Process Management (Cont'd)

```
Get your own PID
Get your parent's PID
Get your own process group ID
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t getpid(void);

       pid_t getppid(void);

       pid_t getpgrp(void);

   returns:  PID of interest, cannot fail

EXAMPLE:

   printf("My PID is %d\n", getpid());
   printf("My parent's PID is %d\n", getppid());
   printf("My process group ID is %d\n",
                     getpgrp());
```

Process Management (Cont'd)

```
Get real and effective UIDs and GIDs
SYNOPSIS
       #include <unistd.h>
       #include <sys/types.h>

       uid_t   getuid ()

       gid_t   getgid ()

       uid_t   geteuid ()

       gid_t   getegid ()

   returns:  ID of interest, cannot fail

EXAMPLE:

   printf("My RUID is %d\n", getuid());
   printf("My RGID is %d\n", getgid());
   printf("My EUID is %d\n", geteuid());
   printf("My EGID is %d\n", getegid());
```

Process Management (Cont'd)

```
Set real and effective UIDs and GIDs
SYNOPSIS
       #include <unistd.h>
       #include <sys/types.h>

       uid_t   setuid (uid)
       uid_t  uid;

       uid_t   setgid (gid)
       gid_t  gid;

   returns:     0 on success or -1
EXAMPLE:
   int rgid, ruid;

   printf("My RUID is %d\n", getuid());
   printf("My RGID is %d\n", getgid());
   printf("My EUID is %d\n", geteuid());
   printf("My EGID is %d\n", getegid());

   if(setuid(ruid) == -1 | setgid(rgid) == -1){
        perror("setting IDs back failed: ");
        exit(1)
   }

   printf("My EUID is now %d\n", geteuid());
   printf("My EGID is now %d\n", getegid());
```

Process Management (Cont'd)

```
Set session and group IDs for job control
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t  setsid ()

       int setpgid (pid, pgid)
       pid_t pid, pgid;
   where:
       pid        The process id of the process whose
                  process group id is to be changed.
       pgid       The new process group id.
   returns:     returns: 0 on success or -1
EXAMPLE:
   int my_pid;

   if((my_pid = getpid()) != getpgrp()){
       if(setpgid((pid_t)0, my_pid) == -1){
          perror("group leader attempt failed: ");
          exit(1);
       }
   }
EXAMPLE:
   if(setsid() == -1){
     perror("cannot become session leader: ");
     exit(1);
   }
```

Process Management (Cont'd)

```
Process priority adjustment
SYNOPSIS
       #include <unistd.h>

       int  nice (incr)
       int  incr;

   where:
       incr   A positive or negative value that
              is to be added to the calling process's
              priority

   returns:   New nice value on success or -1 on
              error. Always clear errno before call
DESCRIPTION
    The value of incr is added to the priority of the
    calling process.  A more positive priority value
    results in a lower level of service from the CPU.

    If the new priority would be greater than 19, the
    process's priority is set to 19.  If the new
    priority would be less than -20, the process's
    priority is set to -20.
```

Process Management (Cont'd)

nice()   cont'd

EXAMPLE:

```
int new_pri;

printf("current nice value is %d\n", nice(0));
if  ((new_pri = nice(+8)) == -1){
    perror("nice change failed: ");
    exit(1);
}
printf("new nice value is %d\n", new_pri);

printf("current nice value is %d\n", nice(0));
if  ((new_pri = nice(-8)) == -1){
    perror("nice privileged change failed: ");
    exit(1);
}
printf("new privileged nice value is %d\n", new_pri);
```

Process Management (Cont'd)

Process priority adjustment BSD style

```
SYNOPSIS
       #include <sys/resource.h>

       int  getpriority (which, who)
       int  which;
       int  who;

       int  setpriority (which, who, prio)
       int  which;
       int  who;
       int  prio;
  where:
    which   How the argument who is to be interpreted
            in identifying one or more processes
            whose  priorities will be set:
            PRIO_PROCESS, PRIO_PGRP, or PRIO_USER

    who     Identifier of one or more processes
            whose priorities will be set: a process
            ID, a process group ID, or user ID,
            depending on the value of which

    prio    The new priority value (range -20 to 20)

  returns:  getpriority -- new nice value on success
            setpriority -- 0 on success
            both        -- -1 on error
```

Process Management (Cont'd)

getpriority(), setpriority()   cont'd

```
EXAMPLE:
   int max_uid_old_pri, new_pri;

   if((max_uid_old_pri =
             getpriority(PRIO_USER, getuid())) == -1){
       perror("getpriority failed: ");
       exit(1);
   }

   new_pri = max_uid_old_pri + 10;
   if(setpriority(PRIO_USER, getuid(), new_pri) == -1){
       perror("set un-privileged priority failed: ");
       exit(1);
   }

   new_pri = max_uid_old_pri - 10;
   if(setpriority(PRIO_USER, getuid(), new_pri) == -1){
       perror("set privileged priority failed: ");
       exit(1);
   }
```

Signal Management

UNIX systems employ an asynchronous process notification mechanism known as the **signal** facility. Signals often cause portability problems since there are several similar but different signal implementations. The common implementations include:

- USL System VR3 (22 defined signals)
- USL System VR4 (32 defined signals)
- BSD 4.3 (32 defined signals)
- OSF/1 (32 defined signals)
- DGUX (64 possible, 38 defined)

Even where the number of signals is the same, the actual signals in the set are often slightly different.

