- Preemption require two process — the preempted process will move from running to ready and new will

★ UNIX System Modes —

Running State — Two states
- Executing state for running process.
- Ready state for ready process

- Blocking state is divided in 3 —
  - Uninterruptable — Moves to ready only when the event is finished.
    - Any other signal cannot move to ready.
    - All I/Os
  - Interruptable — Moves to ready when either the event has occurred or there is a signal to move it to ready.

03/09/19
  - Stopped — Moves from running to stopped if a process is waiting for a signal.
    - When signal is received it moves to ready

* Basic states with value–

- Running – 0 0 0 0 0 0 0 0
- Int. – 0 0 0 0 0 0 0 1
- Unint – 0 0 0 0 0 0 1 0

- TASK_MAX_STATE – 0X1000
  - Max^m no. of state.
  - Value cannot be greater than this.

- TASK_TRACED – 0X0008.
  - Used for debugging
  - Used with gdb
  - Gives o/p after each line.

- TASK_PARKED – 0X 0400 0040
  - Int. stopped where other process/core can wake the process.
  - Process not going to be used for a long time.
- Zombie and dead are both for terminated processes.
  - Process becomes dead after Zombie.
  - When parent collects the data of child then zombie becomes dead.

- If we dont take the data of terminated process then the process stays at zombie indefinitely.
- A User can create multiple process so if a user it create infinite process then it will affect other users.
- So there is a limit to no. of process by a user

— Each zombie takes a process count so the user can have 1 less process to exec so it's better to kill a zombie.

- TASK_WAKEKILL – Uninterrupted process but only specific kill signal can wake it but not any other signal.

- TASK_WAKING – When any core/processor wakes a process but the process is still not ready – it's in an intermediate waking state.

- TASK_NOLOAD – No task loaded/executed by the system.
  – A proc will run without being accounted by the system.

P1	P2
Reg1 ← load(X)	Reg1 ← load(X)
Reg1 ← Reg1 + 1	Reg1 ← Reg1 + 1
store(X) ← Reg1	store(X) ← Reg1.

→ Can result in race condition if either process is switched before they are completed so the variable value in not updated.
– X can have value 9, 10, 11 in diff. scenarios.
– This can happen with a process also so when 1 process is woke woken up by some core and another core is trying to use the process resources before it's ready.

- to the process is kept at waiting state to let the core know that process is not ready.

- To prevent race condition interrupts can be blocked which will prevent time slice expiry because clock cannot raise an int.

* b) - Mode switch takes place
  - Some process switches from user space to kernel space.
  - Process can run in both the spaces.

c) - Context switch
  - kernel also running as a process.
  - Better than (b).

* Process Creation-

- fork - Child process of both parent process.

- Memory - Denied in parent - child.
  - Memory table - keeps the info of memory frames occupied by all the processes.
* I/O Table - keeps info about the i/o resources.
- Devices -
- Resource - i) Req - Req by process to OS.
  OS gives the resource.
  ii) use - Process use the resource
  iii) release - Process release the resource after the use.
  - OS checks it.

* File Table – keeps info about the file opened by any process/es.
  – Per process file table.
  – System wide file table.

* Process Table – Info of all the process including zombie.

1/09/19 Process Control Block :–

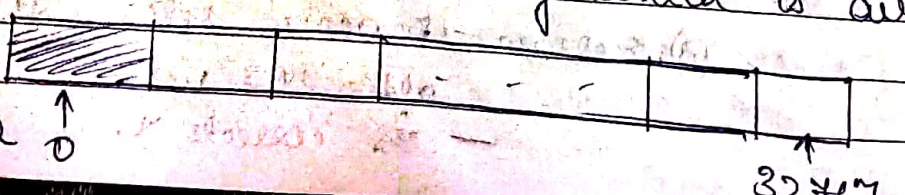
– Even if the process is in swapped state it will have its process control block available in main memory.

– struct task_struct {
  '}' – 600 lines of code.

– Process control block is an instance of this structure.

– An array stores all the process control block
– Pid – The entry in the array at which the process control block is stored for a process.
– Unique pid for each process.
– pid :– 0 – 32767.

