execl vs execlp ??

```
execl("/usr/bin/gcc", "gcc", "hello.c", NULL);

execlp("gcc", "gcc", "hello.c", NULL);

char* argv[] = { "gcc", "hello.c", "-c" , "-o", "hello.o", NULL);

execv("/usr/bin/gcc", argv);
```

```
char *argv[] = { "cal", "10", "2018", NULL };


int main(int argc, char* argv[])        ==> argv[0]
```

main:-

```
kill(pid, signo);
```

Argument parsing hints:-   getopt

Private ready queues in SMP
CPU Affinity
Reason - cache entries (private cahce)
Migration requires cache discard & rebuild
Load Balancing issues
some techniques - push migration, pull migration (TODO)

Inter Process Communication:-

shmget, shmat, shmdt
shm_open ??

POSIX APIs
Sys V APIs

Inter Process Communication:-

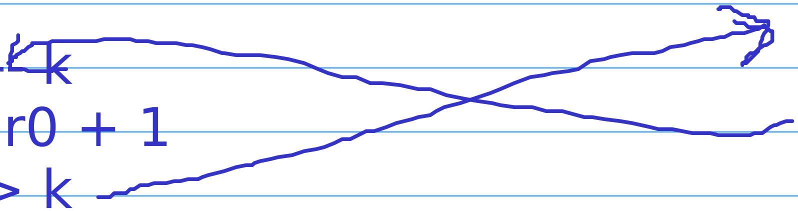
* Data Exchange, shared memory, message Qs, fifos/pipes
* Synchronization
    * Mutual Exclusion semaphore, mutex, spinlocks
    * Dependency/Sequencing semaphores, cond vars/event flags

Semaphores, Mutex, Message Qs

Race conditions

e.g. shared printer, concurrent write ops on a file

		P1/T1	k=10	P2/T2	
		k++		k--	
load		r0 <-- k		r1 <-- k	i0
store		r0 = r0 + 1		r1 = r1 - 1	i1
		r0 --> k		r1 --> k	i2
	11			9	



a) No switching between i0, i1, i2 (i.e. after i0 or i1)

b) switching in between, i.e after i0, i1

Switching is fine before i0 or after i2, but not in between

Critical section
Mutual Exclusion

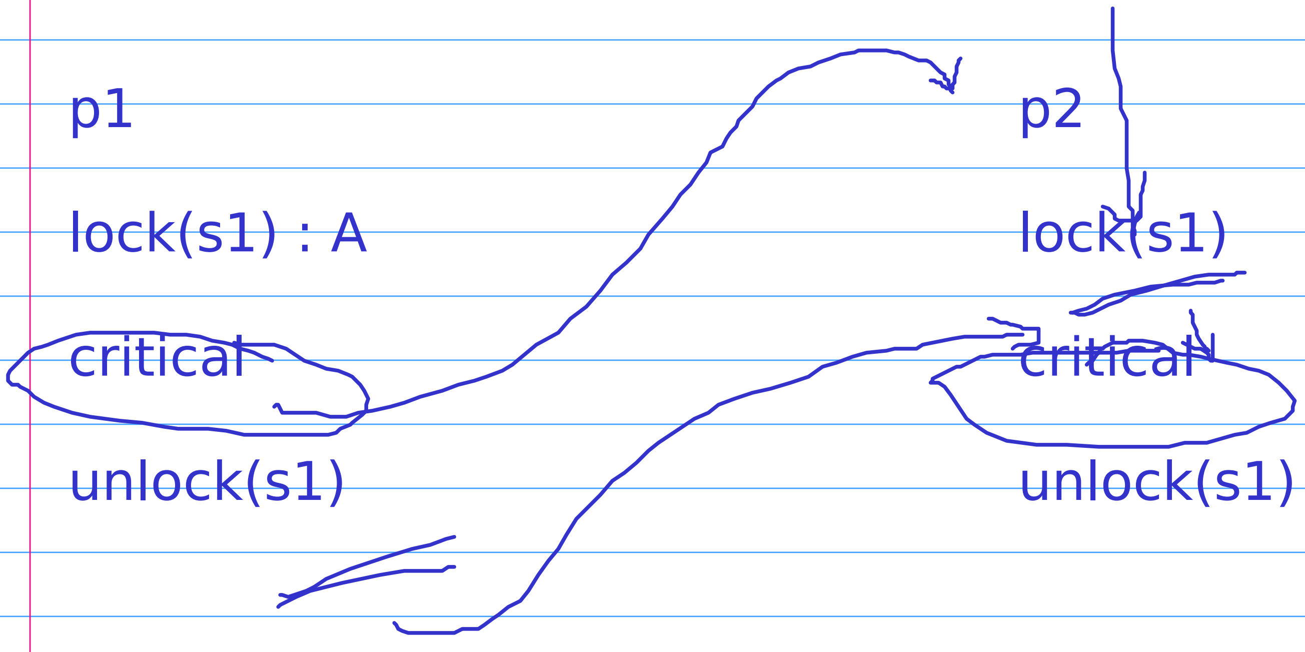
Achevie Mutual Exclusion?
* Disable interrupts?   Limitations/challenges

