or a v.
- Two of the members also have the names execl and execv;the other four are simple derivatives of them and have more similarities than differences.

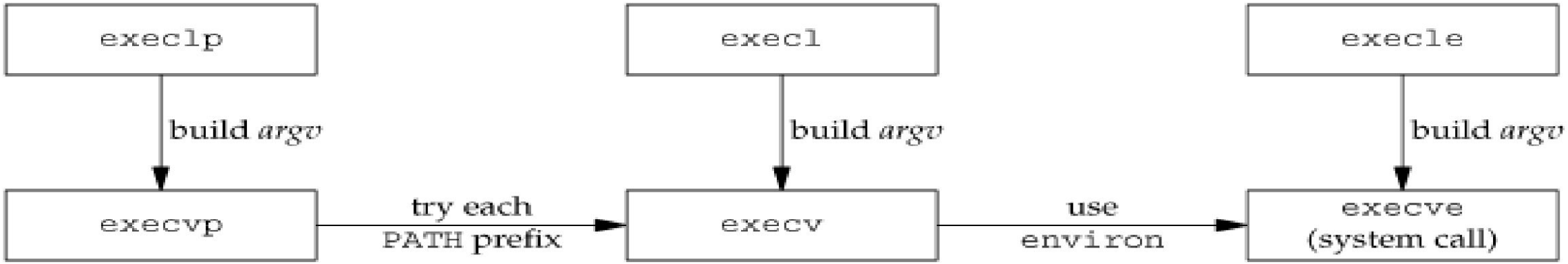
Exec Functions

- When a process calls one of the exec functions,that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created.Exec merely replaces the current process-its text,data,heap,and stack segments-with a brand new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0,  char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp []);

int execlp(const char *filename, const char *arg0,
 ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv []);
```

All six return: −1 on error, no return on success

Exec() Functions



- ***Relationship of the six exec functions***
- In many UNIX system implementations,only one of these six functions,**execve**,is a system call within the kernel.
- The other five are just library functions that eventually invoke this system call.

Exec Operation

- The l in execl(and its variants) represents a fixed list of arguments ,while v in execv (and its variants) signifies a variable number of arguments.
- The execl is used with a list comprising the command name and its arguments.
- int execl(const char *path, const char * arg0, …. /*, (char *) 0 */);
- We use execl when we know the number of arguments in advance .The first argument is a pathname which could be an absolute or relative pathname.
- The arguments to the command to run are represented as separate arguments beginning with the name of the command (* arg0).
- The ellipsis representation in the syntax (…../*) points to varying number of arguments.

Exec Operation

- To consider the example of execl which will run wc –l command with the filename foo as argument:
- execl(“/bin/wc” , “wc” , “ –l” , “foo”, (char *) 0);
- execl does’nt use PATH to locate wc so we must specify the pathname as its first argument .The remaining arguments are specified exactly in the way they will appear as main’s argument in wc.
- When we use exec to run a program ,there’s no provision to specify the number of arguments (no argc) exec has to fill the argument count by hand.The only way for execl to know the size of the argument list is to keep counting till it encounters the null pointer.

Execv :The key member of the V series

- To be able to run a command with any number of arguments ,you must use one of the function of the “execv” set.
- execv needs an array to work with.
- int execv(const char *path, char *const argv[]);
- Like in execl ,path represents the pathname of the command to run.The second argument is a pointer to an array of pointers to char.
- The array has to be populated by addresses that point to strings representing the command name and its arguments in this form they are passed to main function of the program to be executed.
- Eg char *cmdargs[]={“grep” , “-i” ,” –n”, “SUMIT” ,”/etc/passwd”, NULL);
- execv(“/bin/grep” , cmdargs);

Other Members of the “l” and “v” Series

- execlp and execvp :The requirements to provide the pathname of the command makes the previous exec calls somewhat inconvenient to use.
- Fortunately help is at hand in the form of he execlp and execvp functions

that use PATH to locate the command.

- They behave exactly like their other counterparts but overcome two of the four limitations .First the first argument need not be a pathname; it can be a command name.Second these functions can also run a shell script.
- int execlp(const char *file,const char *arg0,……/*, (char *)0 */);

- int execvp(const char *file,char *const argv[]);

Other Members of the “l” and “v” series

- Note that pathname has now become file; the other arguments remain the same.To show how execlp works just replace the line containing the execl call in the program execl.c with this one.
- execlp(“wc”, “wc” , “-l” , “foo”, (char *) 0);
- Now the first and second arguments are same .To run the program execv.c that uses execv just change execv to execvp without disturbing the arguments.