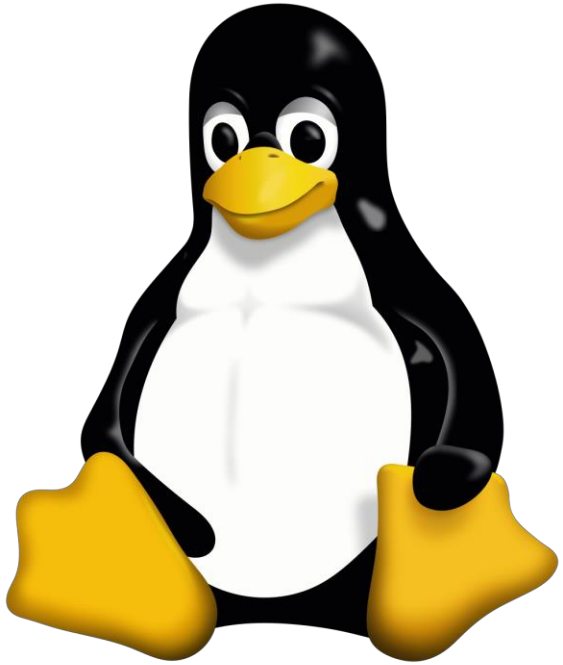
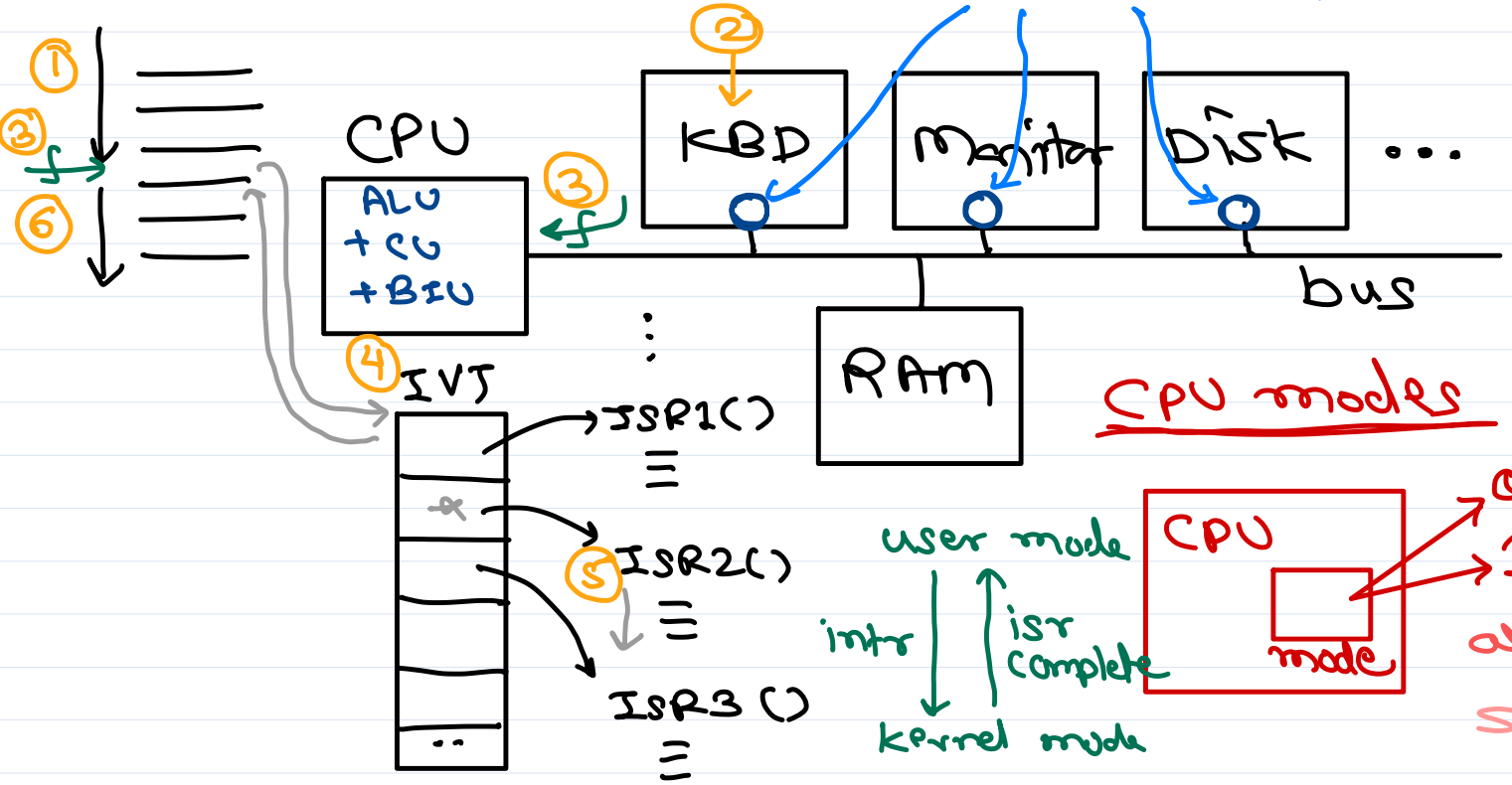