* Process Identification –
• PID – ID in the array which is available first



opened by

increasing

ed state it
available in

of this

al block
which
eed for

lable first

- Circular search - starts from the place where it was allocated last.
- unsigned number. (int).
- Pid 0 - ~~INIT or~~ sched process.
  - To shutdown a OS we give command INIT 0 which will kill the first process.
- Pid 1 - INIT process.
- ppid - Parent process id.
  - Parent's id is also maintained.

- get pid()/ ppid() - Prints the pid/ ppid no.
- To send a signal to a process it's pid has to be mentioned / used in the inst.

- gid - Group ID.
  - In any group if pid = gid then that process is the leader of the group.

- sid - Session ID.
  - When we login we create a session and any process created in that session will have be under the same session.

- Pid and ppid are compulsory id.
- Pid and ppid cannot be set by user. Other ids can be changed using setgid() / setsid().

- If a Parent Process dies before the child then -
  i). either grandparent will become the parent.
  ii). Or INIT process will become the parent.
- In most OS INIT becomes the parent.

- A parent process maintains a link to its first child and last child.
- Each child maintains a link with its previous sibling and next sibling.

* Processor State Information –

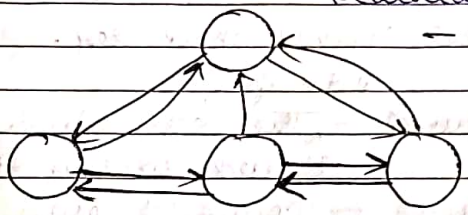
- Stores the info about the processor the process worked last on, i.e. the previous register values, prog. counter, flags value, stack pointer, etc.
- This is backed up whenever the process is moved to blocked / stopped state due to time slice expiry, I/O or event wait.
- This is stored in the process control block.

* Process State Information –

- Argument Vector – Array of 2D char array or 1D string array.
  - Used to store the arguments in command line.
  - Argv[0] – Contains the executable name of the prog.
- Environment Vector – Path Variable / Env. Variable which tells which place the prog should search for if it needs a file.
- Most of the variables are inherited from the parent but it can be modified by child which will not be reflected in parent.

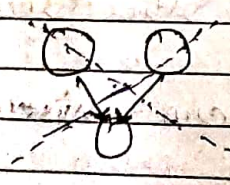
- Any modification done by parent after the creation of child will not be reflected in child process.

- Scheduling and state information —
  - Process State
  - Priority
  - Scheduling algo. — Linux works with a no. of scheduling algo. — class of Algo.
- Data structure — linking with other processes.
  - Maintained as a tree.
  - Parent.



  - Child.

- If there is no link b/w the children then parent becomes the left/right sibling of that child.
- So whenever a child process dies the parent has to collect the info and modify the links of its siblings.



- A child cannot have multiple parent.

- Orphan Process — When the parent dies the child process becomes orphan process and join INIT process as the parent.

- This will increase the breadth the of the process so killing a parent process is not good.

- Process Privileges.
- Inter Process Communication
- Mem. management
- Resource Ownership & utilization

- User Running Time (utime) and kernel running time (stime) is maintained separately for each process which tells the total running time in user space and kernel space resp.

- ~~cpus~~ cpus_allowed - bitmap of The cores which can are allowed to run the process.
- nr_cpu_allowed - Count of cores allowed.

- Total Running Time = utime + stime.

- nvcsw - No. of voluntarily context switching - By the process.

nivcsw - No of involuntary context switching - By the OS or some other process.

Total Context Switching = nvcsw + nivcsw.

06/09/19 * ~~User Stack~~ Private User Area -

- Private user area is global area for all the function of the process.

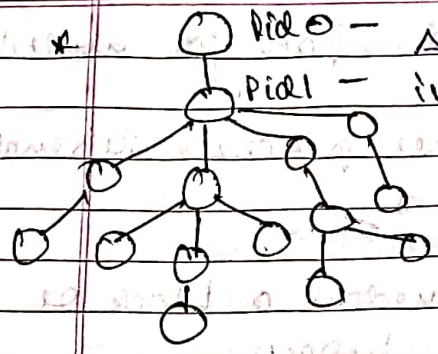
* Shared Addr. space – shared space for multiple processes.
– Accessing this area by other processes depends on the permission

* User stack – Variables defined in a block or function in a process.
– Local variables/data.

★ Process Management :–

• fork () – Creates new process.
– Only function that returns two values
– Return values are not returned to the same processes.
– Returns one value to the parent i.e. child's Pid to the parent.
– Returns 0 to the child always.
– If return value is -ve then child not created. In this case there will be only 1 return value to the parent and not to the child because child not created.

