UNIX System Calls

System Process

- The only active entities in a UNIX system are the processes.
- UNIX processes are very similar to the classical sequential processes.
- Each process runs a single program and initially has a single thread of control i.e. it has one program counter, which keeps track of the next instruction to be executed.
- UNIX is a multiprogramming system i.e. multiple, independent processes may be running simultaneously.

System Process …

- Each user may have several active processes at once, so on a large system, there may be hundreds or even thousands of processes running.
- In single-user workstations, even when the user is absent, dozens of background processes, called daemons (i.e. self employed evil spirit), are running.
- These daemons are started automatically when the system is booted.

System Process …

- “**Cron Daemon**” is a typical daemon.
- It wakes up once a minute to check if there is any work for it to do.
- If so, it does the work.
- Then it goes back to sleep until it is time for the next check.
- This daemon is needed because it is possible in UNIX to schedule activities minutes, hours, days, or even months in the future.

System Process …

- For example, suppose a user has an appointment with dentist at say 5:00 pm on 11.01.2000, he can make an entry in the cron daemon’s database telling the daemon to beep at him at, say, 4:30 pm on 11.01.2000.
- When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

System Process …

- Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth.
- Daemons are straightforward to implement in UNIX because each one is a separate process, independent of all other processes.

System Process …

- For example, suppose a user has a dentist appointment at 3 o'clock next Tuesday.
- He can make an entry in the cron daemon's database telling the daemon to beep at him at, say, 2:30.
- When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

System Process …

- The cron daemon is also used to start up periodic activities, such as making daily disk backups at say 4 A.M., or reminding forgetful users every year on any particular date or month i.e. say December.

System Process …

- Processes are created in UNIX in an especially simple manner.
- The fork system call creates an exact copy of the original process.
- The forking process is called the parent process.
- The new process is called the child process.
- The parent and child each have their own, private memory images.

System Process …

- If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.
- Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward.
- Changes made to the file by either one will be visible to the other.
- This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file as well.

System Process …

- The memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty:
- How do the processes know which one should run the parent code and which one should run the child code?
- The secret is that the fork system call returns a 0 to the child and a nonzero value, the child's PID (Process Identifier) to the parent Processes are named by their PIDs.
- When a process is created, the parent is given the child's PID.

System Process …

- If the child wants to know its own PID, there is a system call, getpid, that provides it.
- PIDs are used in different of ways.
- For example, when a child terminates, the parent is given the PID of the child that just finished.
- This can be important because a parent may have many children.

System Process …

- Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.
- Processes in UNIX can communicate with each other using a form of message passing.
- It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read.
- These channels are called pipes.

System Process …

- Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available.
- Shell pipelines are implemented with pipes. When the shell sees a line like “sort <f | head”, it creates two processes, sort and head, and sets up a pipe between them in such a way that standard output of sort is connected to that of head standard input.

System Process …

- In this way, all the data that sort writes go directly to head, instead of going to a file. If the pipe fills up, the system stops running sort until head has removed some data from the pipe.
- Processes can also communicate through software interrupts.
- A process can send what is called a signal to another process.

System Process …

- Processes can tell the system what they want to happen when a signal arrives. The choices are
  - to ignore it
  - to catch it
  - to let the signal kill the process (the default for most signals)

System Process …

- If a process elects to catch signals sent to it, it must specify a signal handling procedure.
- When a signal arrives, control will abruptly switch to the handler.
- When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts.
- A process can only send signals to members of its process group, which consists of its parent (and further ancestors), siblings, and children (and further descendants).

System Process ...

- A process may also send a signal to all members of its process group with a single system call.
- Signals are also used for other purposes.
- For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0, it gets a SIG**FPE** (**F**loating-**P**oint **E**xception) signal.
- The signals that are required by **POSIX** [**P**ortable **O**perating **S**ystem **I**nterface] are listed given below **...**

System Process …

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	Alarm clock has gone off
SIGFPE	Floating point error has occurred
SIGHUP	Phone line the process was using has hung up
SIGILL	User has hit the DEL key to interrupt the process
SIGQUIT	User has hit the key requesting the core dump

System Process …

Signal	Cause
SIGKILL	Sent to kill a process
SGPIPE	The process written to a pipe which has no header
SIGSEGV	Process has reference an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application defined purpose
SIGUSR2	Available for application defined purpose

System Process …

System Call	Decription
pid=fork()	Create a child process identical to parent
pid=waitpid(pid,staloc, opts)	Wait for child to terminate
s= execve(name, argv, envp)	Replace the core image of process
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask

System Process …

System Call	Decription
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Unix System Calls

- “**System Calls**” are assembly language instructions.
- The “**System Call**” provide interface between a process and an operating system.
- “**System calls**” are usually made when a process in user mode requires access to a resource.
- The “**System Call**” requests the kernel to provide the resource.

Types of Unix System Calls

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication

System Calls Windows v/s UNIX

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()

System Calls Windows v/s UNIX

Types of System Calls	Windows	Linux
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Types of Unix System Calls: Process Control

- These system calls deal with processes like
  - Process Creation
  - Process Termination
  - …

Types of Unix System Calls: File Management

- These system calls are responsible for file manipulation like
  - Creating File
  - Reading File
  - Writing into File
  - …

Types of Unix System Calls: Device Management

- These system calls are responsible for device manipulation like reading from device buffers, writing into device buffers etc.
  - Reading from Device Buffer
  - Writing into Device Buffer
  - ...

Types of Unix System Calls: Information Maintenance

- These system calls handle information and its transfer between the operating system and the user program.

Types of Unix System Calls: Communication

- These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Example of Unix System Call

- “open()”
- “read()”
- “creat()”
- “write()”
- “lseek()”
- “close()”
- “stat()”

Example of Unix System Call

- “fstat()”
- “dup()”
- “link()”
- “access()”
- “chmod()”
- “chown()”
- “unmask()”

Example of Unix System Call

- “ulink()”
- “exec()”
- “fork()”
- “wait()”

Unix System Calls: “exec()”

- This system call runs an executable file in the context of an already running process.
- It replaces the previous executable file.
- This is known as an overlay.
- The original process identifier remains since a new process is not created but data, heap, stack etc. of the process are replaced by the new process.

Unix System Calls: “exit()”

- The exit() system call is used by a program to terminate its execution.
- In a multithreaded environment, this means that the thread execution is complete.
- The operating system reclaims resources that were used by the process after the exit() system call.

Unix System Calls: “fork()”

- Processes use the fork() system call to create processes that are a copy of themselves.
- This is one of the major methods of process creation in operating systems.
- When a parent process creates a child process and the execution of the parent process is suspended until the child process executes.
- When the child process completes execution, the control is returned back to the parent process.

Unix System Calls: “kill()”

- The “kill() system call” is used by the operating system to send a termination signal to a process that urges the process to exit.
- The “kill system call” does not necessary mean killing the process and can have different meaning(s).

Unix System Calls: “wait()”

- In some systems, a process may wait for another process to complete its execution.
- This happens when a parent process creates a child process and the execution of the parent process is suspended until the child process executes.
- The suspending of the parent process occurs with a wait() system call.
- When the child process completes execution, the control is returned back to the parent process.