Dawit Woldegiorgis

Term Project Part V
COP 4600, Fall 2015

I affirm that this project submission is solely the product of my own efforts and that I neither broke nor bent any academic honesty rules.

Dawit Woldegiorgis *December 1, 2015*

TABLE OF CONTENTS:

SECTION I: Answers to the "standard questions"

Q - Does the program compile without errors?
A – Yes, the program compiles without errors.

Q - Does the program compile without warnings?
A – Yes, the program compiles without warnings.

Q - Does the program run without crashing?
A – Yes, the program runs without crashing, as far as I know.

Q - Describe how you tested the program.
A – The testing strategy is detailed in Section VI of this document. The general idea behind the test cases was to take into account what happens when semaphores are used properly, and what happens when they are abused. Additionally, some funky cases dealing with inheritance and access policies are tested to ensure nothing out of the ordinary is happening.

Q - Describe the ways in which the program does not meet
    assignment's specifications.
A – The program meets all the default assignment’s specifications. When it comes to semaphores releasing resources upon exit, there could be a case where a grandchild of a process would lose access to its inheritance if it’s parent dies first. See implementation strategy for more information.
.

Q - Describe all known and suspected bugs.
A – Besides the possible bug* of a process losing its inherited semaphores if it’s parent exits prematurely, there is no other known or suspected bug. Note that this isn’t really a bug, but more of a decision regarding access policy, as the situation regarding what should happen in such situation is unclear.

Q - Does the program run correctly?
A – The program runs correctly. Test results (see Section IX of this document) generated were correct and expected.

Section II: Learning experience

General learning experience:

Broadly speaking, I think I have learnt some tough lessons from this exercise. I learnt that I am more motivated to learn that I had ever thought. While working on this project, there were countless instances where I felt like giving up. In fact, I did give up and throw in the towel once or twice, only to come back to it a day later. I went back to the project because instead of looking it at it as a burden, like I usually do, I decided to tackle it just for the sake of learning and understanding what I don't know. The more I tried, the more frustrated I became, and the more I was motivated to quit, especially because the amount of effort I was putting in didn't seem equivalent to the quantified value put on the project (% of total grade). But, somewhere along the lines, I stopped thinking about all the cost analysis and decided to dedicate myself to the project – just for the sake of learning, if nothing else. I went in with an open mind, considering that I might put in the effort and still end up failing, and I was somewhat ok with that. I'm never ok with failing, but slowly I am learning to accept it as part of a learning process instead of running away. Sure, I didn't get a perfect result in the end, and I probably could have used my time more wisely (here comes that cost analysis again…) but this struggle was a small insight into what I'm likely to face in the future.

In short, I learnt (now deeply embedded in my system) that
- no pain, no gain
- failing is an important part of success

Technical learning experience:

At the beginning of the semester I was not very comfortable running commands on the terminal. Now, I prefer no other way of navigating a system. It's growing on me and I like it. Heck, I even went as far as taking time to learn and write aliases and other command hacks that make life easier – partially motivated by the tedious task of compiling the kernel every time it crashes, which tested my patience quite a few times.

Syntax, syntax, syntax. It's one of the most challenging parts of the process (besides the psychological challenges mentioned above). Learning it, and using it efficiently is quite difficult, and very time consuming. I learnt that I need to learn more. The more that I learn, the more that I learn I need to learn more. It seems like a never ending cycle that I have embraced. I'm more comfortable with macros and preprocessor directives. I wouldn't comfortably say there was an easy part to this project. Some were less daunting that others… let's leave it at that.

I was unable to effectively use SIMPLQ in an earlier project. I was able to overcome that challenge this time around!

Section III: Implementation strategy

There are several parts that were taken into account when implementing the semaphore system calls. The most important question is how do we store semaphores, and where? Semaphores are specific to a process hierarchy, i.e. a semaphore created by a process A can not be accessed by any other process, unless this process is an nth child of process A (0 <= n). Furthermore, we needed to take into account that any child could override an inherited semaphore with one it creates, and be able to get back to the inherited semaphore if the child was to remove the semaphore it created. At the core of the idea is that semaphores are intimately associated with a process and it's family. Therefore, it was logical for a process "object" to have a list of semaphores as part of it's identity.

A process is simply a struct in OpenBSD. Therefore, to add semaphores to the process was as simple (in concept) as adding a data structure in the process structure. The choice of data structure to keep these semaphores was a linked list, implemented in OpenBSD as LIST. A process was to only keep track of semaphores that it created, and nothing else. Then how do we find what a process has inherited? In addition to the list of semaphores, we will need a flag that is set every time a process fork()s. Say a "root" user process (the first process) is forked from a system process. Since it's unlikely that a system process is using our semaphore, this "root" process will not inherit any semaphores. Therefore, we set the *inherited* flag to FALSE. If this process was to create a semaphore and fork(), then the child will have its *inherited* flag set to TRUE. We determine what to set the flag by simply checking if the list of semaphores the process we fork()ed from is empty. If the list is empty, the parent has no semaphores to pass down to it's child, so the child inherits nothing. If the parent has created semaphores, it's list will not be empty, and the child would then set its flag to TRUE, indicating that the parent process has some semaphores that it might want to access later.

But keep track of all this flag and conditions? Why not just copy over the semaphores into the child? One reason is, if there is a tall tree of process hierarchies, and if each child down that process tree creates its own semaphore that has the same name as the parent, by the time we get to the 5^{th} generation children we'll have 5 different semaphores with the same name. This makes it challenging to figure out which semaphore takes precedence over the others when doing the system calls. If the current process has created a semaphore, we can simply check if it owns it, and we have a match. But, if the current process doesn't own the semaphore, then we'd have to find the owner of each semaphore and traverse the process tree upwards to check which process comes first (I'll shortly describe what a semaphore object entails, bot for now let's just assume that we can figure out who owns a semaphore as long as we have access to the semaphore). If we are going to traverse the tree, it would be computationally less demanding if we were to go up the tree in order (this way we know where a process is in the tree), rather than jumping around to figure out where a process lies in the tree. So, whenever a process inherits a semaphore, it sets its *inherited* flag and searching for a semaphore becomes trivial. We search the process's list, and if semaphore is not in there, we get the process's parent and search its list, and so on until we either get the semaphore or we reach a process with an *inherited* flag not set (FALSE). Note that a process structure keeps pointers to its parent and its children, so accessing them is just a matter of grabbing that pointer.

What happens when a process is terminated?
When a process is dying, we'll want to free up all the semaphores it created. Note that if a process has a semaphore, on which other processes are sleeping, and if this process exits before they wake up, it is an error. It is up to the programmer to ensure that a process does not exit while it's semaphore is being used, or else there might be a deadlock. But, when everything is fine and normal, and no one wants a process or its semaphores, then the exit function (the function responsible for freeing up all memory space a process has occupied) will go though the list of semaphores and delete them one by one. Normally, in OpenBSD, when a process exits before the child is terminated, the orphaned child(ren) are now adopted by the init process. This will make them lose all their inherited semaphores, as they now have no family. Logically, it is unclear whether these children should still be able to access their inheritance even after orphaned, but it's a matter of how we want to implement access policy. This implementation tracks and maintains inheritance based on the

process family one is located in, rather than using semantics to determine what is "fair" to a process.

So what is this semaphore and of what is it made?
OpenBSD is mostly written in C, and 'objects' in C are simply made of structs. A semaphore is a structure that has a name, a count that keeps track of access, a lock to supply mutual exclusion and a list that keeps track of all processes that are waiting on the semaphore. Since we're keeping track of semaphores in a process as linked lists (LIST), a semaphore is also a "node" in this linked list. Therefore it has a pointer to the next semaphore in that list, although this could be NULL if the semaphore is a loner. A semaphore also has a pointer to the owner process. Whenever a process creates a semaphore, this pointer is set to the process structure, giving as direct access to the owner through the semaphore.

Now that the definitions and policies are out the way, let's see how operations on semaphores work. There are four system calls that could be performed on a semaphore: allocate_semaphore(), down_semaphore(), up_semaphore(), and free_semaphore().

The allocate_semaphore()function, as it's name implies is responsible for creating, instantiating and attaching a semaphore to a process. OpenBSD system calls proved the calling process as one of the arguments, so we already know with which process to attach the semaphore. This function takes in the semaphore name and initial count as arguments, and instantiates the semaphore with those user supplied values. Naturally, we can't trust the user with providing the correct argument format, so we check to make sure that the name and count meet the necessary conditions. We would also make sure that the calling process does not already own a semaphore with that name, otherwise we return an error to the user. After checking all criteria are met, we now commit to creating the semaphore and allocate memory space for it. If there is enough memory available, we create the semaphore and append the calling process to the semaphore. We also add the semaphore to the calling process's semaphore list, after instantiating and setting all its variables.

The free_semaphore() function removes a specified semaphore and releases memory the semaphore occupied. A process will only be able to successfully perform this operation on an existing semaphore that the process can access (inherited or owned sempahores – uses the find function that I describe shortly). If the process can not find such semaphore, nothing is done and an error is returned to the user. If such semaphore is found, the semaphore is removed from the owner's semaphore list. Note that since the children of a process do not actually get copies of the semaphores they inherit, removing the semaphore form the owner process is enough. If the children were to have copies of the inherited semaphore, we'd have to traverse all the children and delete all instances/pointers to this semaphore to ensure stability, which is computationally expensive. Hence, another reason why this implementation method is used.

Before discussing the remaining system calls, let's take a look at the helper find() function that the system calls use to get the semaphore of choice.

The find function is a recursive function that traverses the list of semaphores a process owns, and checks if a semaphore of specified name exists in that list. If the semaphore is not present, then the find function traverses the process's parent's list. If not found, it keeps on going up the process tree until one of two things happen: it finds the semaphore it was looking for, or it finds a process that has neither inherited semaphores nor owns semaphores – the "root" process. This traversal guarantees that the first semaphore we encounter will be the semaphore that is created by the most recently created process in the list. Note that this traversal is strictly going up the tree, and doesn't check the branches (sibling processes), nor does it go down into the children processes.

The remaining two system calls are the up_semaphore() and the down_semaphore() functions. They both take in the name of a semaphore as the argument. They both use the find function to determine if such semaphore exists, before starting their section. Both functions use the lock on their mutex before going into their critical section, and unlock the mutex on exit. Furthermore, the down_semaphore() operation

releases the mutex lock right before going to sleep, and requires the lock upon waking up.

The down_semaphore() function will decrement the count of the semaphore, indicating that a process is requesting access. If the count is less than 0, then its an indicator that resources are not available, hence the process goes to sleep. Whenever a process sleeps, it does so on a condition. This condition could be an arbitrary object, and the process is awakened when a wakeup signal is sent on the "condition" it went to sleep. Since we want to ensure the semaphore is fair, whenever a process sleeps, it is put on a wait queue and the process that is first in the list (longest waiting) is to be woken up first. But, if we were to make all processes sleep on the same "condition", then a wakeup signal might cause unwanted processes to wakeup our of order. To overcome this, we make a process sleep on itself. Therefore, when we call wakeup on the process address, the process itself is woken up. This is an easy way of ensuring that we're waking up the process we intend to wakeup, and no other process. Once a process has gone to sleep, it waits until another process calls up_semaphore on the semaphore it slept.

