× Interrupts handled via inthand, which initiates a context switch & pushes current context layer for resuming later. Handler is then resolved using interrupt source & interrupt vector.

Sample Interrupt Vector (Order of descending priority of interrupts)	Interrupt # (Priority Desc.)	Handler
	0	clock intr.
	1	disk intr.
	2	tty intr.
	3	devintr
	4	soft intr
	5	other intr

× <u>Process Creation</u> —:

- Using fork() : int fork()
  Returns : pid of child to parent, 0 to child.
  < 0 => Error, Failure
  init (PID 1) : not created via fork(), initiated by Process w/ PID 0.
  (PID 0 has no parent).
- Procedure :
  - Allocate slot in process table for new process.
  - Assigns unique ID to child process.
  - Make logical copy of parent's address space for child (leading to separate address spaces in parent & child).
  - Increment file & inode reference counts.
  - Returns PID of child to parent & 0 to child.
- fork() : First a check for available kernel resources is made. Then a unique PID & free process table slot entry is retrieved. (Only when resources are available). For the new process, the state is marked as 'being created'. Next, data is copied from parent process table slot to child slot, and counts of

current dir inode & current root is incremented, followed by increments of counts in global file table. Next, the parent context (u area, text, data, stack) is copied (in memory). Also, a dummy context (copy of parent sys lvl context) (containing data for child identification & execution for running when scheduled). Next in child process init u-area timer fields & return 0, while in parent change child state to 'ready to run' & return child PID.

× Context fields may be shared instead of being duped to preserve memory.

× File table entries are duped (dup), so count is only increased (processes share file table entries).

× Process Termination —

- Using exit() : void exit(int status_code);
- Process enters zombie state : resources relinquished, context dismantled, process table slot preserved (for parent use).
- Status code investigated by parent process.
- exit() : Ignore signals (termination state), If process is the process group leader, send SIGHUP to members & reset process group. Next, close all files, release cwd, croot, free regions, associated memory (freereg), write accounting record, mark state as zombie, assign ppid of children to init (orphan) If children were zombies, send death of child to init. Send death of child signal to parent of process & perform context switch.
- Death of child signal :

Signal Handling —:

× Signal : Notification to process about occurrence of events (also known as software interrupts)

× Features :
  × Processes are unaware of signal occurrence time.
  × Can be sent by one process to another (or itself) or by kernel to process.
  × Can be either synchronous / asynchronous.
    - Synchronous : Delivered to same process which performed action which caused signal.
    - Asynchronous : Sent to another process, handled by a registered handler.

× Sources : H/W :
  Kernel :
  Other Processes :
  User :

× Handling methods :
  1) Ignore the signal (SIG_IGN).
  2) Default signal handler (kernel generated)
  3) User - defined signal handler (:: signal (<signal>, <handler>))

× Signal types :
  - Termination :
    - Death of child
  - Process Induced Exceptions :
    - Access memory outside addr. spc.
    - Write on read - only memory
    - Privileged instruction or h/w errors
  - Unrecoverable Conditions : (during syscall)
    - System resources exhausted during exec.

- Unexpected errors
  - Non existent syscall
  - Write to pipe w/o reader,
  - Illegal leek reference
- Signals in User Mode
  - Send/Recieve Alarm Signals
- Terminal Interaction
  - Suspend, Hangup
  - Interrupt Execution
- Trace Execution

× Signals must be checked when process is moving from kernel to user mode or from preempted to user mode. Also check upon moving from ready to running or going to sleep from running

× Signals checked using 'issig' : returns status whether process received signals it does not ignore.
  - For all signal bits set in signal field of process table entry, map to signal #, & if signal is death of child, free process table entries of zombie children if this signal is to be ignored, else indicate presence of signal (return true). Also for other signals, if process is not ignoring it, return true. If no bit was set, return false.

× Signal Handler Registration : <signal.h>

(*signal (int sig_no, void (* handler) (int)) (int)

sig_no : Signal to handler

handler : Function to use to handle.

handler spls : SIG_IGN (1) : Ignore signal

SIG_DFL (0) : Default Action

SIG_ERR : Return as error.

