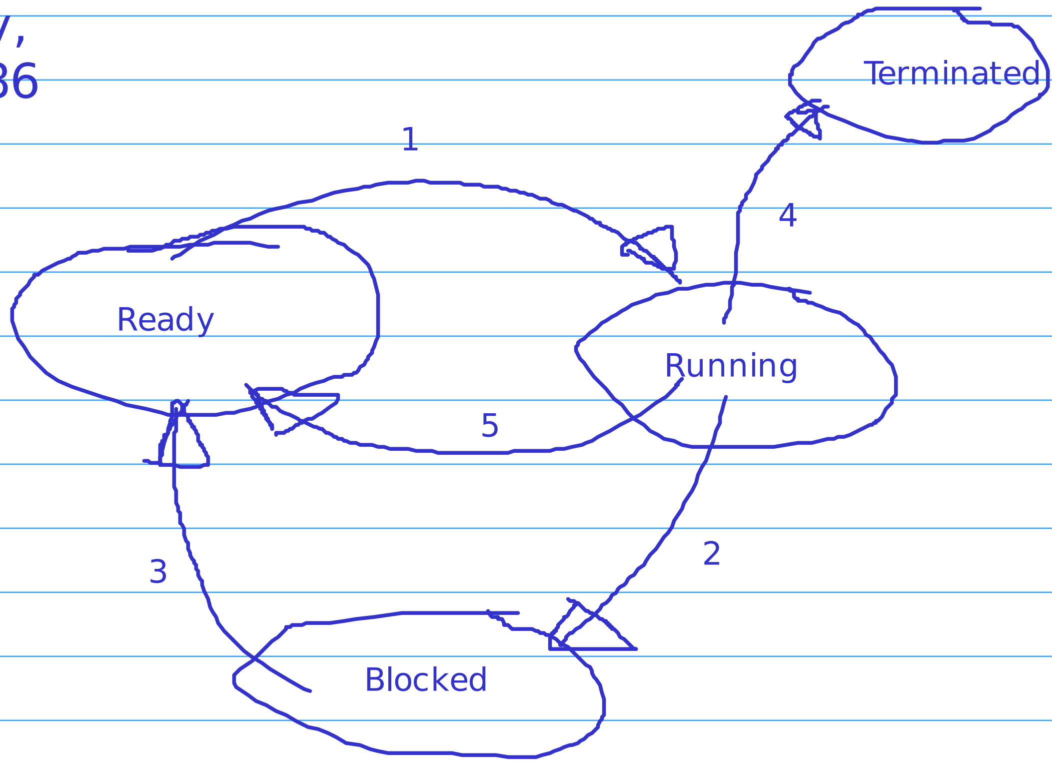
Concurrency & IPC in Kernel:-

child process -- fork
multi threading

multiplexing of CPU, under preemptive, non preemptive conditions
context switching
- context saving
- context loading

context save area in memory,
typically on top of stack in x86

Every user process
will have independent
stack, (independent
address space beyond
address space)




Threads --
Private stack for each

Each thread/process, will have independent/private stack
in userspace, as well as private stack for each thread/process
in kernel space

kernel code execution can be blocked/preempted, due to
private stacks -- process context of kernel execution
-- kernel code running in process context

sometimes kernel codes in non process context (interrupt context)
-- not associated with any user process
-- no private/independent stack
-- blocking/preemption not allowed

concurrency -- physical/true parallelism
-- logical/pseudo parallelism

Task driven parallelim -- each thread does diff task
Data driven parallelism -- same task, but operating on diff data

Userspace threading:-

| | |
|---|---|
| pthread_create | pthread_self |
| pthread_join | pthread_cancel |
| | pthread_equal |
| | pthread_yield |

struct task_struct ==> attributes of each process
current macro ==> address of task_struct var for active process
e.g. current->pid, represents pid of active process

init_task, task_struct variable for init process
next_task, task_struct variable for next process in list
list_for_each
---------------------------------------------------------------------
Race conditions

Critical Section

Mutual Exclusion -- access to shared resource by only
[competition] once process/thread at a time
only can execute critical section
Solutions:-
* Semaphores
* Mutex
* Spinlocks

Synchronization
[cooperation,dependency,sequencing,prod/cons]
* Semaphores
* Condition vars (uspace) Wait Conditions (kspace)

val=10

val++ val--

r0 <-- val r1 <-- val
r0 = r0 + 1 r1 = r1 - 1
r10 --> val r1 --> val

11 9

-------------------------
sem.val=1

lock lock
//critical //critical
unlock unlock
----------------------------
sem.val=0

lock(sem)
//prod:add //cons:remove
unlock(sem)

Semaphore:-
- kernel level data structure
- integer value, process/thread Q

lock/down/acquire:-
A - if val>0, val--, go ahead
B - if val==0, block current
add to waiting Q

unlock/up/release:-
C - if Q is not empty, allow any
one waiting process/thread
to resume
D - if Q is empty, val++
----------------------------------------
Mutex vs Semaphore
Salient features/characteristics
of Mutex
* ownership applicable
* unlocking before locking
unlocking more than once
not allowed
* two state - T/F, unlocked/locked
* strictly mutual exclusion

semaphore/mutex: lock -- block process
unlock -- resume/unblock process
involves context switching

Spinlock -- busyloop technique, to avoid context switching

flag=0

| | P1 | P2 |
|---|---|---|
| lock : | while(flag); | while(flag); |
| | flag=1; | flag=1; |
| | //critical | //critical |
| | flag=0; | flag=0; |

Spinlocks/busyloops are meaningful for multicore (SMP) only

H/w supported atomic instructions, (disable interrupts + bus locking)
e.g XCHG in x86, SWP in ARM

flag=0, reg=1

lock:- while(XCHG(reg,flag)); lock:- while(XCHG(reg,flag));

//critical

unlock:- flag=0 flag=0

SMP:-        Semaphore/Mutex        Spinlock

compare critical section lengt (vs) context switching time

-----------------------------------------------------------------------

```
static DEFINE_SEMAPHORE(s1);

down_interruptible(&s1);
//for loop
up(&s1);
```

-----------------------------------------------------------------------

```
static DEFINE_MUTEX(m1);

mutex_lock(&m1);
//for loop
mutex_unlock(&m1);
```

-----------------------------------------------------------------------

```
static DEFINE_SPINLOCK(s1);

spin_lock(&s1);
val++; //val--;
spin_unlock(&s1);
```

Synchronization with Semaphores:-

struct semaphore s2; //static DEFINE_SEMAPHORE(s2);

init:-
```
  sema_init(&s2,0);
```

Thread-A, before for loop:-
```
  down_interruptible(&s2);
```

Thread-B, end of for loop:-
```
   up(&s2);
```
---------------------------------

```
wait_queue_head_t w1;
int buflen=0;

init:-
  init_waitqueue_head(&w1);

Thread-A:-
        wait_event_interruptible(w1, (buflen > 0) );

Thread-B:-
        buflen++;
        wake_up_interruptible(&w1);
```