The up_semaphore function is conceptually simple. After doing basic check and acquiring the lock, it increments the semaphore count. If the count was below zero, that means a process is waiting on the semaphore, so a wakeup is issued. To ensure that the wakeup is done on the longest waiting semaphore, we simply check to see which process is first in the wait queue of the semaphore. Once we figure that out, we just issue a wakeup on that process to wake it up (remember: a process sleeps on itself, so it should be woken up by issuing a wakeup on itself).
After a process is woken up, we free up the memory that it was occupying as a "node" in the wait queue. We then release the mutex lock we originally acquired and exit.

Section IV: part5.txt

```
Script started on Tue Nov 24 11:54:43 2015
# sh

#

#

# cd /usr/src/sys/kern

#

#

# ls -lt | head

total 15812
-rw-r--r--  1 root  wsrc    13315 Nov 24 10:11 cop4600.c
-rw-r--r--  1 root  wsrc    11195 Nov 24 10:11 kern_fork.c
-rw-r--r--  1 root  wsrc    14643 Nov 24 10:11 kern_exit.c
-rw-r--r--  1 root  wsrc    26655 Nov 24 10:00 init_sysent.c
-rw-r--r--  1 root  wsrc    15707 Nov 24 10:00 syscalls.c
-rw-r--r--  1 root  wsrc    22379 Nov 24 10:00 syscalls.master
-rw-r--r--  1 root  wsrc    11616 Nov 24 09:16 kern_proc.c
-rw-------  1 root  wsrc  6485068 Nov 23 18:57 emacs.core
-rw-r--r--  1 root  wsrc     8424 Nov 23 18:56 cop4600.c~
#

#

# tail syscalls.master

;============================================================================

;added by Dave Small
289     STD             { int sys_hello( void ); }
290     STD             { int sys_showargs( const char *str, int val ); }
291     STD     { int sys_cipher (char *text, int lkey, int nkey); }
292     STD             { int sys_allocate_semaphore (const char *name, int
initial_count); }
293     STD             { int sys_down_semaphore (const char *name); }
294     STD             { int sys_up_semaphore (const char *name); }
295     STD             { int sys_free_semaphore (const char *name); }
#

#

# grep semaphore *

cop4600.c://extern int sys_semaphores;             /* system wide semaphores */
cop4600.c://extern lock_data_t control;            /* lock for updating
sys_semaphores */
cop4600.c:semaphore_t* find_semaphore(struct proc *p, char *kname);
cop4600.c: * Create and initialize semaphore: 292
cop4600.c:sys_allocate_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c:  struct sys_allocate_semaphore_args *uap = v;
cop4600.c:  semaphore_t *sem;
cop4600.c:  LIST_FOREACH(sem, &(p->semaphores), s_next)
cop4600.c:      return EEXIST;   /* process owns semaphore with that name */
cop4600.c:  /* allocate memeory for semaphore right now */
cop4600.c:  sem = (struct semaphore*) malloc(sizeof(struct semaphore), M_PROC,
M_NOWAIT);
cop4600.c:  /* initialize semaphore */
```

```
cop4600.c:  lockinit(&sem->mutex, p->p_priority,"semaphore: another process in
critical section", 0, LK_CANRECURSE);
cop4600.c:  LIST_INSERT_HEAD(&p->semaphores, sem, s_next);
cop4600.c:  //++sys_semaphores;
cop4600.c:sys_down_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c:  struct sys_down_semaphore_args *uap = v;
cop4600.c:  semaphore_t *sem;
cop4600.c:  /* Get semaphore */
cop4600.c:  sem = find_semaphore(p, kname);
cop4600.c:      flag = tsleep((void*) np->p, p->p_priority,"waiting on
semaphore",0);
cop4600.c:sys_up_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c:  struct sys_up_semaphore_args *uap = v;
cop4600.c:  semaphore_t *sem;
cop4600.c:  /* Get semaphore */
cop4600.c:  sem = find_semaphore(p, kname);
cop4600.c: * Delete semaphore: 295
cop4600.c:sys_free_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c:  struct sys_free_semaphore_args *uap = v;
cop4600.c:  semaphore_t *sem;
cop4600.c:  sem = find_semaphore(p, kname);
cop4600.c:    return ENOENT;    /* process doesn't own such semaphore */
cop4600.c:  //--semaphore_count;
cop4600.c:/* Get semaphore with presedence according to creation - traverse the
process tree bottom-up */
cop4600.c:semaphore_t* find_semaphore(struct proc *p, char *kname)
cop4600.c:  semaphore_t *sem;
cop4600.c:  /* Go through all the semaphores I created */
cop4600.c:  while((LIST_EMPTY(&p_find->semaphores) == FALSE || p_find->inherited ==
TRUE) && breakloop == FALSE)
cop4600.c:    /* Until I get to a process that neither has created semaphore or
inherited them */
cop4600.c:    LIST_FOREACH(sem, &p_find->semaphores, s_next)
cop4600.c:      /* Search through the process's semaphore list for such name */
cop4600.c:        break;    /* found semaphore */
cop4600.c:  /* If semaphore is null at this point, then no semaphore has been found
for the process */
cop4600.c~: * System wide list of semaphores (max 64)
cop4600.c~:typedef struct semaphore {
cop4600.c~:  struct proc *owner;               /* process that created the
semaphore */
cop4600.c~:  char *name;                       /* string name of semaphore */
cop4600.c~:  int count;                        /* control variable of semaphore */
cop4600.c~:  SIMPLEQ_HEAD(list,node) head;     /* list of processes waiting on
semaphore */
cop4600.c~:} semaphore_t;
cop4600.c~:semaphore_t* find_semaphore(struct proc *p, char *name);
cop4600.c~:int has_semaphore(struct proc *p, char *name);
cop4600.c~:int owns_semaphore(struct proc *p, char *name);
cop4600.c~:int access_semaphore(struct proc *p, char *name);
cop4600.c~: * Create and initialize semaphore
cop4600.c~:sys_allocate_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c~:  uprintf("allocate_semaphore\n");
cop4600.c~:sys_up_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c~:  uprintf("up_semaphore\n");
cop4600.c~:sys_down_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c~:  uprintf("down_semaphore\n");
cop4600.c~: * Delete semaphore
cop4600.c~:sys_free_semaphore (struct proc *p, void *v, register_t *retval)
cop4600.c~:  uprintf("free_semaphore\n");
Binary file emacs.core matches
init_main.c:    /* Initialize System V style semaphores. */
init_sysent.c:  { 2, s(struct sys_allocate_semaphore_args),
```

```
init_sysent.c:      sys_allocate_semaphore },           /* 292 = allocate_semaphore
*/
init_sysent.c:  { 1, s(struct sys_down_semaphore_args),
init_sysent.c:      sys_down_semaphore },               /* 293 = down_semaphore */
init_sysent.c:  { 1, s(struct sys_up_semaphore_args),
init_sysent.c:      sys_up_semaphore },                 /* 294 = up_semaphore */
init_sysent.c:  { 1, s(struct sys_free_semaphore_args),
init_sysent.c:      sys_free_semaphore },               /* 295 = free_semaphore */
kern_exit.c:    semaphore_t *sem;
kern_exit.c:    LIST_FOREACH(sem, &p->semaphores, s_next)       /* For each
semaphore process created */
kern_exit.c:            while(SIMPLEQ_EMPTY(&sem->p_head) == 0)  /* At least one
process is waiting on semaphore */
kern_fork.c:    LIST_INIT(&p2->semaphores);
kern_fork.c:    if (LIST_EMPTY(&p1->semaphores))        /* nothing to inherit */
kern_fork.c:    else                                                    /*
child should inherit parent's semaphores */
syscalls.c:     "allocate_semaphore",                   /* 292 = allocate_semaphore
*/
syscalls.c:     "down_semaphore",                       /* 293 = down_semaphore */
syscalls.c:     "up_semaphore",                 /* 294 = up_semaphore */
syscalls.c:     "free_semaphore",                       /* 295 = free_semaphore */
syscalls.master:292     STD             { int sys_allocate_semaphore (const char
*name, int initial_count); }
syscalls.master:293     STD             { int sys_down_semaphore (const char *name);
}
syscalls.master:294     STD             { int sys_up_semaphore (const char *name); }
syscalls.master:295     STD             { int sys_free_semaphore (const char *name);
}
sysv_sem.c: * Implementation of SVID semaphores
sysv_sem.c:struct       semid_ds **sema;        /* semaphore id list */
sysv_sem.c:      * Preallocate space for the new semaphore.  If we are going
sysv_sem.c:      * condition in allocating a semaphore with a specific key.
sysv_sem.c:                     DPRINTF(("not enough semaphores left (need %d, got
%d)\n",
sysv_sem.c:              * Make sure that the semaphore still exists
