

## COMP SCI 452 Operating Systems

### Some Linux commands for process creation and waiting, shared memory, signaling, semaphores, message queues, and other facilities

These are just general descriptions and complete descriptions can be found on the man pages and in the books on reserve.

#### **tar -cvf dirname.tar dirname**

*tar* stands for (tape archival) and is originally designed for archiving files on a tape backup. It is now commonly used in *ftp* for transferring multiple files or directories. When used with the options above (*cvf*) this command creates a single file, *dirname.tar* that contains the files and directory structure of the directory *dirname*. It's a lot like *winzip*, but without the compression. Options are as follows: *c* means create an archive, *f* means use the file name to follow for the archive, *v* means verbose which displays on the screen what *tar* is doing as it is doing it.

#### **tar -xvf dirname.tar**

reverses what the previous command does and restores the directory and its contents. Option *x* stands for extract.

#### **gzip file**

Creates a file named *file.gz* that represents the compressed version of the specified file. It uses a variation of the Lempel-Ziv algorithm (LZ77). It is often used to compress a tar file. For example, the command *gzip dirname.tar* creates a file named *dirname.tar.gz*. Thus, the combined *tar* and *gzip* commands function much like *winzip* in Windows.

#### **gunzip file.gz**

Will uncompress the specified file.

#### **compress**

Creates a file named *file.Z* that represents the compressed version of the specified file. It uses a variation of the Lempel-Ziv algorithm (adaptive Lempel-Ziv). Although *compress* was included in the original Unix distributions, *gzip* (if it is installed on your system, and it usually is) will generally compress better.

#### **uncompress file.Z**

Will uncompress the specified file.

#### **void perror(const char \*s)**

Prints string *s* followed by a standard error message associated the *errno*, an external *int* value defined when errors occur in system functions. Should include <stdio.h> and <errno.h>.

#### **unsigned sleep(unsigned i)**

This is a common form of the sleep command with causes the process to sleep *i* seconds. There are other forms which you read about through the *info sleep* command.

#### **int rand(void)**

Returns "random" integer between 0 and *RAND\_MAX*, a constant defined in *stdlib.h*.

#### **void srand(unsigned int seed)**

Initializes "random" number generator. Often used with the argument (*unsigned*) *time* (*NULL*) so that random patterns do not repeat. Should include <stdlib.h>.

#### **void exit(int status)**

Exits and returns a status (the low-order 8 bits of the *status* variable) to the parent process. Calls on any functions specified by the *atexit* function. Also calls on *\_cleanup* which in turn calls *\_exit* function.

### **pid\_t fork(void)**

Creates a child process. Returns process id (in parent), 0 (in child), or -1 if call fails. Sets external *errno* variable. *pid\_t* typically an int should be in *sys/types.h*. When the child finishes it becomes a zombie process and the kernel sends a SIGCHLD signal to the parent notifying the parent of the child's demise (change of status). The child stays in the system until it has been waited for by a parent (*wait* call). If parent dies first, a system process *init* inherits the orphaned zombie and waits for it. Should include <unistd.h>.

### **int execlp("path to executable file", arg0, arg1,... , (char\*) NULL )**

In general an *exec* command redefines the current process by overlaying its code with that of the specified executable. It returns -1 if it fails and sets *errno*; otherwise, it does not return. *execl* means parameters are passed via a parameter list; *execv* means parameters passed via the *argv[]* array. *execlp* and *execvp* mean the PATH string (environment variable \$PATH) is used to search for files that might be specified in a parameter or if code to be executed contains file references. *execle*, *execve* means programmer passes own environment variable list; path specifies directory path to executable; *arg0*, etc. correspond to parameters that would be passed to the program if it were invoked via a command line. Consequently, *arg0* is typically the executable file name.

### **extern int errno;**

An external variable set by system calls when they fail. Its contents are examined by *perror* and an error message displayed.