– x = fork ();
– x will be there in both parent and child process.
– x value in parent process is the pid of child process.
– x value in child process is 0.

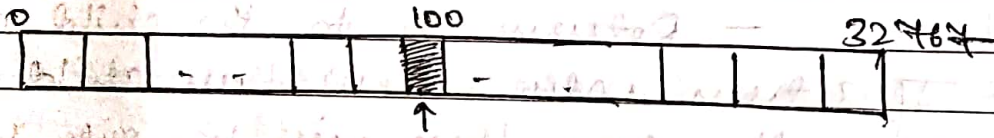
★ 

} Fixed.

- Tree like structure because child can have only one parent.
- Not graph like structure.

(sched)
- INIT 0 - kills the top process ↑ - shutdowns the process.
- If parent dies child usually joins to init process (Pid 1). Child can also join the grandparent depending upon the os.
- sched - Maintains scheduling and swapping.

★ Process Creation -



- The search for an empty pid starts from the last allocated pid and it is a circular array. So child can get a lower pid than parent.
- When a parent opens a file and after opening the file that parent creates a child process then child will also have that file opened in its block and the count of file increases by 1. - All resource sharing.
- Limited Resource sharing
- No Resource sharing.

study time
Page No.:
Date : / /

structure because
have only one
e structure.
ed)
- shutdowns
ns to init
- join the
e os.
swapping.
32767
]
rom the
circular
r than
d opening
process
opened
file increm

study time
Page No.:
Date : / /

- In linux for unlimited resources all file resource sharing is followed by parent and child.
- If parent opens a file after the child is created will not be opened in child.
- If child opens a file then it will not be opened in parent.
- After the creation of child, parent and child are ind. process, any changes made in child and parent will not reflect in each other.
- Child will have a copy of variables/stack/data of parent process ~~during~~ the first time it is created but the modification is not reflected in each other because the actual variable is not shared - it is just copied.
- Similarly with file pointer.
- This happens for the global variable of the prog. also.

* Execution — Two ways to create child — fork() & vfork().

- vfork() — Suspends the parent until child finishes it's process and terminate.
  - Depricated sys. call — Not used now.
  - Parent then can only create 1 child in this ~~process~~ case.
- fork() — Child and process execute concurrently i.e. time sharing is possible.
  - Both process joins the ready queue - child joins first and then parent.
  - Both the process can execute parallely i.e. at the same time if h/w supports it (multicore/multiprocessor).

- The variables are not shared b/w child and parent. Both have diff. copies of all the variables including global variables of the prog.

- Child is created in a separate memory space than parent. So any modification done by either of them will not be reflected in each other.

- Data block and stack block is replicated for the child but code block is not replicated because code is read only so both child and parent follow the same code.
- Both parent and child will continue execution from the next statement of fork().

```
- int g = 100
int main()
{
  int var1, x = 0;
  var1 = fork();
  if (var1 > 0)
    { printf("I am parent\n");
      x = 10; g = 1000;
      wait(NULL);
    }
  else if (var1 == 0)
    { printf("I am child\n");
      x = 200; g = 2000; }
  else
    { printf("Error\n"); }
  return 0;
}
```

(P0) —————— (P1)

Pid = 2000
g = ~~100~~ 1000
n = ~~0~~ 10
var1 = 2100

Pid = 2100
g = ~~100~~ 2000
n = ~~0~~ ~~100~~ 200
var1 = 0

~~– Each process must have to allocate some space to kernel ~~ block.~~

– Each process ~~has~~ is given some address space in the kernel space. If kernel space is full then no more process can be created.

09/09/19 Fork failed reasons –

- Not enough resources available.
- No permission to create more processes or max. no. of process reached.

– fork returns -ve no. to parent process if child is not created in case of an error.

– wait (NULL) – Written in parent process code.
 – wait for a child process to complete and ~~then come out~~ move to next statement in parent process.
– For multiple children ~~so~~ that many wait () has to be written in parent.
– Instead of NULL an address can be passed to wait () which store the status of child process.
– When the parent reaches the wait () statement and executes it, it moves to the blocked state

wait until the child has completed execution.

- On creation of child both the parent and child will continue the same program from the next statement of fork(). Parent will not go to blocked state.