Operating System Concepts

Sunbeam Infotech

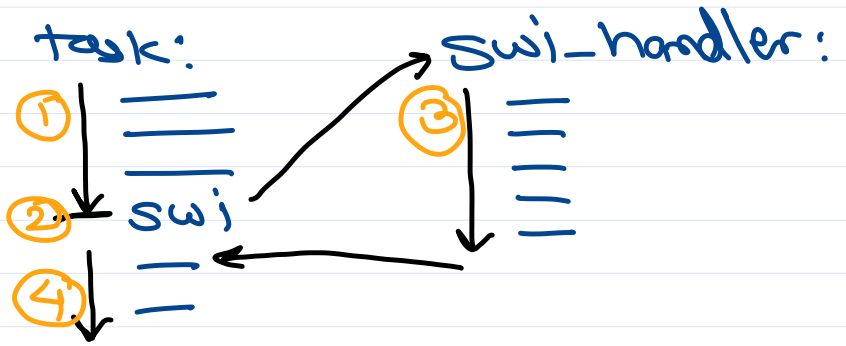
Interrupts and CPU modes

* fns exposed by kernel so that user pgm access. kernel fn.
* UNIX - 64 sys calls
* Linux - 300+
* Windows - 3000+



Interrupts
- Sent by IO devices
- hw intr
- sw intr ← special instruction (CPU specific)