Common Signals (64+ signals) :

- SIGALRM : Alarm timeout, generated by alarm() API.
- SIGILL : Illegal machine instruction execution.
- SIGINT : Process interruption (DEL, ^C)
- SIGSEGV : Segmentation fault
- SIGTERM : Process termination, EOF (^D) (using 'kill <pid>')
- SIGCHLD : Sent to parent to indicate child termination
- SIGABRT : Process termination, using abort().

Signal Handler Dispatch : via 'psig' :

- First set signal # is retrieved & cleared for future use. If signal is to be ignored, return. Else, if a user handler is given, get virtual addr of signal handler in u-area, clear u-area entry containing addr (one time use) & modify user level context : add a user stack frame to mimic signal catcher being called. Also modify system level context : write addr of signal catcher to PC of user saved register context & return. Also, if the signal was of type of dump core image, dump user level context to a core file. Lastly, invoke exit immediately.

× Kernel tasks during signal handling —:

- Access user-saved register context saved for return.
- Clears signal field in u-area (set to default).
- Create new stack frame, writing values of PC & stack ptr retrieved from user saved register context (allocating spc if reqd).

Process Groups —:

- Group ID used by kernel to identify related processes (recieving common signals)
- Group ID saved in process table
- Create group : grp = ~~gr~~ setpgrp();
- Child retains group of parent.

Killing —:

- kill (pid_t pid, int sig_num)
  - Sends signal (sig_num) to given process referred by pid
  - pid value : > 0 = specific process id
    = 0 = all processes in sender's process group.
    <-1 = process group w/ id = - pid (all processes)
    -1 = all processes w/ real user ID == effective user ID of sender

- nice (int priority)
  priority : low → high priority
  high -> low priority
  adds priority to current priority of processes (niceness)

- Alternative for signal handler registration (Unix V3, V4) :
  int (* sigset (int sig_no, void (*handler)(int)))(int)

- Signal Masking : <signal.h>
  int sigprocmask (int cmd, const sigset_t* new_mask, sigset_t* old_mask);
  Returns : 0 : Success, -1 : Failure
  cmd : Defines use of new mask :
  0 : SIG_SETMASK : Override signal mask w/ new mask
  1 : SIG_BLOCK : Add signals from new mask to process mask
  2 : SIG_UNBLOCK : Remove signals from process mask.

new_mask : Set of signals to set / reset
(NULL => mask unaltered)
old_mask : previously set mask
(NULL => no value returned).

× Clear Signals (All) : int sigemptyset (sigset_t*);
× Add Signals : int sigaddset (sigset_t*) int sig_no);
× Clear Signal (Specific) : int sigdelset (sigset_t*, int sig_no);
× Add Signals (All) : int sigfillset (sigset_t*);
× Check signal : int sigismember (sigset_t*, int sig_no);

× Signals w/ value 1 in mask (set) are ignored by processes.

× View list of signals : `kill -l`

- getpid () : Get process id.
- getppid () : Get parent process id.
- getpgrp () : Get process group #
- getpwuid () : Get user name from /etc/passwd
- getgrgid () : Get user group name
- setpgrp () : Set new process group

× Awaiting Process Termination —:

× Processes can synchronize execution w/ termination of child process via wait :

pid = wait (stat_addr)

pid : process id of completed process
stat_addr : user spc. addr. containing exit status code.

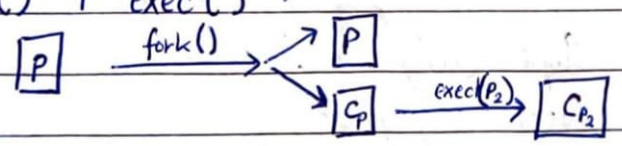
- wait blocks caller until child exits or a signal is received (interruptable sleep state of process).
- Kernel searches for zombie children of processes & returns error if no process is found.
- If a zombie is found, it extracts statistics from the process table & the PID & returns PID in the syscall.
- Also, after retrieval, process table of zombie is freed.

wait : First, throw error when no children to wait for. Next in an infinite loop, check if a zombie child exists & if present, pick a child, add its CPU usage to parent, free its process table entry & return its PID & status code of exit. Otherwise if while waiting no children exist, fail. Else, sleep in interruptible priority state (for signal or child proc. exit).