- Because of wait() in parent code the child will always terminate first and then the parent. This avoids the creation of orphan prog.

- child will join the ready queue before parent but execution depends on the scheduling so parent may or may not execute before the child process.

- ~~If wait() is written outside the parent block then it~~

- If wait() is in a process with no child then the process will not wait for any process to end.

- wait() will wait for any of the n child process to complete. Can be any child which is terminated first.

- waitpid() will wait for the child with specified pid to complete.

- If we remove wait() from parent's block, the parent process may terminate before the child process which will make the child

completed exe.

he parent and
e program/code
ork ().Parent
ate.

code the child
d then the
of orphan prog.

before parent
scheduling so
& before the

the point

s no child then
y process to

e n
child proces
e is termina-

with specific

's block,
for the
child

process an orphan process and it will join INIT as it's parent process. — Bad prog., Avoid it.

* wait (& variable).
— We are passing the add. of variable so the child process (exit) will return a value to that add.

— The return value of wait() is the pid of the child process which completed execution.

— The last return in a process acts as exit().

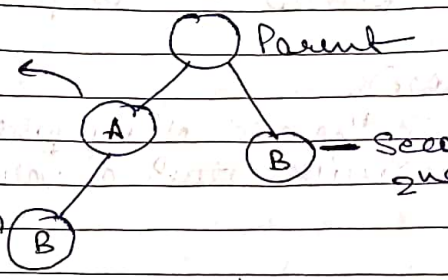
* exit (return/error no.) — In child we write exit() with a diff error no. at diff. blocks so that we can know from where the child prog. exited.
— This no. is stored in the variable add. passed as an argument in the wait() statement of parent process.

• waitpid (pid, & status, 0)
— The parent waits for the child with the that specific pid to terminate.

— If pid = -1 the waitpid() behaves exactly as wait() i.e. it will wait until any of the children has terminated.

Page No.: study time
Date : / /

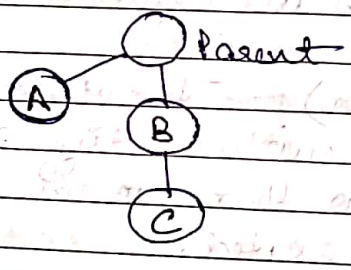
* Process Creation Example :-



* first fork() - 1st child of parent
* Second fork() for parent - 2nd child of parent
* Second fork of proj - 1st fork for the child process - child of the 1st child of parent.

fork(); A fork(); B	- Only two wait() req. bcz the parent can have atmost 2 children process.

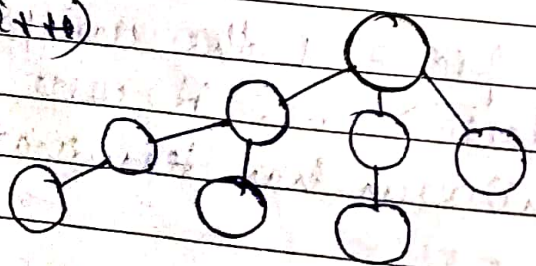
10/09/19 *




```
if(fork()) — A
    if(!fork()) — B
        fork(); — C
```

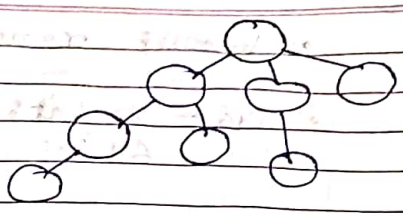
- Max^m two wait() required.

```
* for(i=0; i<3; i++)
      fork();
while(wait(NULL)!
      = -1);
```



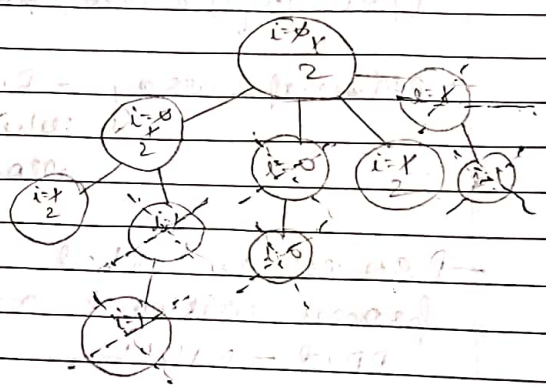
int x = 0;