### **pid\_t wait(int \* status)**

If no child exists, return -1 and set *errno*. If a child exists, wait for one to finish (i.e., wait for a SIGCHLD signal) and return the child *pid* and its exit status. If the child executed normally the low order byte of status contains 0 and the next byte contains the exit code. If the child terminated due a signal, the low order byte contains the signal number and the next byte contains 0. In the second case, if a core file was created the leftmost bit of the low order byte will be 1.

### **pid\_t waitpid(pid\_t pid, int\* status, int options)**

Similar to the *wait* function except it provides more flexibility. The first parameter specifies a specific (or set of) processes to wait for. The second parameter specifies the status of the child process if the wait is successful. The third parameter specifies options. For example, WNOHANG specifies the calling process should not be blocked. That is, it does not wait. The function returns -1 if the process no longer exists. If the child has finished and is a zombie the function returns the child's pid and its status. If the child's status is not available (i.e. if a wait call would have blocked the process) waitpid returns 0.

### **int kill(pid\_t pid, int signal)**

Sends a signal to the specified process or process group. There are many possible signals as defined in *signal.h*. The function returns 0 if successful and -1 if not. Also sets *errno*. When issued at the shell level the default signal is SIGTERM (15), which usually terminates a process if it is not ignoring signals. If it doesn't work, kill -9 sends a KILL signal. Some other common signals are SIGINT=2=^c (interrupt), SIGQUIT=3=^ (quit), SIGKILL=9 (exit).

### **void (\*signal (int sig, void (\*sigcatcher) (int))) (int);**

NOTE the following notation:       int (\*f) (int) => f is a pointer to a function, which returns an int.  
  int \*f (int) => f is a function that returns a pointer to an int.  
  int (\*f(int)) (int)=>f is a function that returns a pointer to a function.

SO! *signal* is a function (with two arguments) that returns a pointer to a function with one *int* parameter. *signal* has two parameters: an integer and a pointer to a function (*sigcatcher*) that returns void. When called, *signal* associates a signal catching function with the specified signal. If that signal occurs, the signal catching function is automatically invoked. The signal catching function has one parameter, typically the signal that occurred. Note that the signal catching function can be specified as SIG\_DFL or SIG\_IGN, which resorts to the default action for the signal or specifies that the signal should be ignored. As specified, *signal* returns a pointer to a function. This function corresponds to the previous disposition of the signal, i.e. the function or value that specified how these signals were handled previously. NOTE: There is a *sigaction* function, which does a similar thing. NOTE: a process currently executing in the signal handler will NOT catch subsequent signals of the same type until the signal function is reinvoked. See *man signal*.

**key\_t ftok(const char \* path, char id);**

Generates a key value dependent on a legitimate path specifier and character. The path ID “.” (the self-referential directory) may be specified as it is always present. Keys are used in IPC (inter-process communication) constructs such as shared memory, semaphores, and message queues.

**int pause(void);**

Suspends a process until it receives a signal, which it has not ignored.

**int shmget(key\_t key, int size, int flags);**

Creates or gains access to a shared memory segment (minimum size specified by *size*) and associated data structure (type *struct shmid\_ds*). Returns -1 if it fails, else returns a shared memory identifier to be used subsequently. Also, sets *errno*. Standards must be set up to determine key values so that different projects don’t accidentally use the same key. One approach uses the *ftok* function to create a key. This function creates a NEW segment in the following cases:

- key is the symbolic constant `IPC_PRIVATE`
- key is not associated with another segment and flag contains `IPC_CREAT`.

flags contains access permissions (9 low order bits defining permissions for owner, group, and world, similar to file access permissions shown via the command *ls -l*) and possibly `IPC_CREAT` (to create a segment) and `IPC_EXCL` (used with `IPC_CREAT` to ensure failure if the segment exists). These are typically logically OR’d together. Should include `<sys/types.h>`, `<sys/ipc.h>`, and `<sys/shm.h>`.

In addition, the *struct shmid\_ds* contains fields such as *shm\_segsz* (segment size), *shm\_cpid* (creator process id), *shm\_nattch* (number of attaches), and other fields such as the last attach, detach, and change time.

