Systems Programming Cheat Sheet

Mutex locks

A *mutex* is a lock that we set before using a shared resource and release after using it. When the lock is set, *no other thread can access the locked region of code*. So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So this *ensures a synchronized access of shared resources in the code.*

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* doSomeThing(void *arg) {
    pthread_mutex_lock(&lock);

    unsigned long i = 0; counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
} int main(void) {
    int i = 0; int err;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init failed\n");
        return 1; }

    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL, &
            doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]"
                , strerror(err));
        i++; }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0; }
```

Signal locks

Signal handling

A **signal** is a condition that may be reported during program execution, and can be ignored, handled specially, or, as is the default, used to terminate the program.

```
FILE *temp_file;

void leave(int sig) {
  fprintf(temp_file,"\nInterrupted..\n");
  fclose(temp_file);
  exit(sig); }
main() {
  (void) signal(SIGINT,leave);
  temp_file = fopen("tmp","w");
  for(;;) {
          printf("Ready...\n");
          (void)getchar();
  }
  exit(EXIT_SUCCESS); }
```

Multiprogramming

Fork

Exec

Wait

Shell scripting

Shared memory

Pointers to functions

```
void func(int);
main() {
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
    exit(EXIT_SUCCESS); }
void func(int arg){
    printf("%d\n", arg);
}
```

Deadlock

Semaphores

A *semaphore* is a special type of variable that can be incremented or decremented, but *crucial access to the variable is guaranteed to be atomic*, even in a multi-threaded program. If two or more threads in a program attempt to change the value of a semaphore, the system guarantees that all the operations will in fact *take place in sequence.*