e.g. ARM → SWI or SVC
x86 → INT

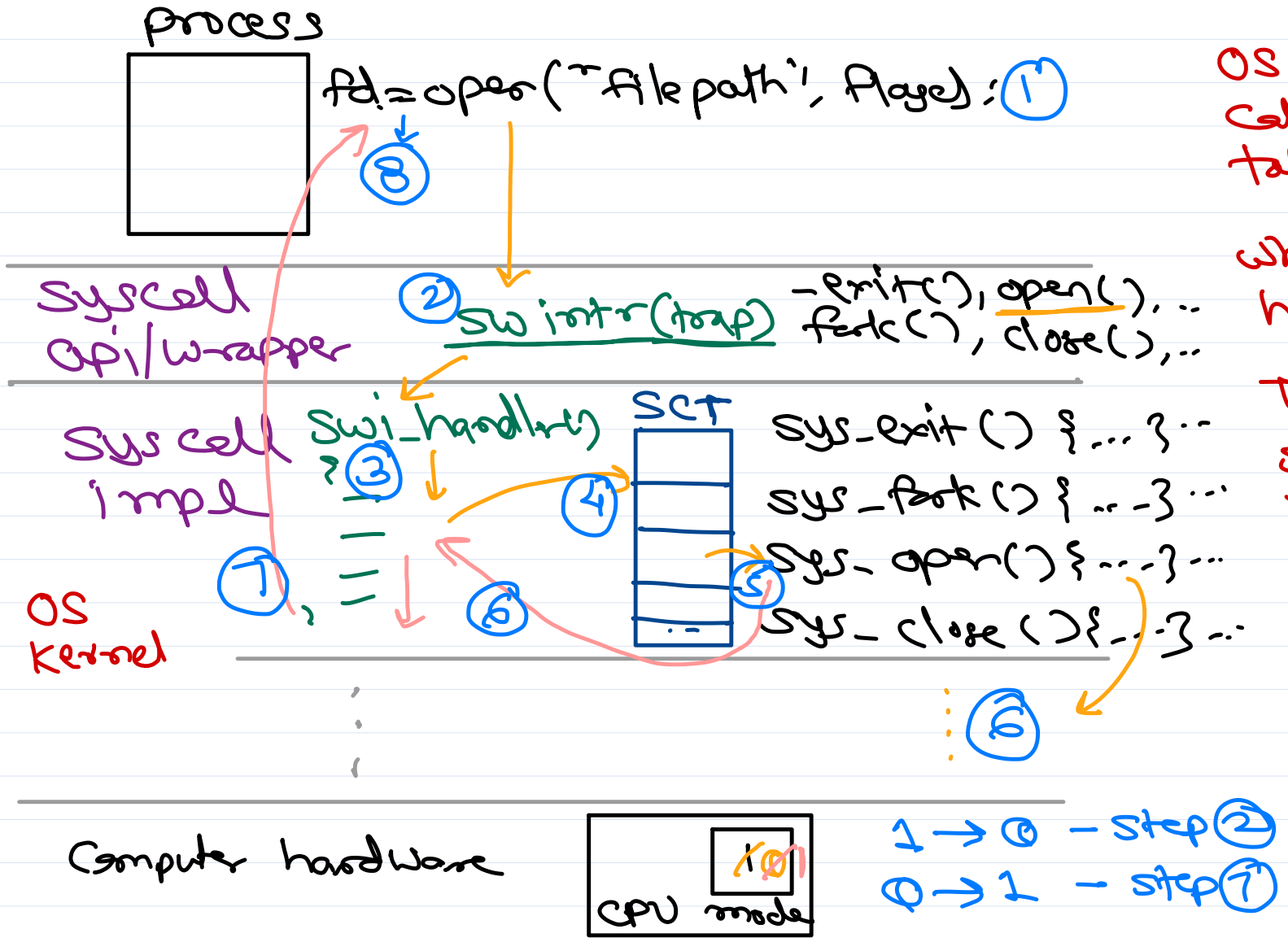


CPU modes

0 : kernel / system / monitor / supervisor privileged mode.
1 : user / unprivileged mode.

all regrs accessible & all ops allowed
special instns not allowed e.g. intr handling, in/out, ...

System calls



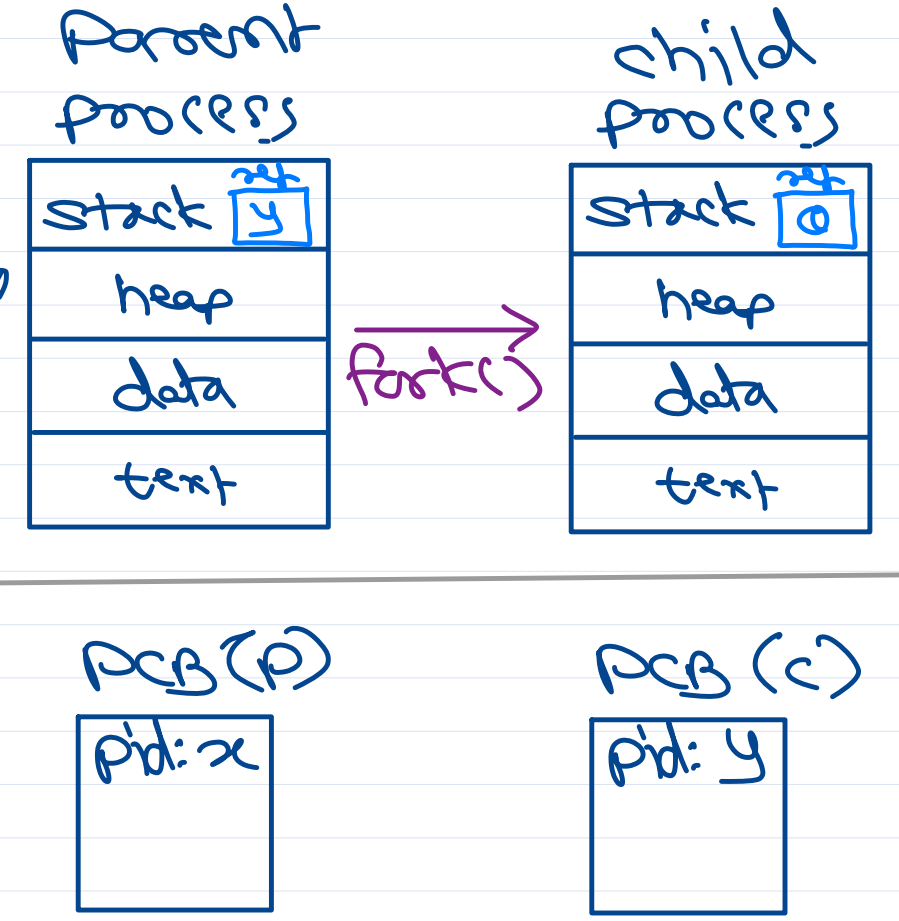
OS stores addresses of all sys call impl into a syscall table (SCT).

When sw intr occurs, swi handler is executed (by hw).