* for (i=0; i<3; i++)
{ fork();
x = x + 5;
}

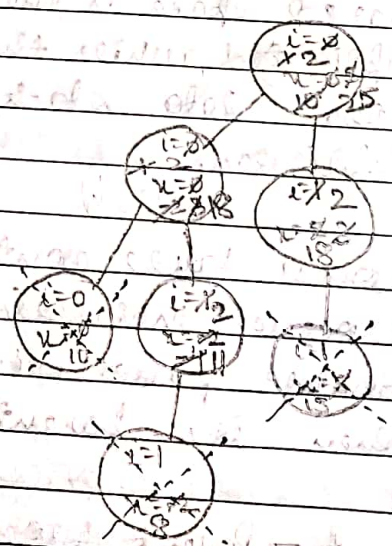


- All process will have the same value of x = 15 at the end of the loop.

* for (i=0; i<2; i++)
{ if (fork())
{ if (!fork())
{ fork();
exit(0);
}
}
}



* x = 0;
for (i=0; i<2; i++)
{ if (fork())
{ x = x + 5; }
else if (!fork())
{ x = x + 10;
exit(0); }
else { x = x - 2; }
}
x = x + 15;



11/09/19.

- Smallest execution unit is a process
- Pid - 32 bit: 2 - 32767
  64 bit: 2 - 4194303

- Process control block is an array of structures which keep the data of each process.
- Size can be configured manually - By default 32767.
  (/Proc/sys/kernel/pid_max)

- Pidmap - array - Bit map array to determine which PIDs are already allocated.

- Process 0 (sched - swapper process) is the only process without a ~~parent~~ parent.
  PPid - NULL.

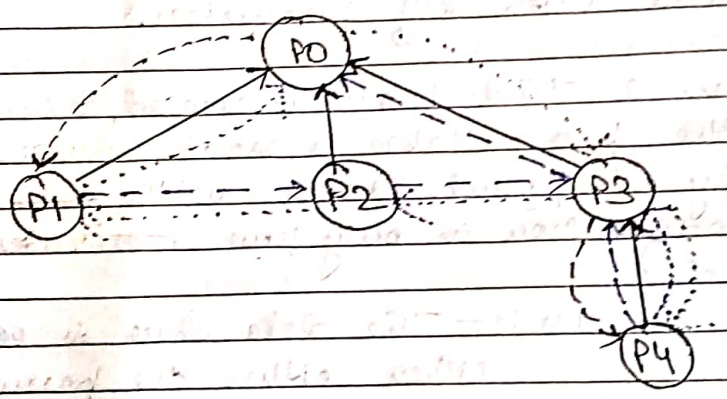
- Shared space is the only space which is not replicated while the creating a child.
- Global data, stack, other variables etc. are replicated.

- Process table contains info about all the processes including the Zombie process.
- Zombie process is removed from the table when it's parent collect the data.

* Parent - Parent process that has can monitor any child process of some other Real-Parent process.

— Used when while debugging — ptrace().
— In ptrace() the child will not respond to Real Parent but to the monitoring Parent.

* Real_Parent — That process which gave birth to the child.
— usually real_parent and parent are the same process.
— Only in exceptional cases like tracing real_parent process is diff. from parent process.



• Children Next — First child created (----→)
Children Previous - Last child created (.....→)

• Sibling Next — Next sibling. (----→)
— For last child the next sibling is the parent process.
Sibling Previous — Previous sibling. (.....→)
— For first child the previous sibling is the parent process.

- Whenever a children is killed then all these linkages have to be modified acc. so only OS can terminate the a process to kill all the because it has to modify all the linkages.

* fork.c - File which contains the execution of fork(). (Linux).

- do_fork() is executed when you call a fork() fn function or when you create a thread with diff. parameters.

- When a child process is created, a new entry in the process table is created which is the copy of parent process with ~~only few~~ modification in only few attributes.

- Copy on Write - The data space is copied only when either the parent or child writes for the first time.
- If either processes doesn't write anything then they can use the same data space.
- Code is read only so no copy is made for child. Both the processes use the same code from the same space.

Eg:- execl("/bin/ls", "-l", "\0")
- The process which execute the command will die so this has to be executed as a child process.

13/09

led then all
odified acc. so
a process to
es to modify all

e execution of

you call a
ut create a

new entry
which is the
only few modi
nts.

copied only
ent or child

anything then
.

for child.
de from

he command
executed

— So, in this case the data will not be copied because it will only display data and not modify anything.

