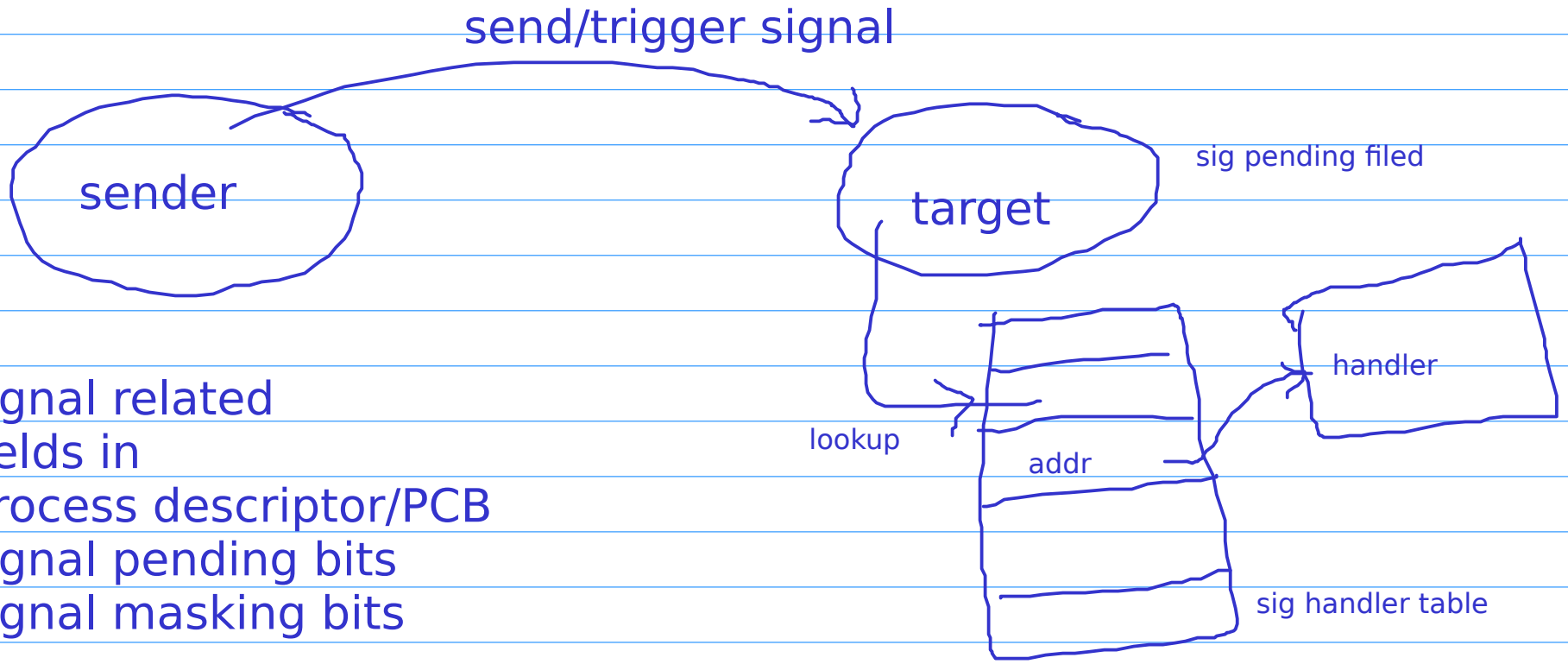
POSIX

waitpid - 3 params
1st param : pid of child
2nd param : exit status (by address, fetch param)
3rd param : flags

execl, execlp, execv

Signals:-



signal related
fields in
process descriptor/PCB
signal pending bits
signal masking bits

when signal reaches target process, corresponding
pending bit will be set

Most of default signal handlers will cause
abnormal termination (typically)

ctrl + C	==>	SIGINT
ctrl + \	==>	SIGQUIT
ctrl + Z	==>	SIGTSTP
kill <pid>	==>	SIGTERM
child exit	==>	SIGCHLD
fg/bg	==>	SIGCONT
floating point	==>	SIGFPE
seg fault	==>	SIGSEGV

other:- SIGKILL, SIGSTOP, SIGPIPE, SIGALRM

kill -l

kill -SIGxxx <pid>
kill -<signo> <pid>

signal handling -- custom handler

Simple APIs:- signal, raise, pause, kill, alarm (Some are deprecated)
Modern:- sigaction, sigprocmask, sigsuspend