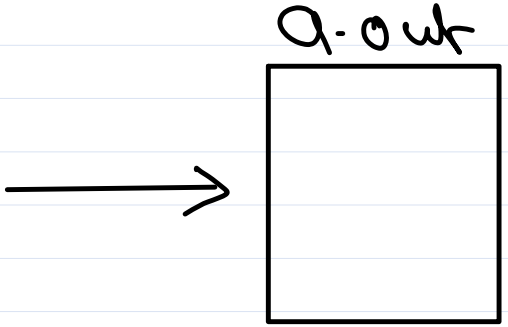
This handler gets addr of syscall from SCT and invoke it.

fork() syscall

* fork() creates a new process by duplicating calling process. The new process is called child; while calling process is called parent.
* child & parent runs in separate memory space
* child & parent are almost same except...
  - a) child pid is different than parent pid.
  - b) ...
* fork() returns 0 in child process; while it returns pid of child in parent process. If failed, fork() returns -1 in parent.

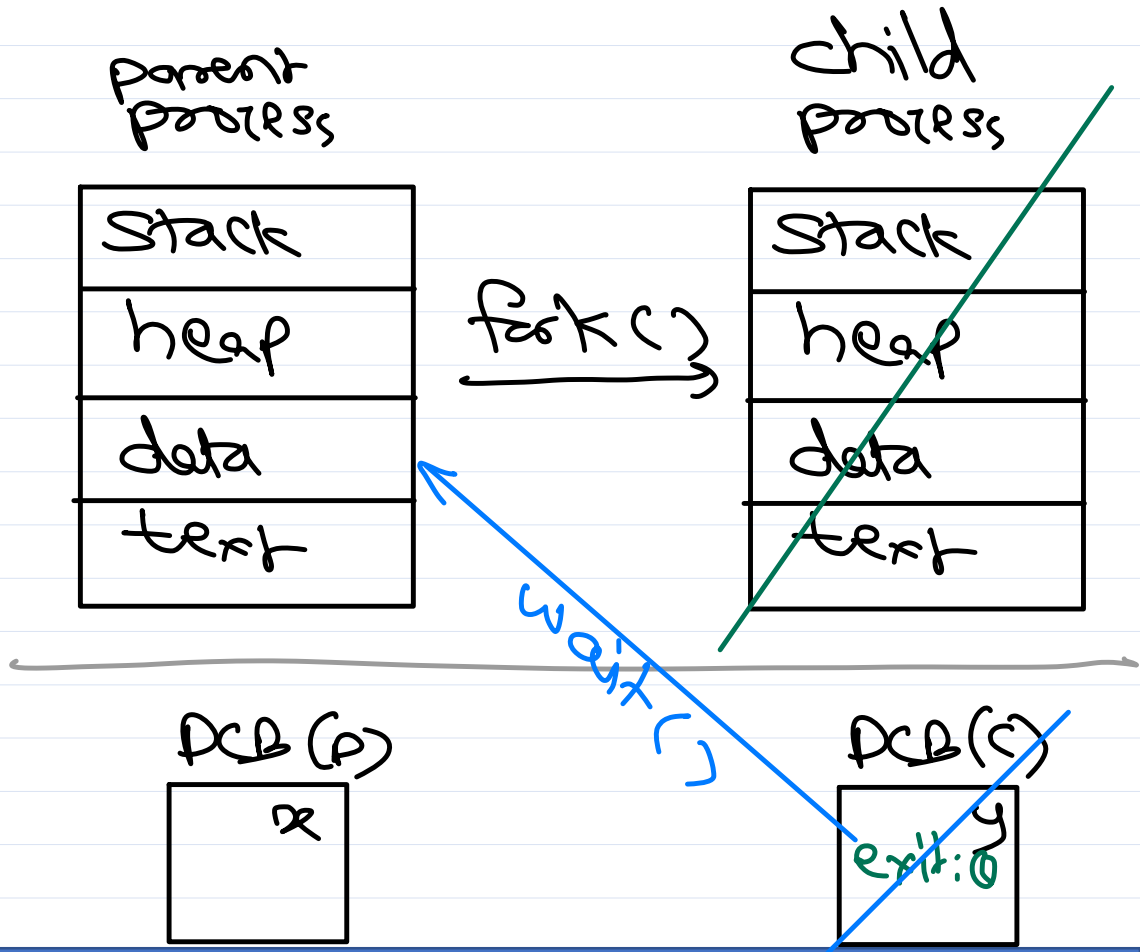



```
int main() {
    int ret;
    —
    ret = fork();
    =
}
```