- Demand Paging / Segmentation —
  Until a block of data is req. by the process it is not loaded in the main mem.

- meth.n — Dynamically linking lib.
  — linked with the prog. when it is used for the first time while execution.

13/09/19 Usually a child process runs on the same core as the parent because the code is already present in the cache - Locality of Reference.

* V fork() — Parent process will be in suspended state until the child completes it's execution.

— New copy of data space is not created and child uses the same space as parent.
— Any changes done by child is reflected to parent also.
and code space
— Shared mem. never gets replicated.
— Global variable gets replicated because global variables are specific to any process when it is shared b/w the various functions but not shared b/w diff. processes.

— Parent gets the return value first and then the child.

Page No.: study time
Date : / /

chance to
- Parent moves to ready queue to give the child the child it to execute but the scheduler does allow a newly created child to execute because so many other processes are running so child also doesn't get executed.
- So both the processes move to the ready queue.

- Vfork() is used when child dies immediately using exec() system call.

- In kernel space 8 kB of mem. must be present to store Process Control Block and kernel mode stack.
- If the space is not available then it will be fork failure.

- Flag updates :-

- Clears PF_SUPERPRIV (super user privileges are not given to child). - Child has to get the privileges on its own.
- Clears PF_USEDFPU
- Int unit pipeline is diff from floating pt unit because if they are in same pipeline all the int. calculations will become floating pt calculation which will take more time.
- So Floating Pt. unit is turned off for child.

- Clears the trace flags.
- Sets the NO_EXEC flag i.e. the child will

16/09/19

Scanned by CamScanner

chance to
to give the child
scheduler doesn't
to execute becau
running so child
to the ready

olies immediat
call.

must be
ee Block
will
hen it

ileges are
the privi

ting pt will
all the
eating pt
ure time.
for

- will

not execute exec() function.
- sets/resets Vfork flag according to the calling inst. (fork() or vfork()).

16/09/19 — Wed

* New Resource allotment to child —

- user can set the mask for signal handler to identify which signal handler to follow.

• Process list — contains the pid of only the running processes in the system. (Process table)

• hash_pid() — To search and insert new process entry in pidhash table.

• nr_tasks — Total no. of tasks in the system (all users combined).

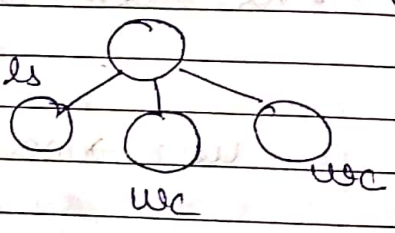
• user → count — Total no. of process only for that user.

* vfork() — Used heavily when fork() did not implemented COW (copy on write).

- It does not make a copy of addr. space and page table.
- Child usually executes exec() / exit() command.
- Now fork() implements copy on write so the only diff. b/w current fork() and vfork() is that page table is replicated by fork() but not by vfork().

- vfork() is dangerous because any changes done by child is reflected to the parent also.
- Main memory is divided into multiple pages. So Page Table keeps track of all pages used by the process.
- The child can only access its own page table so new page table is created for child.
- The page table is replicated when child is created. When first time any data is changed by either child or parent a new block of data is created for the child and it's page table is updated accordingly.

* ls | wc | wc



- All the child process will die immediately after it's execution.
- It's better to execute this command with exec than in command line.

* pid_t wait(int *status);
- Returns the pid of child process that terminated (if successful)
- Stores the exit status of child in the pointer passed as argument.

pid_t waitpid (int pid, int * status, int options)

* waitpid :-

- wait for process with particular pid to complete execution.

pid > 0 – child with pid that is passed.
pid = 0 – child with group pid same as that of calling process.
pid = -1 – child with any pid.
pid < -1 – child with group id is same as the absolute value of what is passed.

- WNOHANG – Return immediately if no child has exited.
- WUNTRACED – Return for children which are stopped but whose status has not been reported.
- WCONTINUED – Returns if a stopped child (by using SIGSTOP) has been resumed by the delivery SIGCONT.
- Return value – pid of child which is exited.
  – 0 if WNOHANG and no child was available.
  – -1 in case of error

14/09/19

★ execl() family :-

- execl() - long representation
  execv() - vectored rep.