Signal Management (Cont'd)

Signals can be used as a means of notifying a process of some event in an asynchronous way. In essence, a signal is sent from a process to a process, and the sender and receiver may be the same process.

Signals are organized into two main groups, those called *synchronous* are always delivered as a result of a process run-time exception, and the offending process delivers the signal to itself during exception processing. Synchronous signals include **SIGSEGV, SIGPIPE** and **SIGFPE** , among others.

The so-called *asynchronous* signals are delivered to a process by another process, and are in no direct way tied to any specific behavior of the target process. Asynchronous signals include **SIGINT, SIGTERM** and **SIGKILL** , among others.

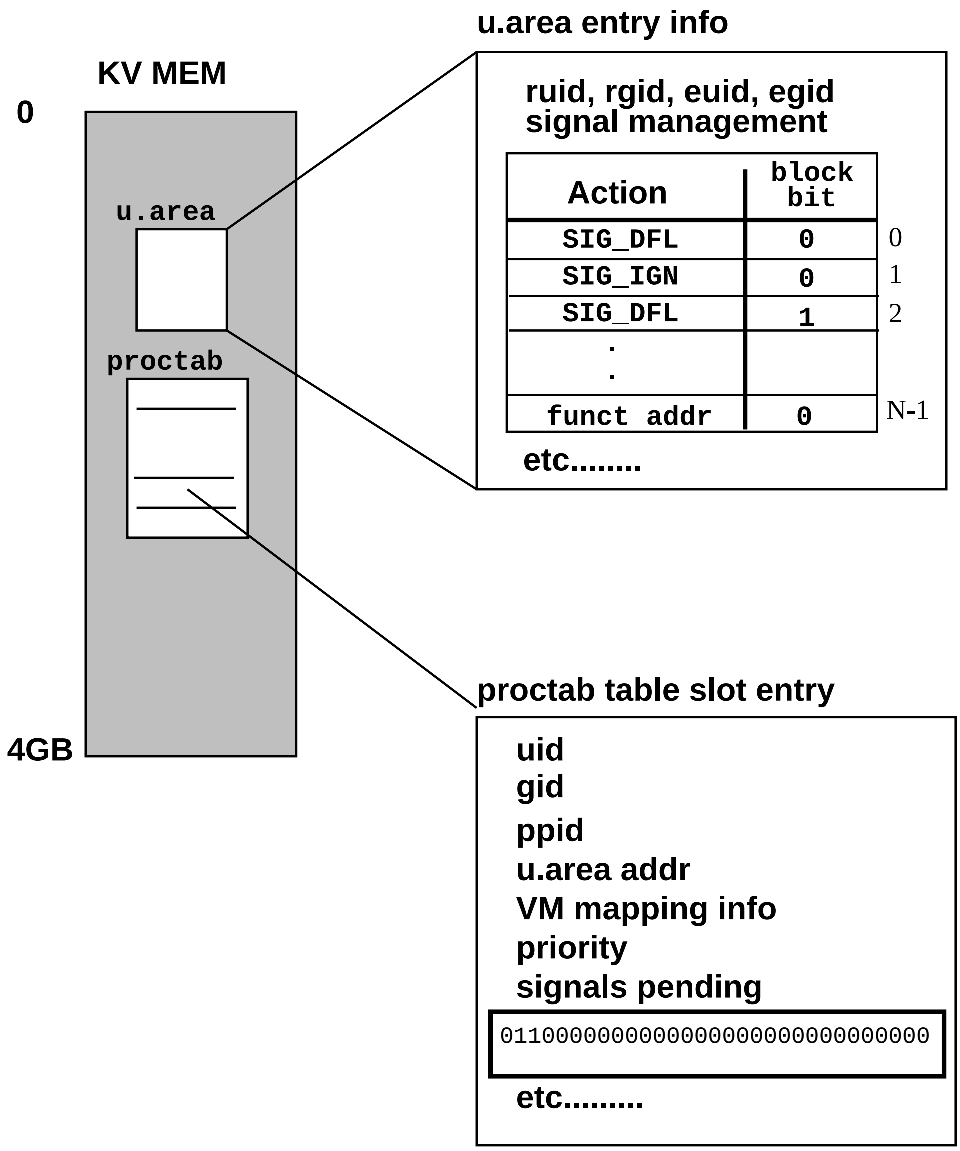
Signal Management (Cont'd)

Each process keeps a *signal action table* in the *u.area* and a *pending signal vector* in its **proctab** entry. When a signals is launched from a sending process to a target process(es), a bit is turned on in the pending signal vector of the proctab entry of the target process.