Semaphores:-

Semaphore s1;

s1.value = ~~1~~ ~~0~~ 1

sem.waitQ: ~~P2~~



p1

lock(s1) : A

critical

unlock(s1)

C

p2

lock(s1)   B

critical

unlock(s1)

D

Semaphore sem
sem.value=0

Producer

// add data (push)
unlock(s1)

C

Consumer

lock(s1) : B
//remove data (pop)

a) consumer schedueled first B, C
b) producer scheduled first D, A

prace.c

POSIX Semaphores:-

```
#include<semaphore.h>

sem_t sm;                  //where?? global

ival=1
sem_init(&s1, 0, ival);    //where?? before create

sem_destroy(&s1);  //where?? after join

sem_wait(&s1);             //lock, before val++/val--

sem_post(&s1);             //unlock, after val++/val--
```

-lpthread
-lrt

prace.c   ==> psemdemo.c

unnamed semaphores (no file name / path associated)
applicable for usage on shared address space (threads)
sem_init, 2nd param : 0 means usage on shared address space

```
pconcur.c        ==> modify this so that only one for loop
                     will execute at a time
```

Dependency / Sequency:-

```
sem_t s2;

sem_init(&s2, 0, 0);        //initial value 0
sem_destroy(&s2);
```

```
before for loop of one thread, say B     :     sem_wait(&s2);
after for loop of other thread, say A    :     sem_post(&s2);
```

```
      A                                  B

sem_wait(&sm);                           sem_wait(&s2);
                                         sem_wait(&sm);

//for loop                               //for loop

sem_post(&sm);                           sem_post(&sm);
sem_post(&s2);
```

Binary Semaphores : 0 and 1 only
Counting Semaphores : 0 and any +ve value

Mutex:-

* Mutex is not just a Binary Semaphore (way beyond)
* Ownership applicable
* Unlocking twice or unlocking before locking is not allowed

pthread.h

```
pthread_mutex_t m1;                    //declare

pthread_mutex_init(&m1);

pthread_mutex_destroy(&m1);

pthread_mutex_lock(&m1);

pthread_mutex_unlock(&m1);
```

Rewrite race cond example (val++, val--) concurrency example(for loop with sleep) using mutex @@

Named semaphores:-

associated with file name (path)
applicable for difference processing running in diff addr space

```
sem_t *ps;
ps = sem_open("s1",O_CREAT, 0666, 1);                    //internal shared mem

sem_destroy(ps);               //after waitpid

sem_wait(ps);
sem_post(ps);
```

Refer example nsdemo.c

```
prod                        cons
                            lock(sm)                    lock(s2)
lock(sm)                    lock(s2)                    lock(sm)


unlock(sm)                  unlock(sm)                         //Don't lock sm first
unlock(s2)                                                     //without checking s2
```

mutual/circular dependency   ==> Deadlock

Avoid deadlock:-
* If multiple locks are required, lock them all at once (atomic locking)
* Don't apply mutual exclusion before resolving depending

```
    copy        : sp  - printer,  ss - scanner

          p1                               p2
          lock(sp)                         lock(ss)                  Atomic locking
          lock(ss)                         lock(sp)                  is a solution
                                                                     follow same order
          //copy                           //copy
```

Message Queues
Pipes & Fifos (after file system)
Shared memory (along with memory concept)

Quick testing of embedded linux (custom kernel, prebuilt rootfs
on Qemu)

Any links on POSIX Semaphores (post read)
POSIX Message Queues (pre read + explore examples)
Pipes / Fifo (pre read + explore examples)
Assignment -2 , first 2-3 questions (prepare stack, circular buffer)

Alpha/Beta User testing:- [Optional/Additional]

https://gitlab.com/gea-training/elinux-bsp/kprog-drivers/
==> first steps