Non Maskable Signals:- SIGKILL, SIGSTOP (No custom handlers)

kill <pid>
kill -9 <pid>

kill(pid,signo);
kill(0, signo); //signal to self, like "raise"

Further explore:-
- sigprocmask
- sigalrm
- sigaction
- sigsusped

killall, pkill, pgrep
fg, bg, jobs
command &

return 0 ==> exit(0) ==> _exit(0)
lib call wrapper for exit system call

What's diff b'n exit and _exit

```
for(i=1;i<=10;i++)
   printf("hello:%d\t",i);          // no \n
exit / _exit  (or) any exception before exit
```

Threads:-
* part of application, corresponds sub activity
* light weight process (part of process??)
* flow of control

Significant:-
* Concurrent execution (multiplexing of CPU)
* Resource sharing

* every process also runs as single thread initially

resource usage point of view ==> process
execution point of view ==> threads

multithreaded application
thread creation is lighter/faster than fork (LWP)

Need for threading/examples:-
* Office Suite
* Media Player
* Browser
* Concurrent server -- each thread for every client

task driven parallelism
data driven parallelism, e.g. sum of large array

some threads are CPU centric (no blocking),
some are I/O centric (freq block)

any access other CPU & memory ==> blocking call

threads will share all resources, except stack
every thread will maintain private stack

stack & context (PC) will vary from thread to thread

Self Study:-

Thread models
- user level threads (many to one mapping)
- kernel supported threads (one to one mapping)

Thread pools
Thread cancellation policies
Thread contention scope
CPU Affinity

---

ps -e -L -o pid,ppid,lwp,nlwp,cmd

POSIX Thread Library - pthread APIs
- pthread_create (based on clone system call)
- pthread_join
- pthread_self
- pthread_equal
- pthread_yield

- pthread_cancel

```
void* task_body(void* pv) {
   printf("A--welcome\n");
   //for loop
}
```

psample.c

```
pthread_t pt1;
pthread_create(&pt1, NULL, task_body, NULL);
//....
pthread_join(pt1,NULL);
```

pthread_create params:

1st : addr of pthread_t variable
2nd : attributes, NULL means default
3rd : entry point for execution
4th : arguments to entry function

```
gcc psample.c  -lpthread
-lpthread        ==> libpthread.a    libpthread.so
```

pthread_exit ==> current thread only, resources are not freed up
exit ==> process exit, resources are released
==> all threads will be terminated

Scheduling:-

Self explore:-

* Scheduling params - CPU utilization, throughput
  Turnaround time
  scheduling latency/delay/jittter
  response team
* Fairness among processes/threads
* No Starvation
* CPU Bound vs I/O Bound processes
* Preemptive vs Non preemptive scheduling
* Classical Algorithms
    * First Come Fisrt Serves (FCFS/FIFO)
    * Priority based scheduling
    * Round Robin
    * Shortest Job First (** Theoritical, not practical)
* Combination of algorithms, e.g. FIFO + RR, FIFO + RR
* Multi level scheduling, Multi level feedback queue scheduling
* Scheduling in SMP

---

* Practical scheduling in Linux (very imp)

Who will be in ready state (ready queue)?

Real time -- deterministic
hard realtime vs soft realtime

Scheduling in Linux:-
* Realtime policies
  * SCHED_FIFO (PRIO + FIFO), 100 levels of priority
  * SCHED_RR (PRIO + RR), 100 levels of priority
* Nonrealtime policies
  * SCHED_OTHER : Timesharing algorithm

Timesharing policy:
  * Modified round robin algos
  * 40 priority levels, controlled by nice vals
  * Time quantum is proportional to priority
  * Process may move among priority levels
    * by explicit command / system calls
    * based on CPU usage or waiting time [e.g. aging technique]
  * There are two queues in each priority levels
    * Active & Expired
    * When active queue is empty, swap the queues


```
ps -e -o pid,ppid,stat,policy,pri,ni,cmd
```

policy - TS, priority - 19, nice - 0

```
nice -n 5 ./a.out          # bc or ping
nice -n -6 ./a.out         # sudo
```

nice	prio	nice value range: -20 to +19
19	0	pri scale : 0 to 39
5	14	
0	19	
-6	25	
-20	39	

Activity:-
* Assignment
* IPC Pre-reading (Semaphores, Mutex, Message Queues,
Shared Mem, Pipe/Fifos)
* Virtual Memory (videos)