- Both of them is executed as execl only.
- The process which executes it dies so they used in child process and not a parent process.

- execl() returns a value only if it is a failure. Returns only -1.

- execl(); - Process dies at this inst.
  printf(); - Goes to this command only if execl() fails

★ execlp(); - Does not req. to give a path
- Path is taken from the path variable of env. variable.
- First arg is the file name.

★ execle() - Req. to give the path of env. variable as the last argument
- Path is req. as 1st argument

★ execv() - Arguments are stored in a 2-D array and the pointer is passed as a parameter.

- In execl() first argument is the path
- In execle() the first arg. is the path because it is not necessary that the env variable has the path.

- Last argument of any of the list is NULL.
- /bin : /sbin : ... - multiple path for execution.

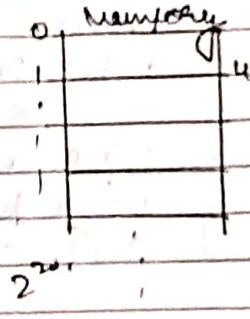
~~★ Multi-Tasking in mobile systems :-~~

★ When to switch a Process :-

- Processor gives logical add. and not physical add. So to access cache if mem. access is req. then it will take so much time to access cache also.

Page No.	Frame No.
22	20.

- Translation look aside Buffer.



34 / LA 22 / PA

- 4k $= 2^{12}$ bits.
- 2^{32} = 4 GB. total memory.

∴ No. of frames $= \frac{4GB}{4KB} = \frac{2^{32}}{2^{12}} = 2^{20}$ bits.

- Sector add i.e. last 12 bits will be same in both LA & PA. (Offset)
- So the remaining 32 bits is given/stored as i/p to TLB and the 20 bits of physical add. is stored/given as o/p from TLB.

- Appending it with 12 bit sector add we get the 32 bit complete physical add.

- 99% of the time PA is taken from TLB.
- Logical Add. always starts from 0.
- In case of context switch the logical add. now corresponds to some other process so TLB is flushed.
- Note there will cache miss every time process and TLB doesn't have any page information so it req. it from main memory.
- In context switching the reg. data of previous process is stored and in PCB and new data of new process is loaded from PCB - Fixed Cost. - Fixed Direct Impact.
- TLB is also flushed and new page data for new process is loaded at and when - Variable Cost. - Indirect Impact
- Virtual Add. is for the swap space available in HDD and not the main memory.
- So max size of proj. that can run is of the size of main memory + swap space.
- But at a time not all the data can be loaded at in the main mem. so some data is stored in swap space which is switched from swap to main memory and the page data in TLB is updated accordingly.

18/09/19

um from TLB.
. 0.
ue logical add.
ee process so TLB
every time because
nformation so
ry.
data of previous
d in PCB and
loaded from
- Impact.
page data for
when - Variable
e available
mory.
lumn is of
space.
ata can be
so some
ch is
memory
updated

* Clock Int. – Context switching happens due to time slice expiry.
– To stop it we can disable the clock int. of the process.

* I/O Int – Int. from an I/O or when a process goes for I/O op.

18/09/19 Direct Impact – Fixed cost in context switching.
– For Reg. data.
– Depends on Architecture.

* Indirect Impact – Because of TLB and caches.
– Variable cost.
– Dominant impact – 80%.

– So Quantum time should be decent enough to avoid too much context switching.
– Preemption time is unproductive work and the time is wasted in context switch.

* Memory Fault – Not segmentation fault.
– When a page is not available in main memory so it has to be requested from virtual space.

** – In intel min. 6 pages should be present in main memory to run a process.
– All the pages are loaded on demand.
– Every block in cache/main memory has a valid/invalid

bit. To flush cache/memory we make all the bits as invalid.

- Each process has a Page Table which keeps track of all the pages which has are available in main memory and which are available in HDD.

- Os can modify the page table ~~any~~ of any process so while loading a page for some process only os can set the bit of previous process as invalid, load the new page and set the bit so as valid for new process.

★ Process Termination :-

- exit() - stdlib.h - Header file.
  - includes malloc(), calloc(), etc.

- A parent can kill it's child using SIGKILL but vice versa not possible.

- void exit (int status).
  - Handled by do-exit().
  - return 0 is also calling exit function.

- PF_EXITING - Process flag to denote process is existing.