sysv_sem.c:              * The semaphore is still alive.  Readjust the count of
sysv_sem.c:                      * rollback the semaphore ups and down so we can
return
sysv_sem.c:     /* Do a wakeup if any semaphore was up'd. */
sysv_sem.c: * semaphores.
#

#

# cd ../arch/i386/compile/GENERIC

#

#

# grep semaphore *

Binary file cop4600.o matches
Binary file init_sysent.o matches
param.c: * Values in support of System V compatible semaphores.
param.c:        SEMMNI,         /* # of semaphore identifiers */
param.c:        SEMMNS,         /* # of semaphores in system */
param.c:        SEMMSL,         /* max # of semaphores per id */
param.c:        SEMVMX,         /* semaphore maximum value */
#

#
```

```
# grep cop4600 Makefile

OBJS=   cop4600.o \
CFILES= $S/kern/cop4600.c \
#

#

# tail Makefile


uscanner.o: $S/dev/usb/uscanner.c
        ${NORMAL_C}

usscanner.o: $S/dev/usb/usscanner.c
        ${NORMAL_C}

if_wi_usb.o: $S/dev/usb/if_wi_usb.c
        ${NORMAL_C}

#

#

# cd /root

#

#

# ls -lt

total 288
-rw-r--r--  1 root  wheel   8756 Nov 24 11:56 part5.txt
-rwxr-xr-x  1 root  wheel  14127 Nov 24 11:52 semtest
drwx------  2 root  wheel    512 Nov 24 10:15 .vnc
-rw-------  1 root  wheel    333 Nov 24 10:15 .Xauthority
-rw-r--r--  2 root  wheel    490 Nov 24 10:03 .profile
drwxr-xr-x  6 root  wheel    512 Nov 24 08:53 Project
-rwxrwxrwx  1 root  wheel     69 Nov 23 02:59 vncup1920
-rw-r--r--  1 root  wheel     64 Nov 22 10:58 XTerm
-rw-r--r--  1 root  wheel     48 Nov 22 10:57 XTerm~
-rw-------  1 root  wheel  76488 Nov 22 09:11 mbox
-rw-r--r--  1 root  wheel    676 Jun  9  2006 kerntest.c
-rwxr-xr-x  1 root  wheel   6536 Jun  9  2006 kerntest
drwx------  2 root  wheel    512 Jun  9  2006 .ssh
-rwxr-xr-x  1 root  wheel     67 Jun  9  2006 vncup800
-rwxr-xr-x  1 root  wheel     68 Jun  9  2006 vncup1024
-rwxr-xr-x  1 root  wheel     69 Jun  9  2006 vncup1280
-rw-r--r--  1 root  wheel    633 Jun  9  2006 .fonts.cache-1
drwxr-xr-x  3 root  wheel    512 Jun  9  2006 .emacs.d
-rw-r--r--  1 root  wheel    482 Jun  9  2006 .emacs
-rw-r--r--  2 root  wheel    769 Jun  9  2006 .cshrc
-rw-------  1 root  wheel    125 Mar 29  2004 .klogin
-rw-r--r--  1 root  wheel    335 Mar 29  2004 .login
#

#

# gcc -o semtest Project/5/kerntest.c

#
```

```
#

# ./semtest


=============== START SEMAPHORE TEST ===============

|-------=================================|
| KEYS:                                  |
| > creating semaphore (name, count)     |
| > removing semaphore (name)            |
| > up on semaphore (name)               |
| > down on semaphore (name)             |
|--------=================================|

___________________ PART 1: BASIC CALLS ____________________
CREATE SEMAPHORE:
creating semaphore (Sem1, 0) .... SUCCESS
creating semaphore (Sem1, 0) .... ERROR: EEXIST
creating semaphore (abcdefghijklmnopqrstuvwxyz01234567890, 0) .... ERROR:
ENAMETOOLONG
creating semaphore (Sem_negative, -1) .... ERROR: EDOM
UP SEMAPHORE:
up on semaphore (Sem1) .... SUCCESS
up on semaphore (Sem_noexist) .... ERROR: ENOENT
DELETE:
removing semaphore (Sem1) .... SUCCESS
removing semaphore (Sem_noexist) .... ERROR: ENOENT
removing semaphore (Sem1) .... ERROR: ENOENT
UP SEMAPHORE:
up on semaphore (Sem1) .... ERROR: ENOENT
up on semaphore (abcdefghijklmnopqrstuvwxyz01234567890) .... ERROR: ENOENT
___________________ END PART 1 _____________________________

___________________ PART 2: INHERITANCE ____________________
creating semaphore (Sem_P, 0) .... SUCCESS
Parent about to fork .... (1)
Parent about to fork .... (2)
Child 1: START
creating semaphore (Sem_C1, 0) .... SUCCESS
creating semaphore (Sem_P, 0) .... SUCCESS
DOWN SEMAPHORE (Child 1):
down on semaphore (Sem_random) .... ERROR: ENOENT
Child 1: sleep for a seconds
Child 2: START
creating semaphore (Sem_C2, 0) .... SUCCESS
DOWN SEMAPHORE (Child 2):
Child 1: wakeup
down on semaphore (Sem_C2) .... ERROR: ENOENT
Child 1: sleep for 2 seconds
Child 1: wakeup
Child 1 (NOTE): At this point Child 2 went down() on inherited Sem_P. Child 1 has
created it's own Sem_P
UP SEMAPHORE (Child 1):
up on semaphore (Sem_P) .... SUCCESS
Let's see if Child 2 wakes up...
Child 1:  sleep for 5 seconds
Child 1: wakeup
REMOVE SEMAPHORE (Child 1):
removing semaphore (Sem_P) .... SUCCESS
Let's try waking up Child 2 one more time by calling up() on Sem_P
UP SEMAPHORE (Child 1):
```

```
down on semaphore (Sem_P) .... SUCCESS
up on semaphore (Sem_P) .... SUCCESS
Child 2: completed down
up on semaphore (Sem_P) .... SUCCESS
Child 2: sleep for 2 second
up on semaphore (Sem_P) .... SUCCESS
up on semaphore (Sem_P) .... SUCCESS
up on semaphore (Sem_P) .... SUCCESS
Child 1: sleep for 5 second
Child 2: wakeup
DOWN SEMAPHORE (Child 2):
down on semaphore (Sem_P) .... SUCCESS
down on semaphore (Sem_P) .... SUCCESS
down on semaphore (Sem_P) .... SUCCESS
down on semaphore (Sem_P) .... SUCCESS
FREE SEMAPHORE (Child 2):
removing semaphore (Sem_P) .... SUCCESS
removing semaphore (Sem_C1) .... ERROR: ENOENT
removing semaphore (Sem_C2) .... SUCCESS
Child 2: END
Child 1: wakeup
FREE SEMAPHORE (Child 1):
removing semaphore (Sem_P) .... ERROR: ENOENT
removing semaphore (Sem_C1) .... SUCCESS
removing semaphore (Sem_C2) .... ERROR: ENOENT
Child 1: END
___________________ END PART 2 _______________________________

___________________ PART 3: FAIRNESS _____________________
creating semaphore (Fair, 0) .... SUCCESS
Parent about to fork .... (1)
Parent about to fork .... (2)
Child 1: DOWN SEMAPHORE
Parent about to fork .... (3)
Child 2: DOWN SEMAPHORE
Parent about to fork .... (4)
Child 3: DOWN SEMAPHORE
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 1: completed down .... exiting
Child 4: wakeup
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 2: completed down .... exiting
Child 4: wakeup
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 3: completed down .... exiting
Child 4: wakeup .... exiting
___________________ END PART 3 _______________________________

___________________ PART 4: FREE ON EXIT _____________________
creating semaphore (Sem A, 0) .... SUCCESS
creating semaphore (Sem B, 0) .... SUCCESS
Parent about to fork .... (1)
Parent: sleep for 2 seconds
Child: START
up on semaphore (Sem A) .... SUCCESS
up on semaphore (Sem B) .... SUCCESS
up on semaphore (Sem Control) .... ERROR: ENOENT
```

```
Child: sleep for 3 seconds
Parent: END
# Child: wakeup
up on semaphore (Sem A) .... ERROR: ENOENT
up on semaphore (Sem B) .... ERROR: ENOENT
up on semaphore (Sem Control) .... ERROR: ENOENT
Child: END
____________________ END PART 4 _______________________________

============== END SEMAPHORE TEST ======================


#

# ls -lt /

total 20080
drwx------   6 root  wheel      512 Nov 24 11:58 root
drwxrwxrwt   4 root  wheel      512 Nov 24 11:57 tmp
drwxr-xr-x   3 root  wheel    19968 Nov 24 10:14 dev
-rwxr-xr-x   1 root  wsrc   5077982 Nov 24 10:13 bsd
-rw-r--r--   2 root  wheel      490 Nov 24 10:03 .profile
drwxr-xr-x  18 root  wheel     2048 May 20  2008 etc
-rw-r--r--   2 root  wheel      769 Jun  9  2006 .cshrc
-rw-r--r--   1 root  wheel  5075323 Jul  7  2005 bsd.0
-rw-r--r--   1 root  wheel    42132 Jul 18  2004 boot
lrwxr-xr-x   1 root  wheel       11 Jul 18  2004 sys -> usr/src/sys
drwxr-xr-x   2 root  wheel     2048 Mar 29  2004 sbin
drwxr-xr-x   2 root  wheel     1024 Mar 29  2004 bin
drwxr-xr-x   2 root  wheel      512 Mar 29  2004 altroot
drwxr-xr-x   2 root  wheel      512 Mar 29  2004 home
drwxr-xr-x   2 root  wheel      512 Mar 29  2004 mnt
drwxr-xr-x   2 root  wheel      512 Mar 29  2004 stand
drwxr-xr-x  22 root  wheel      512 May 16  2003 var
drwxr-xr-x  16 root  wheel      512 May 16  2003 usr
#

#

# dmest   g | head

00 rawdev=0xd02
uhidev0 at uhub0 port 1 configuration 1 interface 0
uhidev0: VMware VMware Virtual USB Mouse, rev 1.10/1.03, addr 2, iclass 3/1
ums0 at uhidev0: 7 buttons and Z dir.
wsmouse1 at ums0 mux 0
uhub1 at uhub0 port 2
uhub1: vendor 0x0e0f VMware Virtual USB Hub, class 9/0, rev 1.10/1.00, addr 3
uhub1: 7 ports with 7 removable, self powered
ugen0 at uhub1 port 1
ugen0: VMware Virtual Bluetooth Adapter, rev 2.00/1.00, addr 4
#

#

# #
Script done on Tue Nov 24 11:58:39 2015
```