- <u>Invoking Other Programs</u> — : (exec* family).

- exec : Allows a process to execute a separate program from its parent, by replacing the program image w/ a new program's. (in the same virtual space).

Eg : fork() + exec() :



- execve : execve (<filename>, <argv>, <envp>)
  - filename : file (executable) to load
  - argv : argument vector
  - envp : environment variable pointer (char **)
  - Invokes another program, overlaying memory space of a program process w/ copy of the executable.

× exec behaviors :
- New program loaded in same process space. (No PID change)
- data, code, stack & heap of process are changed & are replaced w/ those of the newly loaded process. (as well as other regions)
- New process executed from entry point.

× exec family : exec, execl, execv, execle, execve, execlp, execvp
- +e : Include envp. (array of ptrs referring env. variables)
- +l : Include argv as a list. (variadic args)
- +v : Include argv as an array. (array of ptrs)
- +p : Include PATH env (search filename via PATH).

Eg: execle (char*, ..., char **envp)
— va-args for argv, type char*. (terminate w/ NULL)

execvp (char*, char**)

exec : Input : filename, argument list, environment variable list.
: First, fetch inode of file to execute (namei) & check for executable permission, especially for the user invoking it. Read the file header (executable header) & check for type being a loadable module (w/ stubs for shared libs). Next, copy exec params to system space & detach all associated regions of child process (detach). (for shared regions). Now for every region in module: allocate new regions (allocreg), attach regions (attachreg) & load region in memory if appropriate (loadreg). Lastly, copy params to new user space region, init user register save area for return to user mode & release inode (iput)

Eg: - Parent PID : # 1500
P,C - fork() : Create child w/ PID # 1501
C - exec() : Replace child process w/ /bin/date date
P - wait() : Wait for child (PID # 1501).

× Rationale for separating text & data sections - :

× Advantageous for protection & sharing.
× Protection : System can prevent processes from overwriting instructions.
(∵ No way to distinguish addresses for instructions & data).
: Separate regions facilitate secure h/w protection mechanisms to prevent processes from overwriting the text space.
× Sharing : Read-only text region facilitates sharing & memory conservation.
(Several processes can execute a file, maintaining separate data).

× Process User IDs :

- Real user ID (Independent of process ID)
- Effective user ID (setuid (set user ID)) (ID of creator)
+ Real UID : User ID of user responsible for current process (running)
+ Effective UID : Assign ownership of files, file permissions.

- Change effective user ID : setuid () (syscall/program)
(Fields set in u-area entry & process table)
Eg: U_A, U_B
P_A, P_B
P_A exec by U_B => EUID : U_A, RUID : U_B
P_B exec by U_A => EUID : U_B, RUID : U_A
- setuid : setuid (<user_id>)
Change EUID of current process.

× More user IDs maintained, controlled via sticky bits.
× setuid() restricted to use by superuser (since euid determines control)
× ~~Access~~ permissions determined by effective user IDs.
(as opposed to real UID, so user given exec perm on binary created by su can execute code w/ su perms over files)

→ set sticky bits : chmod <permgrp> +s
Eg : touch <file>
ls -l // Setuid not allowed
chmod <file> u+s :
ls -l // Setuid allowed

× login process created by user login action (when logging to system)
(setuid called for root).

× <u>Changing Process Size</u> —:

- int brk (endds) : Changes upper limit of data region (increase / decrease process size)
endds : new value of highest virtual addr. of the data region of process (break value).
- void* sbrk (int increment) ;
Lib. fx, changes break value by specified by count.

Returns : 0 (Success), -1 (Failure).

× Kernel checks new proc. size is less than sys max & new data region does not overlap previous virt. addr spc.
× Size changed via growreg.
× If out of memory, process is swapped out for space.