EE 469L: Operating Systems Lab 2

DLXOS Version P02S.2

Various APIs

1 User Programs

- **Prototype:** `extern int process_create(char *arg1, ...)`

  Usage: `process_create(program_name, param1, param2, ...paramn, NULL);`

  Description: The function `process_create()` creates a user process which runs the program `program_name` and passes the rest of the arguments as command-line parameters to the user program. The parameters param1, param2, ... are passed in the form of character strings. These parameters can be retrieved from the `main(int argc, char *argv[])` function of the user program according to normal C convention. Thus, in the user programs, `argc` denotes the number of command-line arguments (including the `program_name`) passed to `process_create()`. Also `argv[0]` is a character string which contains the program name, `argv[1]` is a character string which contains the first command-line argument, and so on.

  Limitations: The number of parameters to `process_create` must be less than 128. Also the total length of all the parameters, including the terminating '\0' characters must be less than 1024.Also, the argument list has to be terminated by a NULL pointer (see Usage). The results are unpredictable if this NULL is omitted.

2 Shared Memory

- **Prototype:** `uint32 shmget()`

  Usage: `handle = shmget();`

  Description: `shmget()` allocates a shared page (page size 64K) in the physical memory and returns a system-wide unique handle that corresponds to this page. The page is also automatically mapped to the virtual address space of the calling process. The handle as it is cannot be used to access the shared page, but needs to be passed to `shmat()` (see below) to get a virtual address corresponding to this page.

Limitations: At most 32 shared pages can exist in the system at any given time. Also, any process cannot have more than 16 pages allocated to it (including the shared and the non-shared pages). Under such circumstances (i.e. if the total number of shared pages is equal to 32, or the calling process is already using 16 pages), the function returns the value 0, indicating that a shared page was not allocated. The DLXOS shared-memory API does not support any protection. All pages created using `shmget()` have read-write permissions.

- **Prototype:** `void *shmat(uint32 handle)`

  Usage: `ptr = shmat(handle);`

  Description: `shmat()` maps the unique shared-memory page (page size 64K) identified by the `handle` to the address space of the calling process, and returns the address that points to the base of this page. If the memory page identified by `handle` is not a shared page, `shmat()` does not do any mapping, and returns the NULL pointer. Also, if the calling process is already using 16 memory pages, `shmat()` does not do any mapping, and returns the NULL pointer. If the page pointed to by `handle` is already mapped to the address space of the calling process, `shmat()` returns the address of the base of this page without any further re-mapping. A page mapped to the address space of a process remains mapped until the process exits by making a call to `exit()`. A page that is not mapped to the address space of any process is freed and added to the list of the free pages. Hence the process that creates a shared page should not quit till some other process maps the shared page to its address space.

  Limitations: DLXOS shared-memory API does not support protection. Any shared page in the system can be mapped by any process to it's address space. Also, once a page is mapped, the calling process has both read and write permissions to this page. Permissions cannot be given selectively.

3 Semaphores

- **Prototype:** `sem_t sem_create(int count)`

  Usage: `sem = sem_create(int count);`

Description: `sem_create` grabs a semaphore from the static list of semaphores, initializes it to `count`, and returns a unique handle corresponding to this semaphore. All the future references to this semaphore must be made using this handle.

Limitations: There can at most be 32 different semaphores in the system. The function fails if all the 32 semaphores are already being used. In that case, it returns `INVALID_SEM`.

- **Prototype:** `int sem_wait(sem_t sem)`

  Usage: `retval = sem_wait(sem);`

  Description: `sem_wait()` decrements the count of semaphore `sem` and waits if the count is negative. The function fails if `sem` is not an initialized, valid semaphore. In such a case, it returns 1 without waiting and without modifying the count. Otherwise it returns 0 indicating that the call was successful.

  Prototype: `int sem_signal(sem_t sem)`

  Usage: `retval = sem_signal(sem);`

  Description: `sem_wait()` increments the count of semaphore `sem` and wakes up a process waiting on the semaphore. The function fails if `sem` is not an initialized, valid semaphore. In such a case, it returns 1 without without modifying the count. Otherwise it returns 0 indicating that call was successful.