Section V: Novel and modified source code

Subsection V: cop4600.c

```
/*      $OpenBSD: cop4600.c,v 1.00 2003/07/12 01:33:27 dts Exp $        */

#include <sys/param.h>
#include <sys/acct.h>
#include <sys/systm.h>
#include <sys/ucred.h>
#include <sys/proc.h>
#include <sys/timeb.h>
#include <sys/times.h>
#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/filedesc.h>
#include <sys/pool.h>
#include <sys/queue.h>
#include <sys/mount.h>
#include <sys/syscallargs.h>

/*============================================================================**
**  Dave's example system calls                                               **
**============================================================================*/

/*
** hello() prints to the tty a hello message and returns the process id
*/

int
sys_hello( struct proc *p, void *v, register_t *retval )
{
  uprintf( "\nHello, process %d!\n", p->p_pid );
  *retval = p->p_pid;

  return (0);
}

/*
** showargs() demonstrates passing arguments to the kernel
*/

#define MAX_STR_LENGTH  1024

int
sys_showargs( struct proc *p, void *v, register_t *retval )
{
  /* The arguments are passed in a structure defined as:
  **
  **  struct sys_showargs_args
  **  {
  **      syscallarg(char *) str;
  **      syscallarg(int)    val;
  **  }
  */

  struct sys_showargs_args *uap = v;

  char kstr[MAX_STR_LENGTH+1]; /* will hold kernal-space copy of uap->str */
  int err = 0;
  int size = 0;

  /* copy the user-space arg string to kernal-space */

  err = copyinstr( SCARG(uap, str), &kstr, MAX_STR_LENGTH, &size );
  if (err == EFAULT)
    return( err );
```

```
  uprintf( "The argument string is \"%s\"\n", kstr );
  uprintf( "The argument integer is %d\n", SCARG(uap, val) );
  *retval = 0;

  return (0);
}

/*============================================================================**
**  Dawit's COP4600 2004C system calls                                        **
**============================================================================*/


/* --------- SYSTEM CALL --------- */

#define min(a,b) (((a) > (b)) ? (b) : (a))  // compute minimum value

void substitution (char *text, int textLength, int lkey, int nkey);
void transposition (char *text, int textLength, int lkey, int nkey);

/*
** TigerCipher® : Protecting us against canine digital attack
** Note: text has to be mutable, and not a string literal assigned
** to char*
*/

int
sys_cipher (struct proc *p, void *v, register_t *retval)
{
  int err = 0;
  size_t textSize;

  struct sys_cipher_args *args = v;

  char ktext[MAX_STR_LENGTH + 1];    /* kernel space copy of text */
  int klkey;                         /* kernel copy of lkey */
  int knkey;                         /* kernel copy of nkey */

  /* Copy text to kernel space */
  err = copyinstr( SCARG(args, text), &ktext, MAX_STR_LENGTH, &textSize);
  if(err == EFAULT)
    return (err);

  /* Copy values to kernel space */
  klkey = SCARG(args, lkey);
  knkey = SCARG(args, nkey);

  // Pass 1
  substitution(ktext, textSize, klkey, knkey);

  // Pass 2
  transposition(ktext, textSize, klkey, knkey);

  /* Copy text to user space */
  err = copyoutstr(&ktext, SCARG(args, text), MAX_STR_LENGTH+1 , &textSize);
  if(err == EFAULT)
    return(err);

  *retval = textSize;

  return (0);
}
```

```
/* --------- END SYSTEM CALL --------- */

/* --------- HELPER FUCNTION --------- */

/* Mutate characters by by shifting their ascii values */
void substitution (char *text, int textLength, int lkey, int nkey)
{
  /* These variables change every loop iteration */
  char c;                                          // temp char
  int i;                                           // loop counter
  int x, offset;                                   // shift values
  int upperCaseShift, lowecaseShift, digitShift;  // precomputed shifts
  int primaryCondition, secondaryCondtion;         // conditions

  /* Precompute variables constant throughout loop iteration */
  upperCaseShift = ((lkey % 26) - 'A' + 26);
  lowecaseShift = ((lkey % 26) - 'a' + 26);
  digitShift = ((nkey % 10) - '0' + 10);
  primaryCondition = ((lkey < 0) && (lkey & 0x1));

  // for each character c in text, perform the appropriate computation
    for(i = 0; i < textLength; ++i)
    {
      c = text[i];
      // c is uppercase
      if(c >= 'A' && c <= 'Z')
      {
        x = (c + upperCaseShift) % 26;
        secondaryCondtion = ((x - 'A') & 0x1);

        /*
        * conditional shortcut (??)
        *
        * The following double conditional statement:
        * ((a < 0) && (b & 0x1)) ?
        *     (c & 0x1) ? X : Y ) :
        *   (c & 0x1) ? Y : X )
        *
        * can be converted to an equivalent biconditional statement of the form:
        * (((a < 0) && (b & 0x1)) IFF (c & 0x1))) ?
        *      X : Y
        *
        *   Note: The conditionals in this form will always evaluate to 'true' = 1
        * and not some random non-zero integer, because (some_number & 0x1) always
        *   evaluates to either 0 or 1
        *
        *   Therefore, we can represent IFF as:
        * (EXPRESSION1 == EXPRESSION2) ? TRUE : FALSE
        */
        offset = ( (primaryCondition == secondaryCondtion) ? 'A' : 'a' );
        text[i] = x + offset;
      }
      // c is lowecase
      else if (c >= 'a' && c <= 'z')
      {
        x = (c + lowecaseShift) % 26;
        secondaryCondtion = ((x - 'a') & 0x1);
        offset = ( (primaryCondition == secondaryCondtion) ? 'a' : 'A' );

        text[i] = x + offset;
      }
      // c is digit
      else if (c >= '0' && c <= '9')
```

```
      {
        text[i] = ((c + digitShift) % 10) + '0';
      }
      // c is something else: text[i] = c
    }

}

/* Split text into quads and transpose elements */
void transposition (char *text, int textLength, int lkey, int nkey)
{
  // detemine extra chars at the end
  int i;                            // iterator
  char temp;                        // swap space
  int nonQuad = textLength % 4;     // quad length < 4

  // process each normal quad with length = 4
  for(i = 0; i < (textLength - nonQuad); i+=4)
  {
    // swap 1st and 3th chars
    temp = text[i];
    text[i] = text[i+2];
    text[i+2] = temp;

    // swap 2nd and 4th chars
    temp = text[i+1];
    text[i+1] = text[i+3];
    text[i+3] = temp;
  }

  /*
   * This will only be evaluated when the last "quad" of the
   * string contains a quad of length 1 to 3
   */

  if(nonQuad)
  {
    /* i is now at the 1st index of this weird "quad" */
    if (nonQuad == 3)
    {
      // swap 1st and 3rd chars
      temp = text[i];
      text[i] = text[i+2];
      text[i+2] = temp;
    }
    else if (nonQuad == 2)
    {
      // swap 1st and 2nd chars
      temp = text[i];
      text[i] = text[i+1];
      text[i+1] = temp;
    }
      /* leave the lonely char alone */
  }
}

/* --------- END HELPER FUNCTION --------- */

/*============================================================================**
**  Part 5: Semahores                                                         **
**============================================================================*/

#define EQUAL 0                         /* for strcmp */
```

```
#define FALSE 0
#define TRUE 1
//#define MAX_SEM_COUNT 64               /* max allowed sempahores system wide */

//extern int sys_semaphores;             /* system wide semaphores */
//extern lock_data_t control;            /* lock for updating sys_semaphores */

#define COPYNAME(kname, uap, length) do { \
    if (copyinstr(SCARG(uap,name), &kname, MAX_NAME_LENGTH, &length) == EFAULT) \
    {return EFAULT;}  \
} while(0)

#define NAMECHECK(kname, length, err) do {  \
    if (length == MAX_NAME_LENGTH && kname[MAX_NAME_LENGTH-1] != '\0')  \
    {return err;} \
} while (0)


/* helper functions */
semaphore_t* find_semaphore(struct proc *p, char *kname);

/*
 * Create and initialize semaphore: 292
 */
int
sys_allocate_semaphore (struct proc *p, void *v, register_t *retval)
{
  struct sys_allocate_semaphore_args *uap = v;
  semaphore_t *sem;
  char kname[MAX_NAME_LENGTH];
  int kcount;
  int length;

  length = 0;

  COPYNAME(kname, uap, length);
  NAMECHECK(kname, length, ENAMETOOLONG);
  LIST_FOREACH(sem, &(p->semaphores), s_next)
    if (strcmp(sem->name, kname) == EQUAL && sem->owner == p)
      return EEXIST;   /* process owns semaphore with that name */

  kcount = SCARG(uap, initial_count);
  if (kcount < 0)
    return EDOM;            /* out of range */

  /* allocate memeory for semaphore right now */
  sem = (struct semaphore*) malloc(sizeof(struct semaphore), M_PROC, M_NOWAIT);
  if (sem == NULL)
    return ENOMEM;     /* not enough memeory */

  /* initialize semaphore */
  if (copystr(&kname, &sem->name, MAX_NAME_LENGTH, &length) == EFAULT)
  {
    /* something bad happaned. abort*/
    free(sem, M_PROC);
    return EFAULT;
  }
  sem->owner = p;
  sem->count = kcount;
  SIMPLEQ_INIT(&(sem->p_head));
  lockinit(&sem->mutex, p->p_priority,"semaphore: another process in critical
section", 0, LK_CANRECURSE);
  LIST_INSERT_HEAD(&p->semaphores, sem, s_next);
```

```
  //++sys_semaphores;
  return(0);
}

/*
 * Semaphore down: 293
 */

int
sys_down_semaphore (struct proc *p, void *v, register_t *retval)
{
  struct sys_down_semaphore_args *uap = v;
  semaphore_t *sem;
  char kname[MAX_NAME_LENGTH];
  int length;
  int flag;
  struct p_node *np;

  length = 0;
  flag = 1;

  /* Get semaphore */
  COPYNAME(kname, uap, length);
  NAMECHECK(kname, length, ENOENT);
  sem = find_semaphore(p, kname);
  if(sem == NULL)
    return ENOENT;

  lockmgr(&sem->mutex, LK_EXCLUSIVE, NULL, p);  /* Lock mutex */
  --sem->count;
  if(sem->count < 0)
  {
    /* create and instantiate node of SIMPLEQ */
    np = (struct p_node*) malloc (sizeof(struct p_node), M_PROC, M_NOWAIT);
    if(np == NULL)
      return ENOMEM;
    np->p = p;
    /*
     * Make a process sleep on itself. This way, we wont have to worry about
     * notifying other processes upon wakeup. Each process will sleep on a
     * unique "object" i.e itself
     */
    SIMPLEQ_INSERT_TAIL(&sem->p_head, np, p_next);   /* add process to wait queue */
    lockmgr(&sem->mutex, LK_RELEASE, NULL, p);       /* release lock before sleeping
*/

    while (flag != 0)         /* break out of this loop only if wakeup() is called
*/
      flag = tsleep((void*) np->p, p->p_priority,"waiting on semaphore",0);

    lockmgr(&sem->mutex, LK_EXCLUSIVE, NULL, p);     /* lock mutex */
  }
  lockmgr(&sem->mutex, LK_RELEASE, NULL, p);      /* Unlock mutex */
  return(0);
}

/*
 * Semaphore up: 294
 */

int
sys_up_semaphore (struct proc *p, void *v, register_t *retval)
{
```

```
  struct sys_up_semaphore_args *uap = v;
  semaphore_t *sem;
  char kname[MAX_NAME_LENGTH];
  int length;
  struct p_node *np;

  length = 0;

  /* Get semaphore */
  COPYNAME(kname, uap, length);
  NAMECHECK(kname, length, ENOENT);

  sem = find_semaphore(p, kname);
  if(sem == NULL)
    return ENOENT;

  lockmgr(&sem->mutex, LK_EXCLUSIVE, NULL, p);  /* Lock mutex */
  ++sem->count;

  if(sem->count <= 0)
  {
    /* Signal first process in wait list */
    np = SIMPLEQ_FIRST(&sem->p_head);
    wakeup((void*) np->p);
    SIMPLEQ_REMOVE_HEAD(&sem->p_head, np, p_next);  /* delete node */
    free(np, M_PROC);                               /* free memory */
  }
  /* Unlock mutex */
  lockmgr(&sem->mutex, LK_RELEASE, NULL, p);
  return(0);
}

/*
 * Delete semaphore: 295
 */

int
sys_free_semaphore (struct proc *p, void *v, register_t *retval)
{

  struct sys_free_semaphore_args *uap = v;
  semaphore_t *sem;
  char kname[MAX_NAME_LENGTH];
  int length;

  length = 0;

  /* if name is not in proper format, don't bother checking */
  COPYNAME(kname, uap, length);
  NAMECHECK(kname, length, ENOENT);

  sem = find_semaphore(p, kname);
  if(sem == NULL)
    return ENOENT;    /* process doesn't own such semaphore */

  LIST_REMOVE(sem, s_next);   /* Remove from system list*/
  /* Delete all internals */
  /* Do I need to empty the queue? WHEN? HOW?* --- SEE DAVE'S COMMENT ON HINTS?*/
  lockmgr(&sem->mutex, LK_DRAIN, NULL, p);    /* drain lock */
  free(sem, M_PROC);                          /* Free memory */
  //--semaphore_count;
  return(0);
}
```

```
/* Get semaphore with presedence according to creation - traverse the process tree
bottom-up */
semaphore_t* find_semaphore(struct proc *p, char *kname)
{
  semaphore_t *sem;
  struct proc *p_find;      /* Process to search through */
  int breakloop;            /* flag to exit */

  p_find = p;             /* Start with current process */
  sem = NULL;
  breakloop = FALSE;

  /* Go through all the semaphores I created */
  while((LIST_EMPTY(&p_find->semaphores) == FALSE || p_find->inherited == TRUE) &&
breakloop == FALSE)
  {
    /* Until I get to a process that neither has created semaphore or inherited them
*/
    LIST_FOREACH(sem, &p_find->semaphores, s_next)
    {
      /* Search through the process's semaphore list for such name */
      if(strcmp(sem->name, kname) == EQUAL && sem->owner == p_find)
      {
        breakloop = TRUE;
        break;    /* found semaphore */
      }
    }
    p_find = p_find->p_pptr;  /* repate process in parent */
  }
  /* If semaphore is null at this point, then no semaphore has been found for the
process */
  return sem;
}
```

