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 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

The `fork()` system call will spawn a new child process which is an identical process to the parent except that has a new system process ID. The process is copied in memory from the parent and a new process structure is assigned by the kernel. The return value of the function is which discriminates the two threads of execution. A zero is returned by the fork function in the child's process.
The environment, resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process.

```
char * message; int n;

pid_t pid = fork();

switch (pid) {
    case -1:
        perror("fork failed\n");
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        break;
    default:
        message = "This is the parent";
        n = 3;
        break;
}

for(; n > 0; n--) {
    puts(message);
    sleep(1);
}
```

Exec

The `exec()` family of functions will initiate a program from within a program. They are also various front-end functions to `execve()`.

The functions return an integer error code.
The function call `execl()` initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that `arg0` is the command/file name to execute.

```
execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char
    *) 0);
```

Wait

`wait()`: Blocks calling process *until the child process terminates*. If child process has already terminated, the wait() call returns immediately. if the calling process has multiple child processes, the function returns when one returns.
`waitpid()`: Options available to block calling process for a particular child process not the first one.

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);
}
```

Condition variables

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables *allow threads to synchronize based upon the actual value of data*.
Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. *A condition variable is a way to achieve the same goal without polling.*
A condition variable is *always used in conjunction with a mutex lock*.

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*.