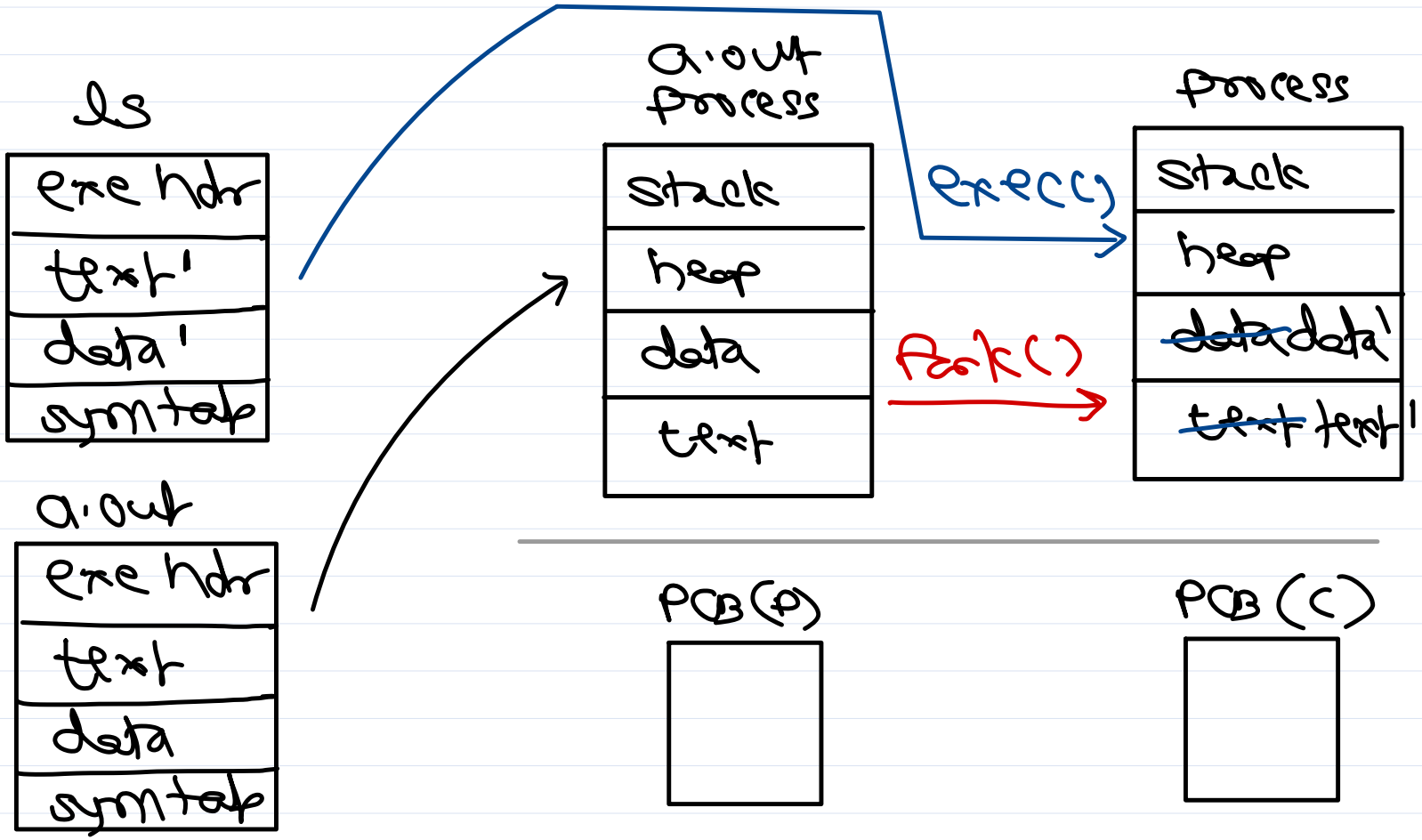
Orphan and Zombie process

- when parent is terminated before child process, the child is said to be orphan.
- ownership/parentship of such child process is handed over to init/systemd (pid=1).
- when child is terminated before parent and parent is not reading exit status of the child, the child process is in zombie state
- In this state all resources of child are released except the PCB.
- The parent must read exit status of child from that PCB, to make process dead(x).
- This is done using wait() sys call,
  1. pause execution of parent until one of its child is terminated
  2. read exit status of child from its PCB.
  3. release PCB of child