**int shmctl(int shmid, int cmd, struct shmid\_ds \* buf);**

Performs some functions on a shared memory segment. *shmid* specifies the segment. *buf* points to the data structure describing the segment. The function (or command) is specified by *cmd* as follows:

- `IPC_STAT` places values of the *shmid\_ds* structure into a *struct* located by *buf*.
- `SHM_LOCK` or `SHM_UNLOCK` locks or unlocks shared memory segment in memory (must be superuser).
- `IPC_RMID` removes system data structure associated with segment. System will remove actual segment when all references to the segment are eliminated. *buf* is usually 0 here.
- `IPC_SET` to change some items in the *shmid\_ds* structure.

**void \* shmat(int shmid, void \* shmaddr, int flags)**

Attaches to a particular shared memory segment and returns the virtual address used to reference it. Returns -1 if it fails. First parameter *shmid* specifies the segment. Second parameter allows calling process to specify which address it wants to use for the segment. If 0 is specified (advisable), system picks the address. Third parameter can be used for access conditions or special requests such as read only segments of aligned addresses. Usually use 0 here.

**int shmdt(void \* shmaddr)**

Detaches from specified memory segment.

**ipcrm**

Can be used to remove IPC resources that did not get removed for some reason. i.e. the command

```
ipcrm shm id
```

removes the shared memory resource associated with *id* (found by entering *ipcs*)

## ipcs

When entered at the command line, shows all IPC resources owned by user.

## int semget( key\_t key, int numsems, int semflag)

must include <sys/sem.h>

semaphore data structure:

```
struct semid_ds
{
    struct ipc_perm    sem_perm;    // operation permission struct
    struct sem        *sembase     // location of first semaphore
    :
    :
}

struct sem
{
    ushort    semvalue;    //semaphore value
    pid_t    sempid;    // pid of last operation
    ushort    semncnt;    // # processes waiting for semval > cval
    ushort    semzcnt;    // # processes waiting for semval = 0
}
```

- the *key* parameter (perhaps created by *flock*) is used by the system to generate a unique semaphore identifier.
- *numsems* is the number of semaphores requested in the set.
- *semflag*-low order bits specify access permissions; can also use flags IPC\_CREAT and IPC\_EXCL
- functions returns -1 if failure or semaphore id if success.
- *semid\_ds* structure locates an array of semaphores via the *sembase* field.

## int semctl( int semid, int semnum, int command, union semun arg)

- *semid* is the semaphore identifier
- *semnum* is the number **or index of a semaphore within** the semaphore set

```
union semun {
    int val;
    struct semid_ds * buf;
    unsigned short int *array;
    struct seminfo * __buf;
}
```

The *buf* and *array* fields must be defined to locate user defined structures/arrays before the call to *semctl*.

- *command* specifies the semaphore command (the 2nd parameter *semnum* can be 0 in the first 3 cases):  
IPC\_STAT returns the current values of the semaphores in a user structure specified by *arg.array*;  
IPC\_SET can be used to change the *uid*, *gid*, and mode fields of the *sem\_perm* field of the *semid\_ds* structure.  
It requires superuser status or if current process id matches that in *cuid* or *uid*.  
IPC\_RMID removes semaphore set associated with semaphore id.

GETALL returns current values of semaphore values to a user-defined array located by *array*.

SETALL initializes all semaphores in a set to the values contained in the user-defined array.

GETVAL returns value of one semaphore (specified by *semnum*, 1 is first semaphore in position 0 of the sets array)

SETVAL sets value of specified semaphore (*semnum*) to that in *arg.val*.

GETPID returns process ID from *sem\_perm*.

GETNCNT returns the number of processes waiting for the specified semaphore to increase.

GETZCNT returns the number of processes waiting for the specified semaphore to be 0.