Subsection V: syscalls.master

```
;	$OpenBSD: syscalls.master,v 1.68 2004/02/28 19:44:16 miod Exp $
;	$NetBSD: syscalls.master,v 1.32 1996/04/23 10:24:21 mycroft Exp $

;	@(#)syscalls.master 8.2 (Berkeley) 1/13/94

; OpenBSD system call name/number "master" file.
; (See syscalls.conf to see what it is processed into.)
;
; Fields: number type [type-dependent ...]
;	number system call number, must be in order
;	type	one of STD, OBSOL, UNIMPL, NODEF, NOARGS, or one of
;		the compatibility options defined in syscalls.conf.
;
; types:
;	STD	always included
;	OBSOL	obsolete, not included in system
;	UNIMPL	unimplemented, not included in system
;	NODEF	included, but don't define the syscall number
;	NOARGS	included, but don't define the syscall args structure
;	INDIR	included, but don't define the syscall args structure,
;		and allow it to be "really" varargs.
;
; The compat options are defined in the syscalls.conf file, and the
; compat option name is prefixed to the syscall name.  Other than
; that, they're like NODEF (for 'compat' options), or STD (for
; 'libcompat' options).
;
; The type-dependent arguments are as follows:
; For STD, NODEF, NOARGS, and compat syscalls:
;	{ pseudo-proto } [alias]
; For other syscalls:
;	[comment]
;
; #ifdef's, etc. may be included, and are copied to the output files.
; #include's are copied to the syscall switch definition file only.

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/signal.h>
#include <sys/mount.h>
#include <sys/syscallargs.h>
#include <sys/poll.h>
#include <sys/event.h>
#include <xfs/xfs_pioctl.h>

; Reserved/unimplemented system calls in the range 0-150 inclusive
; are reserved for use in future Berkeley releases.
; Additional system calls implemented in vendor and other
; redistributions should be placed in the reserved range at the end
; of the current calls.

0	INDIR		{ int sys_syscall(int number, ...); }
1	STD		{ void sys_exit(int rval); }
2	STD		{ int sys_fork(void); }
3	STD		{ ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4	STD		{ ssize_t sys_write(int fd, const void *buf, \
				size_t nbyte); }
5	STD		{ int sys_open(const char *path, \
				int flags, ... int mode); }
6	STD		{ int sys_close(int fd); }
7	STD		{ pid_t sys_wait4(pid_t pid, int *status, int options, \
				struct rusage *rusage); }
8	COMPAT_43	{ int sys_creat(const char *path, int mode); } ocreat
```

```
9      STD          { int sys_link(const char *path, const char *link); }
10     STD          { int sys_unlink(const char *path); }
11     OBSOL        execv
12     STD          { int sys_chdir(const char *path); }
13     STD          { int sys_fchdir(int fd); }
14     STD          { int sys_mknod(const char *path, int mode, \
                        dev_t dev); }
15     STD          { int sys_chmod(const char *path, int mode); }
16     STD          { int sys_chown(const char *path, uid_t uid, \
                        gid_t gid); }
17     STD          { int sys_obreak(char *nsize); } break
18     COMPAT_25    { int sys_getfsstat(struct statfs *buf, long bufsize, \
                        int flags); } ogetfsstat
19     COMPAT_43    { long sys_lseek(int fd, long offset, int whence); } \
                        olseek
20     STD          { pid_t sys_getpid(void); }
21     STD          { int sys_mount(const char *type, const char *path, \
                        int flags, void *data); }
22     STD          { int sys_unmount(const char *path, int flags); }
23     STD          { int sys_setuid(uid_t uid); }
24     STD          { uid_t sys_getuid(void); }
25     STD          { uid_t sys_geteuid(void); }
#ifdef PTRACE
26     STD          { int sys_ptrace(int req, pid_t pid, caddr_t addr, \
                        int data); }
#else
26     UNIMPL       ptrace
#endif
27     STD          { ssize_t sys_recvmsg(int s, struct msghdr *msg, \
                        int flags); }
28     STD          { ssize_t sys_sendmsg(int s, \
                        const struct msghdr *msg, int flags); }
29     STD          { ssize_t sys_recvfrom(int s, void *buf, size_t len, \
                        int flags, struct sockaddr *from, \
                        socklen_t *fromlenaddr); }
30     STD          { int sys_accept(int s, struct sockaddr *name, \
                        socklen_t *anamelen); }
31     STD          { int sys_getpeername(int fdes, struct sockaddr *asa, \
                        int *alen); }
32     STD          { int sys_getsockname(int fdes, struct sockaddr *asa, \
                        socklen_t *alen); }
33     STD          { int sys_access(const char *path, int flags); }
34     STD          { int sys_chflags(const char *path, u_int flags); }
35     STD          { int sys_fchflags(int fd, u_int flags); }
36     STD          { void sys_sync(void); }
37     STD          { int sys_kill(int pid, int signum); }
38     COMPAT_43    { int sys_stat(const char *path, struct ostat *ub); } \
                        ostat
39     STD          { pid_t sys_getppid(void); }
40     COMPAT_43    { int sys_lstat(char *path, \
                        struct ostat *ub); } olstat
41     STD          { int sys_dup(int fd); }
42     STD          { int sys_opipe(void); }
43     STD          { gid_t sys_getegid(void); }
44     STD          { int sys_profil(caddr_t samples, size_t size, \
                        u_long offset, u_int scale); }
#ifdef KTRACE
45     STD          { int sys_ktrace(const char *fname, int ops, \
                        int facs, pid_t pid); }
#else
45     UNIMPL       ktrace
#endif
46     STD          { int sys_sigaction(int signum, \
```

```
                        const struct sigaction *nsa, \
                        struct sigaction *osa); }
47      STD             { gid_t sys_getgid(void); }
48      STD             { int sys_sigprocmask(int how, sigset_t mask); }
49      STD             { int sys_getlogin(char *namebuf, u_int namelen); }
50      STD             { int sys_setlogin(const char *namebuf); }
51      STD             { int sys_acct(const char *path); }
52      STD             { int sys_sigpending(void); }
53      STD             { int sys_osigaltstack(const struct osigaltstack *nss, \
                        struct osigaltstack *oss); }
54      STD             { int sys_ioctl(int fd, \
                        u_long com, ... void *data); }
55      STD             { int sys_reboot(int opt); }
56      STD             { int sys_revoke(const char *path); }
57      STD             { int sys_symlink(const char *path, \
                        const char *link); }
58      STD             { int sys_readlink(const char *path, char *buf, \
                        size_t count); }
59      STD             { int sys_execve(const char *path, \
                        char * const *argp, char * const *envp); }
60      STD             { int sys_umask(int newmask); }
61      STD             { int sys_chroot(const char *path); }
62      COMPAT_43       { int sys_fstat(int fd, struct ostat *sb); } ofstat
63      COMPAT_43       { int sys_getkerninfo(int op, char *where, int *size, \
                        int arg); } ogetkerninfo
64      COMPAT_43       { int sys_getpagesize(void); } ogetpagesize
65      COMPAT_25       { int sys_omsync(caddr_t addr, size_t len); }
66      STD             { int sys_vfork(void); }
67      OBSOL           vread
68      OBSOL           vwrite
69      STD             { int sys_sbrk(int incr); }
70      STD             { int sys_sstk(int incr); }
71      COMPAT_43       { int sys_mmap(caddr_t addr, size_t len, int prot, \
                        int flags, int fd, long pos); } ommap
72      STD             { int sys_ovadvise(int anom); } vadvise
73      STD             { int sys_munmap(void *addr, size_t len); }
74      STD             { int sys_mprotect(void *addr, size_t len, \
                        int prot); }
75      STD             { int sys_madvise(void *addr, size_t len, \
                        int behav); }
76      OBSOL           vhangup
77      OBSOL           vlimit
78      STD             { int sys_mincore(void *addr, size_t len, \
                        char *vec); }
79      STD             { int sys_getgroups(int gidsetsize, \
                        gid_t *gidset); }
80      STD             { int sys_setgroups(int gidsetsize, \
                        const gid_t *gidset); }
81      STD             { int sys_getpgrp(void); }
82      STD             { int sys_setpgid(pid_t pid, int pgid); }
83      STD             { int sys_setitimer(int which, \
                        const struct itimerval *itv, \
                        struct itimerval *oitv); }
84      COMPAT_43       { int sys_wait(void); } owait
85      COMPAT_25       { int sys_swapon(const char *name); }
86      STD             { int sys_getitimer(int which, \
                        struct itimerval *itv); }
87      COMPAT_43       { int sys_gethostname(char *hostname, u_int len); } \
                        ogethostname
88      COMPAT_43       { int sys_sethostname(char *hostname, u_int len); } \
                        osethostname
89      COMPAT_43       { int sys_getdtablesize(void); } ogetdtablesize
90      STD             { int sys_dup2(int from, int to); }
```

```
91      UNIMPL          getdopt
92      STD             { int sys_fcntl(int fd, int cmd, ... void *arg); }
93      STD             { int sys_select(int nd, fd_set *in, fd_set *ou, \
                            fd_set *ex, struct timeval *tv); }
94      UNIMPL          setdopt
95      STD             { int sys_fsync(int fd); }
96      STD             { int sys_setpriority(int which, id_t who, int prio); }
97      STD             { int sys_socket(int domain, int type, int protocol); }
98      STD             { int sys_connect(int s, const struct sockaddr *name, \
                            socklen_t namelen); }
99      COMPAT_43       { int sys_accept(int s, caddr_t name, \
                            int *anamelen); } oaccept
100     STD             { int sys_getpriority(int which, id_t who); }
101     COMPAT_43       { int sys_send(int s, caddr_t buf, int len, \
                            int flags); } osend
102     COMPAT_43       { int sys_recv(int s, caddr_t buf, int len, \
                            int flags); } orecv
103     STD             { int sys_sigreturn(struct sigcontext *sigcntxp); }
104     STD             { int sys_bind(int s, const struct sockaddr *name, \
                            socklen_t namelen); }
105     STD             { int sys_setsockopt(int s, int level, int name, \
                            const void *val, socklen_t valsize); }
106     STD             { int sys_listen(int s, int backlog); }
107     OBSOL           vtimes
108     COMPAT_43       { int sys_sigvec(int signum, struct sigvec *nsv, \
                            struct sigvec *osv); } osigvec
109     COMPAT_43       { int sys_sigblock(int mask); } osigblock
110     COMPAT_43       { int sys_sigsetmask(int mask); } osigsetmask
111     STD             { int sys_sigsuspend(int mask); }
112     COMPAT_43       { int sys_sigstack(struct sigstack *nss, \
                            struct sigstack *oss); } osigstack
113     COMPAT_43       { int sys_recvmsg(int s, struct omsghdr *msg, \
                            int flags); } orecvmsg
114     COMPAT_43       { int sys_sendmsg(int s, caddr_t msg, int flags); } \
                            osendmsg
115     OBSOL           vtrace
116     STD             { int sys_gettimeofday(struct timeval *tp, \
                            struct timezone *tzp); }
117     STD             { int sys_getrusage(int who, struct rusage *rusage); }
118     STD             { int sys_getsockopt(int s, int level, int name, \
                            void *val, socklen_t *avalsize); }
119     OBSOL           resuba
120     STD             { ssize_t sys_readv(int fd, \
                            const struct iovec *iovp, int iovcnt); }
121     STD             { ssize_t sys_writev(int fd, \
                            const struct iovec *iovp, int iovcnt); }
122     STD             { int sys_settimeofday(const struct timeval *tv, \
                            const struct timezone *tzp); }
123     STD             { int sys_fchown(int fd, uid_t uid, gid_t gid); }
124     STD             { int sys_fchmod(int fd, int mode); }
125     COMPAT_43       { int sys_recvfrom(int s, caddr_t buf, size_t len, \
                            int flags, caddr_t from, int *fromlenaddr); } \
                            orecvfrom
126     STD             { int sys_setreuid(uid_t ruid, uid_t euid); }
127     STD             { int sys_setregid(gid_t rgid, gid_t egid); }
128     STD             { int sys_rename(const char *from, const char *to); }
129     COMPAT_43       { int sys_truncate(const char *path, long length); } \
                            otruncate
130     COMPAT_43       { int sys_ftruncate(int fd, long length); } oftruncate
131     STD             { int sys_flock(int fd, int how); }
132     STD             { int sys_mkfifo(const char *path, int mode); }
133     STD             { ssize_t sys_sendto(int s, const void *buf, \
                            size_t len, int flags, const struct sockaddr *to, \
```

```
                        socklen_t tolen); }
134    STD             { int sys_shutdown(int s, int how); }
135    STD             { int sys_socketpair(int domain, int type, \
                        int protocol, int *rsv); }
136    STD             { int sys_mkdir(const char *path, int mode); }
137    STD             { int sys_rmdir(const char *path); }
138    STD             { int sys_utimes(const char *path, \
                        const struct timeval *tptr); }
139    OBSOL           4.2 sigreturn
140    STD             { int sys_adjtime(const struct timeval *delta, \
                        struct timeval *olddelta); }
141    COMPAT_43       { int sys_getpeername(int fdes, caddr_t asa, \
                        socklen_t *alen); } ogetpeername
142    COMPAT_43       { int32_t sys_gethostid(void); } ogethostid
143    COMPAT_43       { int sys_sethostid(int32_t hostid); } osethostid
144    COMPAT_43       { int sys_getrlimit(int which, \
                        struct ogetrlimit *rlp); } ogetrlimit
145    COMPAT_43       { int sys_setrlimit(int which, \
