★ Parents & children.

→ Just as a file has a parent every process also has one.

→ This parents itself is another process & a process born from it is said to be its child.

→ When you run the command cat emp.lst from the keyboard a process representing the cat command is started by the shell process.

→ Since every process has a parent you can't have orphaned process.

★ Wait or not wait

→ Two different attitudes that can be taken by the parent towards its child.

① It may wait for the child to die so that it can spawn the next process. The death is informed by the kernel to the parent.

When you execute a command from

the shell the shell process waits for the command to die before it return the prompt to take a next command.

(2) It may not wait for the child to die at all & may continue to spawn other processes.

→ This is ~~what~~ what the init process does.

★ How the shell is created.

When a user attempts to login getty wakes up ~~the~~ & forks exec ~~forks exec~~ the login program to verify the login name & password enter ~~&~~ on successful login, login forks exec the process representing the ~~login~~ logic shell.

init ⟹ (fork) getty ⟹ (fork-exec) login ⟹ (fork-exec) shell

→ The shell as can be seen from this sequence.

→ inits goes off to sleep, waiting for the death of its children. The other processes getty & login had extinguish themself by overlap when the user logs out the shell is killed & the death is intimated to init.

→ Inits then wakes up & spawns another getty for that line to monitor the next login.

⟹ Getty is unix program running on a host computer that manages physical or virtual terminals.

→ When init detects a connection it prompts for a username & runs the login program to authenticate the user.

★ Internal & External commands.

→ From the process point of view shell recognizes three types of commands.

① External commands.

→ The shell creates a process for each of this commands that it executes while

remaining their parent e.g cat command.

② Shell scripts.

→ The shell executes this scripts by spawning another shell which then executes the commands listed in the script.

→ The child shell becomes the parent of the commands that featured in the script.

③ Internal commands.

The shell has a number of built in commands like cd & echo ~~both~~ don't generate a process. and our executed directory by the shell.

Similarly variable assignment of the Statement $x=5$ for instance doesn't generate a process either.

✱ Process states & Zombies.

→ A process after creation is in the runnable state before it ~~acc~~ actually runs.

While the process is running it may be invoke a disk i/o operation when it has nothing to do except wait for the i/o to complete.

→ The process then moves to the sleeping state to be woken up when the i/o operation is over.

→ Process can also be suspended by a key.

→ Processes whose parents don't wait for that their death moves to be zombies state.

→ When the process dies it ~~e~~immediately moves to the zombie state.

→ It remains in the state until the parent picks up the child ~~exceed~~ exit status from the process table. ~~ent~~ entry.

★ Running jobs in Background.

→ Unix provides the facility for backgro-und processing i.e when one process is running in the foreground, the another process can be executed in the background.

→ The & symbol placed at the end of a command sends the command for background processing.

```
$ sort emp.doc &
        PID   for background.
550
^d
```

range of pid = 0 to 32767.

1) → nohup : log out safely.

→ If a user wants a process that he has executed not to die even when he logged out from the system you can used this command.

nohup (no hangup).

syntax

$nohup <command> &

e.g $nohup sort emp.doc &

551

↳ output on nohup.out

2) $at

This command is used to execute the specified unix command at future time

syntax:-

at <time>
<commands>
^d. (control d)

e.g $ at 12:00
echo "LUNCH BREAK"
^d

* Keywords.

1) now
2) noon
3) midnight
4) today
5) tomorrow
6) hours
7) days
8) weeks
9) months
10) years

e.g $at 12:00 + 1 day } Gives remainder tomorrow on 12:00
echo "LUNCH BREAK"

3) atq

→ atq command is used to list the jobs submitted by you at queue.

→ This command list the job no., schedu-led date of execution.

e.g $atq

Job no.	date	hours	queue	username
7	2002-12-16	12:00	a	bmi
8	2002-12-17	12:00	a	bmi
9	2003-01-22	13:00	a	bmi

4) $atrm is used to remove job from the at queue.

$ atrm <jobnumber>

e.g $ atrm 8

O/p

7 & 9 /of
Job no. 7 & 9 is display-ed.

5) * $batch

→ This command is used to execute the specified command when the system load permits (when CPU becomes nearly free).

Syntax
```
$ batch
<commands>
^d
```

e.g
```
$ batch
a.sort
b.sort
^d.
```

6) $Kill.

→ If you want a command to terminate prematurely press ctrl d.

→ This type of interrupt characters does not affecting background processes because the background processes are protected by the shell from these interrupt signals.

→ This kill command is used to terminate the background process.

→ Syntax $ kill [-signal Number] <PID>

e.g $ kill 551

→ By default this kill command uses the signal number 15 to terminate a process but some programs like login shell simply ignore this signal of interruption & continue execution normally.

In this case you can use the signal number 9. Often referred as sure kill.

(-9 for login process kill).

Syllabus

9.1 Process basics
- 9.1.1 The shell process
- 9.1.2 Parents & children
- 9.1.3 Wait or not wait

9.2 PS (process status)
- 9.2.1 PS options.

9.3 System processes.

9.4 Mechanism of process creation
9.4.1 How the shell is created.
&
9.5 Internal & external command

9.6 Process states & zombies

9.7 Running jobs in background.
9.7.1 &: No loggin out
9.7.2 nohup: log out safely

- 9.8 Nice: job execution with low priority

9.9 Killing processes with signals
9.9.1 Kill: Premature termination of process.

9.11 at & batch: execute later.
9.11.1 at: one time execution
9.11.2 batch: execute in batch queue.

7) Nice $nice (reduce the priority).

→ Unix offers this command which is used with the (&) operator to reduce the priority of jobs.

→ A higher nice value implies a lower priority.

→ Nice reduces the priority of any process thereby rising its Nice value.

→ We can also specified the Nice value explicitly with -n option.

e.g $ nice wc -l uxmanual &

e.g $ nice -n 5 wc -l uxmanual &
(1-19 range.)

(More priority more delay)