exec() syscall

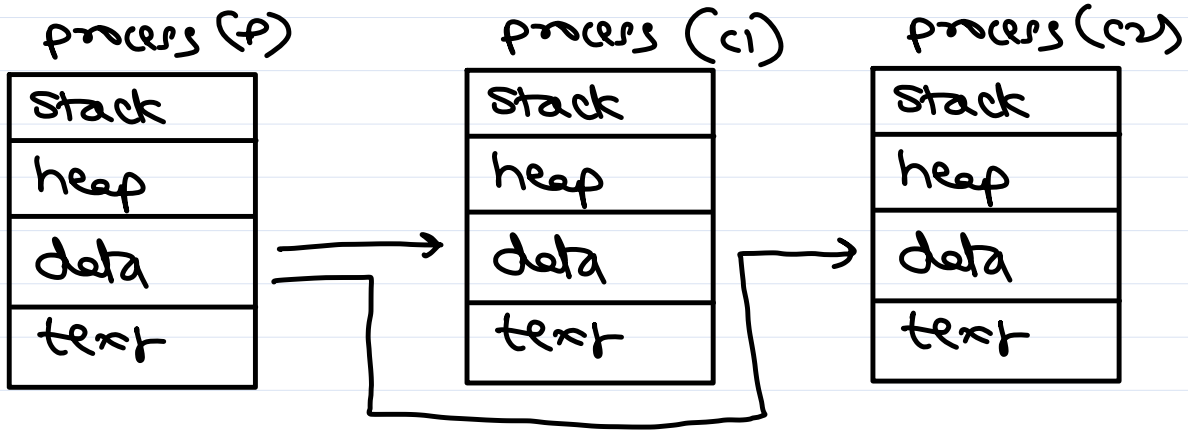
exec() loads a new program in calling process's memory replacing old program image.

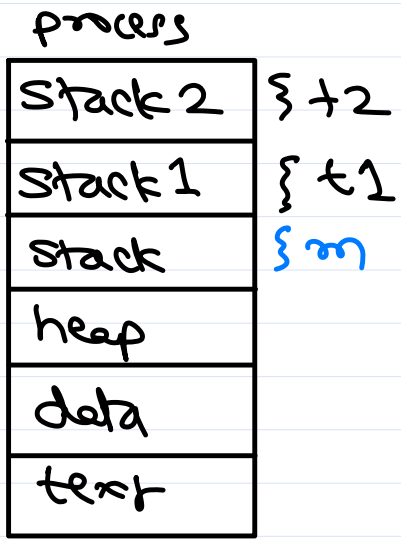



```
r = fork();
if(r==0) {
    e = execl("/bin/ls",
        "ls", "-l", 0);
} else {
    wait(&s);
}
```

Multi-threading

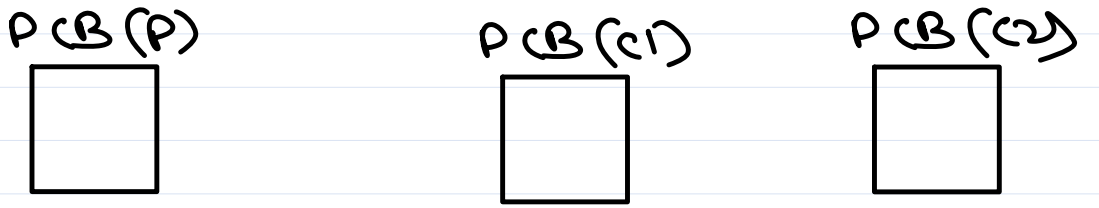
* Threads are used to do multiple tasks concurrently within a process.
* Thread is a light-weight process.

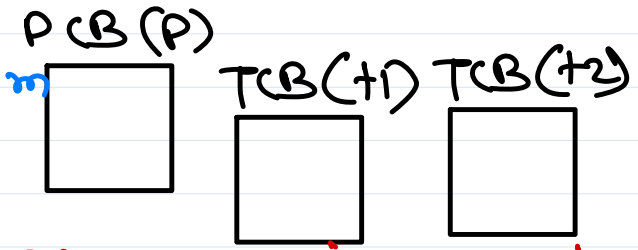




process is a container that holds resources required for execution; while threads are unit of exec/ scheduling.

* PCB contains resource info:
  - pid, exit, mem info, ipc info, files info, ...
* TCB contains exec. info:
  - tid, sched info, state, kernel stack, exec ctx.





for each process one thread created by default called as main thread.

for each thread a new stack & a new TCB is created. The remaining sections are shared with parent process.

Thread

POSIX thread lib.

step1 → create a thread fn.

```
void* thread_func(void *param) {
}
```

step2 → create a thread

```
ret = pthread_create(&th, NULL, thread_func, NULL);
```

arg1: thread id (out param)

arg2: thread attrib (stack size, priority, sched, ...)

NULL = default attrib

arg3: func to be executed by thread.