                        struct ogetrlimit *rlp); } osetrlimit
146    COMPAT_43       { int sys_killpg(int pgid, int signum); } okillpg
147    STD             { int sys_setsid(void); }
148    STD             { int sys_quotactl(const char *path, int cmd, \
                        int uid, char *arg); }
149    COMPAT_43       { int sys_quota(void); } oquota
150    COMPAT_43       { int sys_getsockname(int fdec, caddr_t asa, \
                        int *alen); } ogetsockname

; Syscalls 151-180 inclusive are reserved for vendor-specific
; system calls.  (This includes various calls added for compatibity
; with other Unix variants.)
; Some of these calls are now supported by BSD...
151    UNIMPL
152    UNIMPL
153    UNIMPL
154    UNIMPL
#if defined(NFSCLIENT) || defined(NFSSERVER)
155    STD             { int sys_nfssvc(int flag, void *argp); }
#else
155    UNIMPL
#endif
156    COMPAT_43       { int sys_getdirentries(int fd, char *buf, \
                        int count, long *basep); } ogetdirentries
157    COMPAT_25       { int sys_statfs(const char *path, \
                        struct ostatfs *buf); } ostatfs
158    COMPAT_25       { int sys_fstatfs(int fd, struct ostatfs *buf); } \
                        ostatfs
159    UNIMPL
160    UNIMPL
161    STD             { int sys_getfh(const char *fname, fhandle_t *fhp); }
162    COMPAT_09       { int sys_getdomainname(char *domainname, int len); } \
                        ogetdomainname
163    COMPAT_09       { int sys_setdomainname(char *domainname, int len); } \
                        osetdomainname
164    COMPAT_09       { int sys_uname(struct outsname *name); } ouname
165    STD             { int sys_sysarch(int op, void *parms); }
166    UNIMPL
167    UNIMPL
168    UNIMPL
#if defined(SYSVSEM) && !defined(__LP64__)
169    COMPAT_10       { int sys_semsys(int which, int a2, int a3, int a4, \
                        int a5); } osemsys
#else
169    UNIMPL          1.0 semsys
```

```
#endif
#if defined(SYSVMSG) && !defined(__LP64__)
170    COMPAT_10     { int sys_msgsys(int which, int a2, int a3, int a4, \
                        int a5, int a6); } omsgsys
#else
170    UNIMPL        1.0 msgsys
#endif
#if defined(SYSVSHM) && !defined(__LP64__)
171    COMPAT_10     { int sys_shmsys(int which, int a2, int a3, int a4); } \
                        oshmsys
#else
171    UNIMPL        1.0 shmsys
#endif
172    UNIMPL
173    STD           { ssize_t sys_pread(int fd, void *buf, \
                      size_t nbyte, int pad, off_t offset); }
174    STD           { ssize_t sys_pwrite(int fd, const void *buf, \
                      size_t nbyte, int pad, off_t offset); }
175    UNIMPL        ntp_gettime
176    UNIMPL        ntp_adjtime
177    UNIMPL
178    UNIMPL
179    UNIMPL
180    UNIMPL

; Syscalls 181-199 are used by/reserved for BSD
181    STD           { int sys_setgid(gid_t gid); }
182    STD           { int sys_setegid(gid_t egid); }
183    STD           { int sys_seteuid(uid_t euid); }
#ifdef LFS
184    STD           { int lfs_bmapv(fsid_t *fsidp, \
                        struct block_info *blkiov, int blkcnt); }
185    STD           { int lfs_markv(fsid_t *fsidp, \
                        struct block_info *blkiov, int blkcnt); }
186    STD           { int lfs_segclean(fsid_t *fsidp, u_long segment); }
187    STD           { int lfs_segwait(fsid_t *fsidp, struct timeval *tv); }
#else
184    UNIMPL
185    UNIMPL
186    UNIMPL
187    UNIMPL
#endif
188    STD           { int sys_stat(const char *path, struct stat *ub); }
189    STD           { int sys_fstat(int fd, struct stat *sb); }
190    STD           { int sys_lstat(const char *path, struct stat *ub); }
191    STD           { long sys_pathconf(const char *path, int name); }
192    STD           { long sys_fpathconf(int fd, int name); }
193    STD           { int sys_swapctl(int cmd, const void *arg, int misc); }
194    STD           { int sys_getrlimit(int which, \
                        struct rlimit *rlp); }
195    STD           { int sys_setrlimit(int which, \
                        const struct rlimit *rlp); }
196    STD           { int sys_getdirentries(int fd, char *buf, \
                        int count, long *basep); }
197    STD           { void *sys_mmap(void *addr, size_t len, int prot, \
                        int flags, int fd, long pad, off_t pos); }
198    INDIR         { quad_t sys___syscall(quad_t num, ...); }
199    STD           { off_t sys_lseek(int fd, int pad, off_t offset, \
                        int whence); }
200    STD           { int sys_truncate(const char *path, int pad, \
                        off_t length); }
201    STD           { int sys_ftruncate(int fd, int pad, off_t length); }
202    STD           { int sys___sysctl(int *name, u_int namelen, \
```

```
                        void *old, size_t *oldlenp, void *new, \
                        size_t newlen); }
203     STD             { int sys_mlock(const void *addr, size_t len); }
204     STD             { int sys_munlock(const void *addr, size_t len); }
205     STD             { int sys_undelete(const char *path); }
206     STD             { int sys_futimes(int fd, \
                        const struct timeval *tptr); }
207     STD             { pid_t sys_getpgid(pid_t pid); }
208     STD             { int sys_xfspioctl(int operation, char *a_pathP, \
                        int a_opcode, struct ViceIoctl *a_paramsP, \
                        int a_followSymlinks); }
209     UNIMPL
;
; Syscalls 210-219 are reserved for dynamically loaded syscalls
;
#ifdef LKM
210     NODEF           { int sys_lkmnosys(void); }
211     NODEF           { int sys_lkmnosys(void); }
212     NODEF           { int sys_lkmnosys(void); }
213     NODEF           { int sys_lkmnosys(void); }
214     NODEF           { int sys_lkmnosys(void); }
215     NODEF           { int sys_lkmnosys(void); }
216     NODEF           { int sys_lkmnosys(void); }
217     NODEF           { int sys_lkmnosys(void); }
218     NODEF           { int sys_lkmnosys(void); }
219     NODEF           { int sys_lkmnosys(void); }
#else  /* !LKM */
210     UNIMPL
211     UNIMPL
212     UNIMPL
213     UNIMPL
214     UNIMPL
215     UNIMPL
216     UNIMPL
217     UNIMPL
218     UNIMPL
219     UNIMPL
#endif /* !LKM */
; System calls 220-240 are reserved for use by OpenBSD
#ifdef SYSVSEM
220     COMPAT_23       { int sys___semctl(int semid, int semnum, int cmd, \
                        union semun *arg); } __osemctl
221     STD             { int sys_semget(key_t key, int nsems, int semflg); }
222     STD             { int sys_semop(int semid, struct sembuf *sops, \
                        u_int nsops); }
223     OBSOL           sys_semconfig
#else
220     UNIMPL          semctl
221     UNIMPL          semget
222     UNIMPL          semop
223     UNIMPL          semconfig
#endif
#ifdef SYSVMSG
224     COMPAT_23       { int sys_msgctl(int msqid, int cmd, \
                        struct omsqid_ds *buf); } omsgctl
225     STD             { int sys_msgget(key_t key, int msgflg); }
226     STD             { int sys_msgsnd(int msqid, const void *msgp, size_t msgsz, \
                        int msgflg); }
227     STD             { int sys_msgrcv(int msqid, void *msgp, size_t msgsz, \
                        long msgtyp, int msgflg); }
#else
224     UNIMPL          msgctl
225     UNIMPL          msgget
```

```
226    UNIMPL        msgsnd
227    UNIMPL        msgrcv
#endif
#ifdef SYSVSHM
228    STD           { void *sys_shmat(int shmid, const void *shmaddr, \
                         int shmflg); }
229    COMPAT_23     { int sys_shmctl(int shmid, int cmd, \
                         struct oshmid_ds *buf); } oshmctl
230    STD           { int sys_shmdt(const void *shmaddr); }
231    STD           { int sys_shmget(key_t key, int size, int shmflg); }
#else
228    UNIMPL        shmat
229    UNIMPL        shmctl
230    UNIMPL        shmdt
231    UNIMPL        shmget
#endif
232    STD           { int sys_clock_gettime(clockid_t clock_id, \
                         struct timespec *tp); }
233    STD           { int sys_clock_settime(clockid_t clock_id, \
                         const struct timespec *tp); }
234    STD           { int sys_clock_getres(clockid_t clock_id, \
                         struct timespec *tp); }
235    UNIMPL        timer_create
236    UNIMPL        timer_delete
237    UNIMPL        timer_settime
238    UNIMPL        timer_gettime
239    UNIMPL        timer_getoverrun
;
; System calls 240-249 are reserved for other IEEE Std1003.1b syscalls
;
240    STD           { int sys_nanosleep(const struct timespec *rqtp, \
                         struct timespec *rmtp); }
241    UNIMPL
242    UNIMPL
243    UNIMPL
244    UNIMPL
245    UNIMPL
246    UNIMPL
247    UNIMPL
248    UNIMPL
249    UNIMPL
250    STD           { int sys_minherit(void *addr, size_t len, \
                         int inherit); }
251    STD           { int sys_rfork(int flags); }
252    STD           { int sys_poll(struct pollfd *fds, \
                         u_int nfds, int timeout); }
253    STD           { int sys_issetugid(void); }
254    STD           { int sys_lchown(const char *path, uid_t uid, gid_t gid); }
255    STD           { pid_t sys_getsid(pid_t pid); }
256    STD           { int sys_msync(void *addr, size_t len, int flags); }
#ifdef SYSVSEM
257    STD           { int sys___semctl(int semid, int semnum, int cmd, \
                         union semun *arg); }
#else
257    UNIMPL
#endif
#ifdef SYSVSHM
258    STD           { int sys_shmctl(int shmid, int cmd, \
                         struct shmid_ds *buf); }
#else
258    UNIMPL
#endif
#ifdef SYSVMSG
```

```
259    STD          { int sys_msgctl(int msqid, int cmd, \
                        struct msqid_ds *buf); }
#else
259    UNIMPL
#endif
260    STD          { int sys_getfsstat(struct statfs *buf, size_t bufsize, \
                        int flags); }
261    STD          { int sys_statfs(const char *path, \
                        struct statfs *buf); }
262    STD          { int sys_fstatfs(int fd, struct statfs *buf); }
263    STD          { int sys_pipe(int *fdp); }
264    STD          { int sys_fhopen(const fhandle_t *fhp, int flags); }
265    STD          { int sys_fhstat(const fhandle_t *fhp, \
                        struct stat *sb); }
266    STD          { int sys_fhstatfs(const fhandle_t *fhp, \
                        struct statfs *buf); }
267    STD          { ssize_t sys_preadv(int fd, \
                      const struct iovec *iovp, int iovcnt, \
                      int pad, off_t offset); }
268    STD          { ssize_t sys_pwritev(int fd, \
                      const struct iovec *iovp, int iovcnt, \
                      int pad, off_t offset); }
269    STD          { int sys_kqueue(void); }
270    STD          { int sys_kevent(int fd, \
                        const struct kevent *changelist, int nchanges, \
                        struct kevent *eventlist, int nevents, \
                        const struct timespec *timeout); }
271    STD          { int sys_mlockall(int flags); }
272    STD          { int sys_munlockall(void); }
273    STD          { int sys_getpeereid(int fdes, uid_t *euid, gid_t *egid); }
#ifdef UFS_EXTATTR
274    STD          { int sys_extattrctl(const char *path, int cmd, \
                        const char *filename, int attrnamespace, \
                        const char *attrname); }
275    STD          { int sys_extattr_set_file(const char *path, \
                        int attrnamespace, const char *attrname, \
                        void *data, size_t nbytes); }
276    STD          { ssize_t sys_extattr_get_file(const char *path, \
                        int attrnamespace, const char *attrname, \
                        void *data, size_t nbytes); }
277    STD          { int sys_extattr_delete_file(const char *path, \
                        int attrnamespace, const char *attrname); }
278    STD          { int sys_extattr_set_fd(int fd, int attrnamespace, \
                        const char *attrname, void *data, \
                        size_t nbytes); }
279    STD          { ssize_t sys_extattr_get_fd(int fd, \
                        int attrnamespace, const char *attrname, \
                        void *data, size_t nbytes); }
280    STD          { int sys_extattr_delete_fd(int fd, int attrnamespace, \
                        const char *attrname); }
#else
274    UNIMPL       sys_extattrctl
275    UNIMPL       sys_extattr_set_file
276    UNIMPL       sys_extattr_get_file
277    UNIMPL       sys_extattr_delete_file
278    UNIMPL       sys_extattr_set_fd
279    UNIMPL       sys_extattr_get_fd
280    UNIMPL       sys_extattr_delete_fd
#endif
281    STD          { int sys_getresuid(uid_t *ruid, uid_t *euid, \
                        uid_t *suid); }
282    STD          { int sys_setresuid(uid_t ruid, uid_t euid, \
                        uid_t suid); }
```

```
283    STD           { int sys_getresgid(gid_t *rgid, gid_t *egid, \
                         gid_t *sgid); }
284    STD           { int sys_setresgid(gid_t rgid, gid_t egid, \
                         gid_t sgid); }
285    OBSOL         sys_omquery
286    STD           { void *sys_mquery(void *addr, size_t len, int prot, \
                         int flags, int fd, long pad, off_t pos); }
287    STD           { int sys_closefrom(int fd); }
288    STD           { int sys_sigaltstack(const struct sigaltstack *nss, \
                         struct sigaltstack *oss); }

;=============================================================================
; COP4600 syscalls
;=============================================================================

;added by Dave Small
289    STD           { int sys_hello( void ); }
290    STD           { int sys_showargs( const char *str, int val ); }
291    STD    { int sys_cipher (char *text, int lkey, int nkey); }
292    STD           { int sys_allocate_semaphore (const char *name, int
initial_count); }
293    STD           { int sys_down_semaphore (const char *name); }
294    STD           { int sys_up_semaphore (const char *name); }
295    STD           { int sys_free_semaphore (const char *name); }
```