Before a process is allowed to transition from *kernel running* to *user running* the process must check its pending signal vector for any outstanding signals, and must manage such signals before it can return to its place in user space.

Just exactly how the detecting process will handle a specific signal depends on what information the process has in its u.area signal action table in the table slot which corresponds to the outstanding signal.

Signal Management (Cont'd)



Signal Management (Cont'd)

A process can arrange to handle arriving signals in one of three basic ways:

- The process can set the signal for a signal defined default action with **SIG_DFL**
- The process can set the signal to be discarded upon arrival with **SIG_IGN**
- The process can install a user supplied function address pointing to a function to be invoked should the signal arrive

Most signal defined default actions force the process to terminate by calling exit upon signal detection.

User defined signal handling functions must be of type **void** and will be passed a single integer parameter when invoked, which is the signal number which caused the invocation.

If the *block bit* is set to **1** for a given signal and the signal should arrive to a process, the process will not attempt to handle the signal until the corresponding block bit is set to **0**. If additional instances of the signal should arrive while the block bit is set, such instances will be queued for future disposition.

Signal Management (Cont'd)

The following signals are defined for USL SVR4

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGABRT	6*	abort
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe, no open read
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user signal 1
SIGUSR2	17	user signal 2
SIGCHLD	18@	child status changed
SIGPWR	19@	power fail/restart
SIGWINCH	20@	signal window size change
SIGURG	21@	urgent socket condition
SIGPOLL	22	pollable event
SIGSTOP	23|+	stop (cannot be caught, blocked, or ignored)

Signal Management (Cont'd)

USL SVR4 Signals   cont'd

SIGTSTP	24|+	stop signal from keyboard
SIGCONT	25@	continue after stop
SIGTTIN	26|+	background read attempted
SIGTTOU	27|+	background write attempted
SIGVTALRM	28	virtual time alarm
SIGPROF	29	profiling timer alarm
SIGXCPU	30*	CPU time limit exceeded
SIGXFSZ	31*	file size limit exceeded

where: no indication means default termination
* means termination with a core dump
|+ means specific job control action
@ means default ignore action

Signal Management (Cont'd)

Examine or change signal action

SYNOPSIS

```
   #include <signal.h>

   int    sigaction (sig, act, oact)
   int    sig;
   const struct sigaction *act;
   struct sigaction *oact;
```

```
 where:
   sig        A signal number.
   act        NULL, or a new action to be
              installed for sig.
   oact       NULL, or the current action associated
              with sig.  If act is not NULL and the
              call is successful, oact will be replaced
              by act.
```

```
  struct sigaction{
   void (*)() sa_handler; /* SIG_DFL, SIG_IGN, or
                             pointer to a function. */
   sigset_t   sa_mask;    /* Additional set of signals
                             to be blocked during execution
                             of signal-catching function. */
   int        sa_flags ;  /* Special flags to affect
                             behavior of signal. */
  };
```

Signal Management (Cont'd)

Use the sa_flags field to modify the delivery of the specified signal. The values you can specify, defined in the header <sys/signal.h>, are listed below:

SA_ONSTACK	If set and the signal is caught, and an alternate signal stack has been declared, the signal is delivered to the calling process using the alternate stack. Otherwise, the signal is delivered on the same stack as the main program.
SA_RESETHAND	If set and the signal is caught, the action of the signal is reset to SIG_DFL. (Note: SIGKILL, SIGTRAP, and SIGPWR cannot be automatically reset when delivered. With these signals, setting SA_RESETHAND has no effect.)
SA_NODEFER	If set and the signal is caught, sig will not be automatically blocked while the handler is active.
SA_RESTART	If set and the signal is caught, and if the system call is restartable, the kernel will restart the system call on behalf of the caller after a signal handler completes processing some signal that has interrupted the call. If this flag is not set, a system call that is interrupted will return EINTR. A non-restartable system call that is interrupted will return EINTR regardless of this flag.