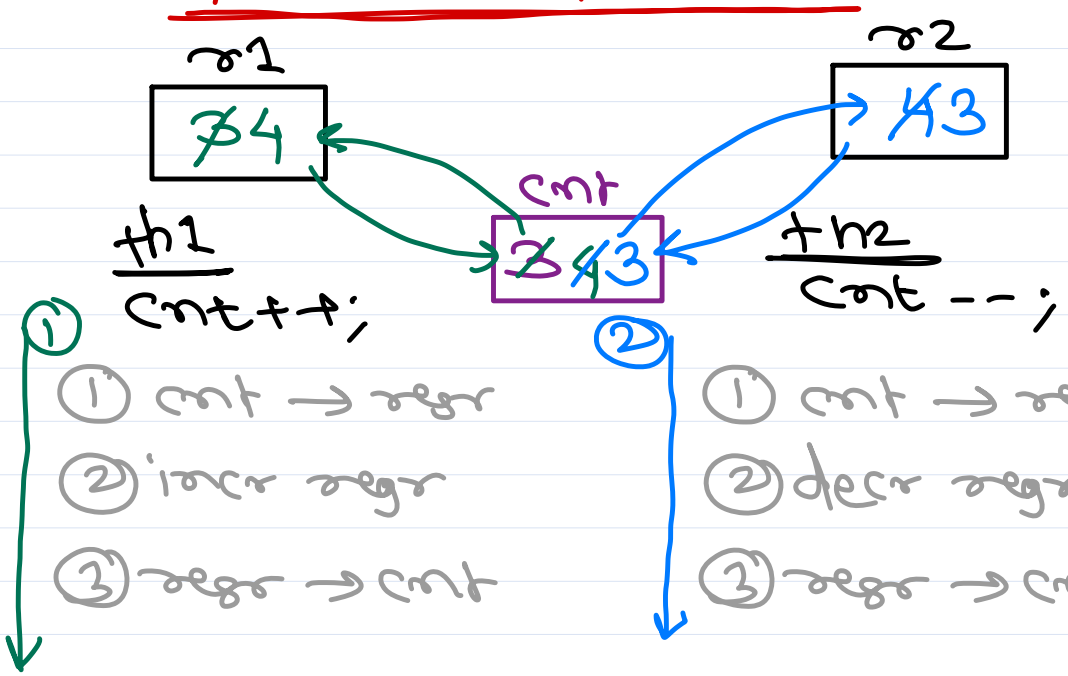
arg4: arg to thread func.

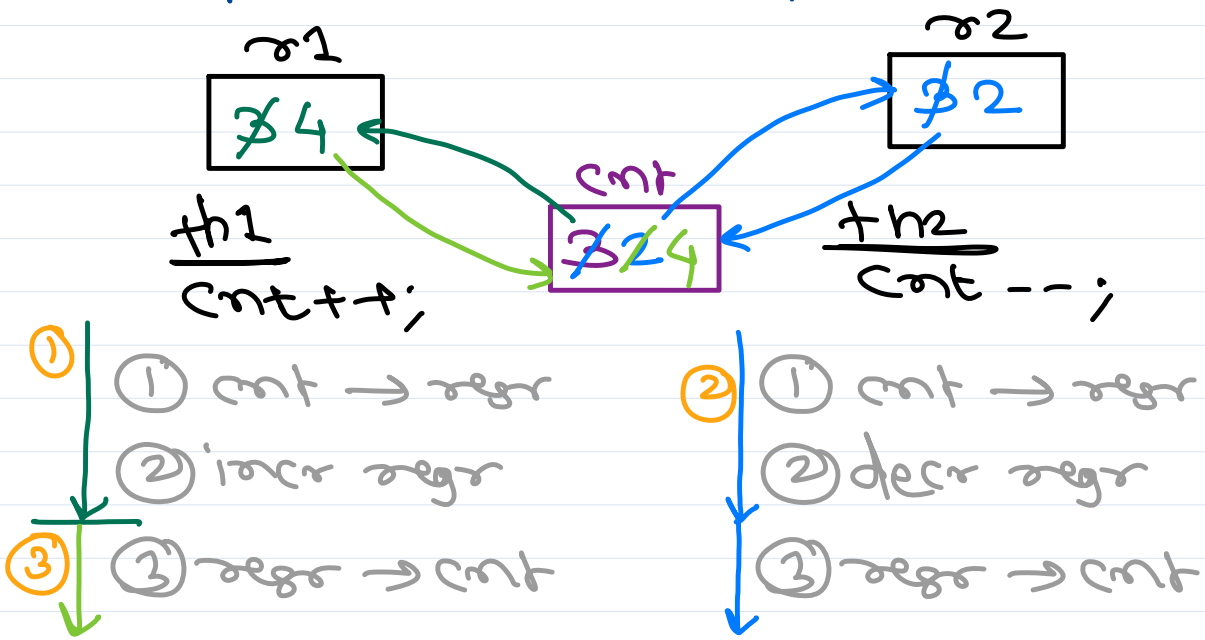
*POSIX - Unix Std (by IEEE)