Subsection V: usr/src/sys/sys/proc.h

```
/*     $OpenBSD: proc.h,v 1.68 2003/11/08 06:11:11 nordin Exp $    */
/*     $NetBSD: proc.h,v 1.44 1996/04/22 01:23:21 christos Exp $   */

/*-
 * Copyright (c) 1986, 1989, 1991, 1993
 *     The Regents of the University of California.  All rights reserved.
 * (c) UNIX System Laboratories, Inc.
 * All or some portions of this file are derived from material licensed
 * to the University of California by American Telephone and Telegraph
 * Co. or Unix System Laboratories, Inc. and are reproduced herein with
 * the permission of UNIX System Laboratories, Inc.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *     @(#)proc.h    8.8 (Berkeley) 1/21/94
 */

#ifndef _SYS_PROC_H_
#define       _SYS_PROC_H_

#include <machine/proc.h>        /* Machine-dependent proc substruct. */
#include <sys/select.h>                 /* For struct selinfo. */
#include <sys/queue.h>
#include <sys/timeout.h>         /* For struct timeout. */
#include <sys/event.h>                  /* For struct klist */
#include <sys/lock.h>                   /* Dawit modified */
/*
 * One structure allocated per session.
 */
struct session {
       int    s_count;            /* Ref cnt; pgrps in session. */
       struct proc *s_leader;             /* Session leader. */
       struct vnode *s_ttyvp;             /* Vnode of controlling terminal. */
       struct tty *s_ttyp;        /* Controlling terminal. */
       char   s_login[MAXLOGNAME];       /* Setlogin() name. */
};

/*
 * One structure allocated per process group.
 */
```

```
struct pgrp {
	LIST_ENTRY(pgrp) pg_hash;	/* Hash chain. */
	LIST_HEAD(, proc) pg_members;	/* Pointer to pgrp members. */
	struct session *pg_session;	/* Pointer to session. */
	pid_t	pg_id;			/* Pgrp id. */
	int	pg_jobc;	/* # procs qualifying pgrp for job control */
};

/*
 * One structure allocated per emulation.
 */
struct exec_package;
struct ps_strings;
struct uvm_object;
union sigval;

struct emul {
	char	e_name[8];		/* Symbolic name */
	int	*e_errno;		/* Errno array */
					/* Signal sending function */
	void	(*e_sendsig)(sig_t, int, int, u_long, int, union sigval);
	int	e_nosys;		/* Offset of the nosys() syscall */
	int	e_nsysent;		/* Number of system call entries */
	struct sysent *e_sysent;	/* System call array */
	char	**e_syscallnames;	/* System call name array */
	int	e_arglen;		/* Extra argument size in words */
					/* Copy arguments on the stack */
	void	*(*e_copyargs)(struct exec_package *, struct ps_strings *,
				    void *, void *);
					/* Set registers before execution */
	void	(*e_setregs)(struct proc *, struct exec_package *,
				    u_long, register_t *);
	int	(*e_fixup)(struct proc *, struct exec_package *);
	char	*e_sigcode;		/* Start of sigcode */
	char	*e_esigcode;		/* End of sigcode */
	int	e_flags;		/* Flags, see below */
	struct uvm_object *e_sigobject;	/* shared sigcode object */
					/* Per-process hooks */
	void	(*e_proc_exec)(struct proc *, struct exec_package *);
	void	(*e_proc_fork)(struct proc *p, struct proc *parent);
	void	(*e_proc_exit)(struct proc *);
};
/* Flags for e_flags */
#define	EMUL_ENABLED	0x0001		/* Allow exec to continue */
#define	EMUL_NATIVE	0x0002		/* Always enabled */

extern struct emul *emulsw[];		/* All emuls in system */
extern int nemuls;		/* Number of emuls */

/*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes.  The process structure and the substructures
 * are always addressable except for those marked "(PROC ONLY)" below,
 * which might be addressable only on a processor on which the process
 * is running.
 */
struct proc {
	struct proc *p_forw;		/* Doubly-linked run/sleep queue. */
	struct proc *p_back;
```

```
        LIST_ENTRY(proc) p_list;   /* List of all processes. */

        /* substructures: */
        struct pcred *p_cred;            /* Process owner's identity. */
        struct filedesc *p_fd;           /* Ptr to open files structure. */
        struct pstats *p_stats;    /* Accounting/statistics (PROC ONLY). */
        struct plimit *p_limit;    /* Process limits. */
        struct vmspace *p_vmspace; /* Address space. */
        struct sigacts *p_sigacts; /* Signal actions, state (PROC ONLY). */

#define       p_ucred             p_cred->pc_ucred
#define       p_rlimit     p_limit->pl_rlimit

        int    p_exitsig;          /* Signal to send to parent on exit. */
        int    p_flag;                  /* P_* flags. */
        u_char p_os;               /* OS tag */
        char   p_stat;                  /* S* process status. */
        char   p_pad1[2];

        pid_t  p_pid;              /* Process identifier. */
        LIST_ENTRY(proc) p_hash;   /* Hash chain. */
        LIST_ENTRY(proc) p_pglist; /* List of processes in pgrp. */
        struct proc *p_pptr;       /* Pointer to parent process. */
        LIST_ENTRY(proc) p_sibling;      /* List of sibling processes. */
        LIST_HEAD(, proc) p_children;    /* Pointer to list of children. */

/* The following fields are all zeroed upon creation in fork. */
#define       p_startzero  p_oppid

        pid_t  p_oppid;      /* Save parent pid during ptrace. XXX */
        int    p_dupfd;      /* Sideways return value from filedescopen. XXX */

        /* scheduling */
        u_int  p_estcpu;     /* Time averaged value of p_cpticks. */
        int    p_cpticks;    /* Ticks of cpu time. */
        fixpt_t      p_pctcpu;     /* %cpu for this process during p_swtime */
        void   *p_wchan;     /* Sleep address. */
        struct timeout p_sleep_to;/* timeout for tsleep() */
        const char *p_wmesg;        /* Reason for sleep. */
        u_int  p_swtime;     /* Time swapped in or out. */
        u_int  p_slptime;    /* Time since last blocked. */
        int    p_schedflags; /* PSCHED_* flags */

        struct itimerval p_realtimer;     /* Alarm timer. */
        struct timeout p_realit_to;       /* Alarm timeout. */
        struct timeval p_rtime;    /* Real time. */
        u_quad_t p_uticks;         /* Statclock hits in user mode. */
        u_quad_t p_sticks;         /* Statclock hits in system mode. */
        u_quad_t p_iticks;         /* Statclock hits processing intr. */

        int    p_traceflag;        /* Kernel trace points. */
        struct vnode *p_tracep;    /* Trace to vnode. */

        void   *p_systrace;        /* Back pointer to systrace */

        int    p_siglist;          /* Signals arrived but not delivered. */

        struct vnode *p_textvp;    /* Vnode of executable. */

        int    p_holdcnt;          /* If non-zero, don't swap. */
        struct emul *p_emul;       /* Emulation information */
        void   *p_emuldata;        /* Per-process emulation data, or */
                                   /* NULL. Malloc type M_EMULDATA */
```

```
        struct klist p_klist;             /* knotes attached to this process */
                                  /* pad to 256, avoid shifting eproc. */


/* End area that is zeroed on creation. */
#define       p_endzero     p_startcopy

/* The following fields are all copied upon creation in fork. */
#define       p_startcopy   p_sigmask

        sigset_t p_sigmask; /* Current signal mask. */
        sigset_t p_sigignore;       /* Signals being ignored. */
        sigset_t p_sigcatch;        /* Signals being caught by user. */

        u_char p_priority;   /* Process priority. */
        u_char p_usrpri;     /* User-priority based on p_cpu and p_nice. */
        char   p_nice;              /* Process "nice" value. */
        char   p_comm[MAXCOMLEN+1];

        struct        pgrp *p_pgrp; /* Pointer to process group. */
        vaddr_t       p_sigcode;    /* user pointer to the signal code. */

/* End area that is copied on creation. */
#define       p_endcopy     p_addr

        struct user *p_addr; /* Kernel virtual addr of u-area (PROC ONLY). */
        struct mdproc p_md;  /* Any machine-dependent fields. */

        u_short       p_xstat;      /* Exit status for wait; also stop signal. */
        u_short       p_acflag;     /* Accounting flags. */
        struct rusage *p_ru; /* Exit information. XXX */

        /***** BEGIN ADDITION by Dawit Woldegiorgis
************************************/

        int inherited;              /* Flag to check if process should semaphores from
parent */
        LIST_HEAD(s_list, semaphore) semaphores;        /* Semaphores the process owns
*/
        /* Check end of file for semaphore */

        /***** END ADDITION by Dawit Woldegiorgis
************************************/
};

#define       p_session     p_pgrp->pg_session
#define       p_pgid        p_pgrp->pg_id

/* Status values. */
#define       SIDL   1              /* Process being created by fork. */