### **int semop( int semid, struct sembuf \*sops, size\_t nsops)**

- *semid* is the semaphore identifier.
- *sops* locates the base address of an array of semaphore operations. The operations are done on an all or none basis.
- *nsops* is the number of semaphore operations specified in *sops*.

```
struct sembuf{
    ushort    sem_num;           // semaphore # (0 IS FIRST NUMBER)
    short     sem_op;           // operation
    short     sem_flg;           // flags
}
```

flags:

IPC\_NOWAIT if a *semop* cannot be done, return immediately & do not change any other semaphores.

SEM\_UNDO if a process exits, semaphore value is changed by the value of *semadj*.

Possible operations are:

negative *sem\_op* value: subtract  $|sem\_op|$  from *semval* if result is  $\geq 0$ ; if SEM\_UNDO is one of the flags, add  $|sem\_op|$  to *semadj*.

if result would be  $< 0$ , increment *semncnt* (# processes waiting on semaphore), block process until (1) *semval*  $\geq |sem\_op|$ , (2) semaphore is removed, or (3) signal is caught. In case (1), proceed as above; in case (2) return -1 and set *errno* to EIDRM; in case (3), adjust *semncnt* and set *errno* to EINTR.. If IPC\_NOWAIT was set, then just return -1 immediately and set *errno* to EAGAIN.

positive *sem\_op* value: add *sem\_op* to *semval*; if SEM\_UNDO is one of the flags specified then also subtract *sem\_op* from *semadj*.

zero *sem\_op* value: if *semval* is 0, return immediately; if *semval*  $\neq 0$  increment *semzcnt* and wait until (1) *semval* is 0; (2) semaphore is removed; (3) signal is caught. In case (1), adjust *semzcnt* and return; in case (2), return -1 and set *errno* to EIDRM; in case (3) adjust *semzcnt* and set *errno* to EINTR; if *semval*  $\neq 0$  and IPC\_NOWAIT is specified, return -1 and set *errno* to EAGAIN;

### **int msgget(key\_t key, int msgflg)**

If successful, returns a message queue identifier associated with the key (obtained via *ftok*). Returns -1 if it fails. Message flags are similar to the flags defined for shared memory and semaphores.

### **int msgctl(int msqid, int command, struct msqid\_ds \* buf, . . .)**

Must include <sys/msg.h>

assumes the declaration

```
struct msg{
    struct msg * next_msg;       //pointer to next message in queue
    long msg_type                //message type
    ushort msg_ts                // message size
    char * msg_spot              // message text address
}

struct msqid_ds{
    struct ipc_perm msg_perm;    // permissions
    struct msg *msg_first;       // first message on queue
    struct msg *msg_last;       // last message on queue
    ulong msg_cbytes;           // # bytes in queue
    ulong msg_qnum;             // # messages in queue
}
```

```
:  
:  
}
```

*msqid* is the queue identifier returned by *msgget*.

the command parameter can be one of the following:

IPC\_STAT: return current values for each member of the *msqid\_ds* list; uses third argument to receive this information

IPC\_SET: can modify some fields of the *msqid\_ds* structure.

IPC\_RMID: removes all message queue structures

NOTE: all messages must begin with a field of type *long*;

**int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg)**

- *msqid* is the message queue ID;
- *msgp* is a pointer to the message to be sent
- *msgsz* is the message size
- *msgflg* specifies what to do if system limits governing the queue have been reached. If the flag is IPC\_NOWAIT the message is not sent and control returns to the calling process with *errno* set to EAGAIN. If set to 0, the function will block (wait) until the limit is no longer at system max at which time the message is sent.

**int msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);**

- As a function *msgrcv* returns the number of bytes received.
- *msqid* is the message queue ID;
- *msgp* locates a buffer in which the message is placed; the first field must be a long integer to hold the message type;
- *msgsz* is the maximum size of the message;
- *msgtyp* is the message type requested; 0 means get the first message of any type; positive means get the first message with the specified type; negative means get the first message of lowest type  $\leq |msgtyp|$ ;
- *msgflg* specifies what to do if a specified message does not exist or is too big; IPC\_NOWAIT means return immediately if message not there; MSG\_NOERROR means truncate messages that are too long (otherwise, *msgrcv* returns -1 and *errno* is set to E2BIG);