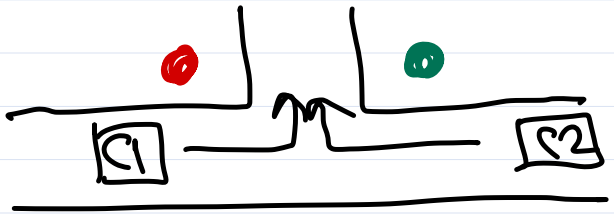
↳ Portable Operating System Interface for Xwindows

Race condition and Synchronization

When multiple threads try to access same resource at the same time, then race condn occurs. It may lead to unexpected results.



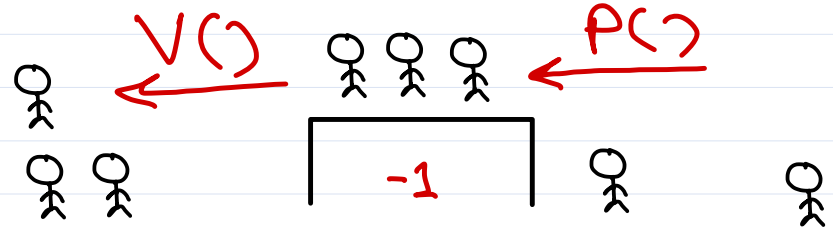




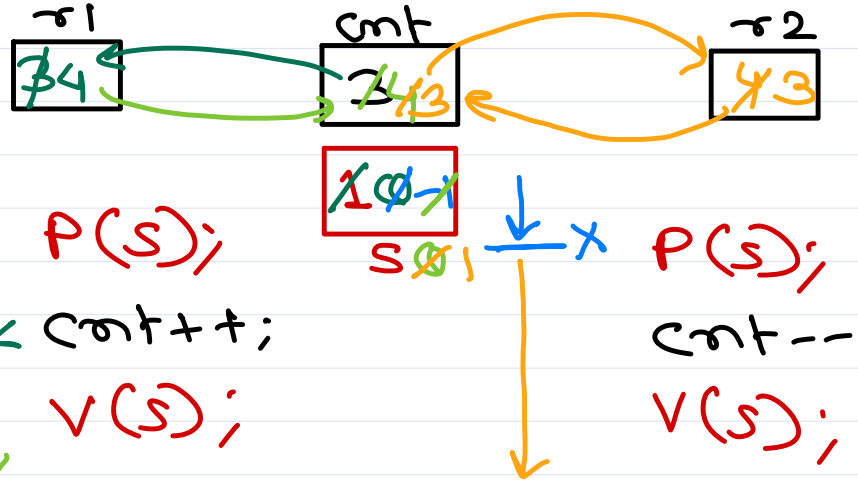
To solve race cond, only one process/thread should be given access to resource while blocking others. this is called as "synchronization".

Semaphore vs Mutex

- Dijkstra
- Semaphore is counter.
- dec op - wait op - P op
  - decr cnt by 1.
  - if cnt < 0, block cur process/thread.
- inc op - signal op - V op
  - incr cnt by 1.
  - if one/more threads/processes are blocked, then wake up one of them.



- Semaphore types
  - counting sema
  - binary sema



Sema applns:

1. Counting - resource/ processes
2. mutual exclusion
3. Condition / event

Mutex: specialized sync obj for mutual exclusion. Two states: locked / unlocked.
Operations: lock(m); unlock(m);

th1
```
lock(m);
== Res
unlock(m);
```

th2
```
lock(m);
== Res
unlock(m);
```

Thread locking the mutex become owner of the mutex and only owner can unlock it.

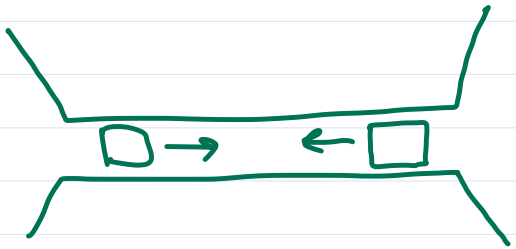
Deadlock

program 1

1. open file
2. connect printer
3. print file
4. disconnect prn
5. close file

program 2

1. connect printer
2. open file
3. print file
4. close file
5. disconnect prn



deadlock characteristics

1. no preemption
2. mutual exclusion
3. hold & wait
4. circular wait

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>