Signal Management (Cont'd)

SA_SIGINFO	If this flag is not set and the signal is caught, sig is passed as the only argument to the signal handling function. If this flag is set and the signal is caught, two additional arguments will be passed to the signal handling function. If the second argument is not equal to NULL, it will point to an object of type siginfo_t, which will explain the reason the signal was generated (see siginfo.h). The third argument will point to an object of type ucontext_t, which will describe the receiving process' context at the time it received the signal (see ucontext.h).
SA_NOCLDWAIT	If set and sig equals SIGCHLD, the system will clean up after the calling process's dead children. If the calling process subsequently calls wait(), it will block until all of its processes terminate and then return a value of -1 with errno set to ECHILD.
SA_NOCLDSTOP	If set and sig equals SIGCHLD, sig will not be sent to the calling process when its child processes stop.

Signal Management (Cont'd)

```
sigaction()    cont'd

EXAMPLE:
   struct sigaction   new, old;
   sigset_t   mask_sigs;
   int  i, nsigs;
   int sigs[] = {SIGHUP,SIGINT,SIGQUIT,
                 SIGTERM, SIGXFSZ};

   void  handler(int signum);
           .
           .
   nsigs = sizeof(sigs)/sizeof(int)
   sigemptyset(&mask_sigs);
   for(i=0; i< nsigs; i++)
       sigaddset(&mask_sigs, sigs[i]);       .
   for(i=0; i< nsigs; i++){
      new.sa_handler = handler;
      new.sa_mask = mask_sigs;
      new.sa_flags = SA_RESTART;
      if(sigaction(sigs[i], &new, &old) == -1){
         perror("can't set signals: ");
         exit(1);
      }
      printf("signal # %d previous action was %x\n",
                 sigs[i],  old.sa_handler);
   }
}  /* close main  */
```

Signal Management (Cont'd)

```
sigaction()    cont'd

            .
            .
            .
            .
void  handler(int signum)
{
  switch(signum){
    case SIGHUP: printf("SIGHUP caught\n")
                 /* clean up environment */
                 printf("going down on SIGHUP\n");
                 exit(2);

    case SIGINT: /* interrupt code */
            .
            .
            .
    default:     /* unexpected sig ?? */
  }
}
```

Signal Management (Cont'd)

```
Manipulate sets of signals (from libc)
SYNOPSIS
    #include <signal.h>

    int sigemptyset (sigset_t *set);
    int sigfillset (sigset_t *set);
    int sigaddset (sigset_t *set, int signo);
    int sigdelset (sigset_t *set, int signo);
    int sigismember (sigset_t *set, int signo);

   returns:    sigismember -- 1 if true, 0 if false
               all others  -- 0 on success or -1

DESCRIPTION
       sigemptyset initializes the set pointed to by
       set to exclude all signals defined by the system.

       sigfillset initializes the set pointed to by set
       to include all signals defined by the system.

       sigaddset adds the individual signal specified by
       the value of signo to the set pointed to by set.

       sigdelset deletes the individual signal specified
       by the value of signo from the set pointed to by set.

       sigismember checks whether the signal specified
       by the value of signo is a member of the set pointed
       to by set.

       Any object of type sigset_t must be initialized
       by applying either sigemptyset or sigfillset before
       applying any other operation.
```

Signal Management (Cont'd)

```
Examine and change the block bits
SYNOPSIS
       #include <signal.h>

       int sigprocmask (how, set, oset)
       int how;
       const sigset_t *set;
       sigset_t       *oset;

   where:
       how   The manner in which the current set
             of blocked signals is changed.
       set   NULL, or the signal set used to change
             the current set of blocked signals.
       oset  NULL, or the current set of blocked
             signals.

                 POSSIBLE "how" VALUES
   SIG_BLOCK     The resulting set shall be the union
                 of the current set and the signal set
                 pointed to by the argument set.
   SIG_UNBLOCK   The resulting set shall be the
                 intersection of the current set and
                 the complement of the signal set
                 pointed to by the argument set.
   SIG_SETMASK   The resulting set shall be the signal
                 set pointed to by the argument set.
```

Signal Management (Cont'd)

```
Send a signal to a process(es)
SYNOPSIS
       #include <sys/types.h>
       #include <signal.h>

       int kill (pid, sig)
       pid_t pid;
       int sig;
   where:
       pid    An integer (positive, negative, or
              zero) indicating a process or a group
              of processes to be sent the signal
       sig    A signal number that is either one
              from the list given in <signal.h> or zero

       If pid is greater than zero, sig shall be sent to
       the process whose process ID is equal to pid.

       If pid is zero, sig shall be sent to all processes
       (excluding an implementation-defined set of system
       processes) whose process group ID is equal to the
       process group ID of the sender, and for which the
       process has permission to send a signal.

       If pid is negative, but not -1, sig shall be sent
       to all processes whose process group ID is equal
       to the absolute value of pid, and for which the
       process has permission to send a signal.

       If the value of pid causes sig to be generated for
       the sending process, and if sig is not blocked,
       either sig or at least one pending unblocked
       signal shall be delivered to the sending process
       before the kill() function returns.
```