#define       SRUN   2              /* Currently runnable. */
#define       SSLEEP 3              /* Sleeping on an address. */
#define       SSTOP  4              /* Process debugging or suspension. */
#define       SZOMB  5              /* Awaiting collection by parent. */
#define SDEAD 6             /* Process is almost a zombie. */

#define P_ZOMBIE(p) ((p)->p_stat == SZOMB || (p)->p_stat == SDEAD)

/* These flags are kept in p_flag. */
#define       P_ADVLOCK     0x000001      /* Proc may hold a POSIX adv. lock. */
#define       P_CONTROLT    0x000002      /* Has a controlling terminal. */
#define       P_INMEM             0x000004      /* Loaded into memory. */
#define       P_NOCLDSTOP   0x000008      /* No SIGCHLD when children stop. */
```

```
#define       P_PPWAIT      0x000010      /* Parent waits for child exec/exit. */
#define       P_PROFIL      0x000020      /* Has started profiling. */
#define       P_SELECT      0x000040      /* Selecting; wakeup/waiting danger. */
#define       P_SINTR             0x000080      /* Sleep is interruptible. */
#define       P_SUGID             0x000100      /* Had set id privs since last exec.
*/
#define       P_SYSTEM      0x000200      /* No sigs, stats or swapping. */
#define       P_TIMEOUT     0x000400      /* Timing out during sleep. */
#define       P_TRACED      0x000800      /* Debugged process being traced. */
#define       P_WAITED      0x001000      /* Debugging proc has waited for child. */
/* XXX - Should be merged with INEXEC */
#define       P_WEXIT             0x002000      /* Working on exiting. */
#define       P_EXEC        0x004000      /* Process called exec. */

/* Should be moved to machine-dependent areas. */
#define       P_OWEUPC      0x008000      /* Owe proc an addupc() at next ast. */

/* XXX Not sure what to do with these, yet. */
#define       P_FSTRACE     0x010000      /* tracing via fs (elsewhere?) */
#define       P_SSTEP             0x020000      /* proc needs single-step fixup ???
*/
#define       P_SUGIDEXEC   0x040000      /* last execve() was set[ug]id */

#define       P_NOCLDWAIT   0x080000      /* Let pid 1 wait for my children */
#define       P_NOZOMBIE    0x100000      /* Pid 1 waits for me instead of dad */
#define P_INEXEC    0x200000      /* Process is doing an exec right now */
#define P_SYSTRACE  0x400000      /* Process system call tracing active*/
#define P_CONTINUED 0x800000      /* Proc has continued from a stopped state. */
#define P_SWAPIN    0x1000000     /* Swapping in right now */

#define       P_BITS \
    ("\20\01ADVLOCK\02CTTY\03INMEM\04NOCLDSTOP\05PPWAIT\06PROFIL\07SELECT" \
     "\010SINTR\011SUGID\012SYSTEM\013TIMEOUT\014TRACED\015WAITED\016WEXIT" \
     "\017EXEC\020PWEUPC\021FSTRACE\022SSTEP\023SUGIDEXEC\024NOCLDWAIT" \
     "\025NOZOMBIE\026INEXEC\027SYSTRACE\030CONTINUED")

/* Macro to compute the exit signal to be delivered. */
#define P_EXITSIG(p) \
    (((p)->p_flag & (P_TRACED | P_FSTRACE)) ? SIGCHLD : (p)->p_exitsig)

/*
 * These flags are kept in p_schedflags.  p_schedflags may be modified
 * only at splstatclock().
 */
#define PSCHED_SEENRR             0x0001 /* process has been in roundrobin() */
#define PSCHED_SHOULDYIELD 0x0002 /* process should yield */

#define PSCHED_SWITCHCLEAR (PSCHED_SEENRR|PSCHED_SHOULDYIELD)

/*
 * MOVE TO ucred.h?
 *
 * Shareable process credentials (always resident).  This includes a reference
 * to the current user credentials as well as real and saved ids that may be
 * used to change ids.
 */
struct pcred {
       struct ucred *pc_ucred;    /* Current credentials. */
       uid_t  p_ruid;                   /* Real user id. */
       uid_t  p_svuid;            /* Saved effective user id. */
       gid_t  p_rgid;                   /* Real group id. */
       gid_t  p_svgid;            /* Saved effective group id. */
       int    p_refcnt;           /* Number of references. */
```

```
};

#ifdef _KERNEL
/*
 * We use process IDs <= PID_MAX; PID_MAX + 1 must also fit in a pid_t,
 * as it is used to represent "no process group".
 * We set PID_MAX to (SHRT_MAX - 1) so we don't break sys/compat.
 */
#define	PID_MAX			32766
#define	NO_PID		(PID_MAX+1)

#define SESS_LEADER(p)	((p)->p_session->s_leader == (p))
#define	SESSHOLD(s)	((s)->s_count++)
#define	SESSRELE(s) {                                           \
	if (--(s)->s_count == 0)                                \
		pool_put(&session_pool, s);                     \
}

#define	PHOLD(p) {                                              \
	if ((p)->p_holdcnt++ == 0 && ((p)->p_flag & P_INMEM) == 0)  \
		uvm_swapin(p);                                  \
}
#define	PRELE(p)	(--(p)->p_holdcnt)

/*
 * Flags to fork1().
 */
#define FORK_FORK   0x00000001
#define FORK_VFORK  0x00000002
#define FORK_RFORK  0x00000004
#define FORK_PPWAIT 0x00000008
#define FORK_SHAREFILES    0x00000010
#define FORK_CLEANFILES    0x00000020
#define FORK_NOZOMBIE      0x00000040
#define FORK_SHAREVM       0x00000080
#define FORK_VMNOSTACK     0x00000100
#define FORK_SIGHAND       0x00000200

#define	PIDHASH(pid)	(&pidhashtbl[(pid) & pidhash])
extern LIST_HEAD(pidhashhead, proc) *pidhashtbl;
extern u_long pidhash;

#define	PGRPHASH(pgid)	(&pgrphashtbl[(pgid) & pgrphash])
extern LIST_HEAD(pgrphashhead, pgrp) *pgrphashtbl;
extern u_long pgrphash;

#ifndef curproc
extern struct proc *curproc;		/* Current running proc. */
#endif
extern struct proc proc0;		/* Process slot for swapper. */
extern int nprocs, maxproc;		/* Current and max number of procs. */
extern int randompid;			/* fork() should create random pid's */

LIST_HEAD(proclist, proc);
extern struct proclist allproc;		/* List of all processes. */
extern struct proclist zombproc;	/* List of zombie processes. */

extern struct proclist deadproc;	/* List of dead processes. */
extern struct simplelock deadproc_slock;

extern struct proc *initproc;		/* Process slots for init, pager. */
extern struct proc *syncerproc;		/* filesystem syncer daemon */
```

```
extern struct pool proc_pool;           /* memory pool for procs */
extern struct pool rusage_pool;         /* memory pool for zombies */
extern struct pool ucred_pool;          /* memory pool for ucreds */
extern struct pool session_pool; /* memory pool for sessions */
extern struct pool pcred_pool;          /* memory pool for pcreds */

#define      NQS    32                  /* 32 run queues. */
extern int whichqs;              /* Bit mask summary of non-empty Q's. */
struct prochd {
       struct proc *ph_link;            /* Linked list of running processes. */
       struct proc *ph_rlink;
};
extern struct prochd qs[NQS];

struct simplelock;

struct proc *pfind(pid_t); /* Find process by id. */
struct pgrp *pgfind(pid_t);      /* Find process group by id. */
void   proc_printit(struct proc *p, const char *modif,
    int (*pr)(const char *, ...));

int    chgproccnt(uid_t uid, int diff);
int    enterpgrp(struct proc *p, pid_t pgid, int mksess);
void   fixjobc(struct proc *p, struct pgrp *pgrp, int entering);
int    inferior(struct proc *p);
int    leavepgrp(struct proc *p);
void   yield(void);
void   preempt(struct proc *);
void   mi_switch(void);
void   pgdelete(struct pgrp *pgrp);
void   procinit(void);
#if !defined(remrunqueue)
void   remrunqueue(struct proc *);
#endif
void   resetpriority(struct proc *);
void   setrunnable(struct proc *);
#if !defined(setrunqueue)
void   setrunqueue(struct proc *);
#endif
void   sleep(void *chan, int pri);
void   uvm_swapin(struct proc *);  /* XXX: uvm_extern.h? */
int    ltsleep(void *chan, int pri, const char *wmesg, int timo,
           volatile struct simplelock *);
#define tsleep(chan, pri, wmesg, timo) ltsleep(chan, pri, wmesg, timo, NULL)
void   unsleep(struct proc *);
void    wakeup_n(void *chan, int);
void    wakeup(void *chan);
#define wakeup_one(c) wakeup_n((c), 1)
void   reaper(void);
void   exit1(struct proc *, int);
void   exit2(struct proc *);
int    fork1(struct proc *, int, int, void *, size_t, void (*)(void *),
           void *, register_t *);
void   rqinit(void);
int    groupmember(gid_t, struct ucred *);
#if !defined(cpu_switch)
void   cpu_switch(struct proc *);
#endif
#if !defined(cpu_wait)
void   cpu_wait(struct proc *);
#endif
void   cpu_exit(struct proc *);
```

```
void   child_return(void *);

int    proc_cansugid(struct proc *);
void   proc_zap(struct proc *);
#endif /* _KERNEL */
#endif /* !_SYS_PROC_H_ */

#define MAX_NAME_LENGTH 32             /* max sempahore name length, including null
terminator */

/***** BEGIN ADDITION by Dawit Woldegiorgis ***********************************/

#ifndef SEMAPHORE_P
#define SEMAPHORE_P

/* Semahore struct; Dawit modified */
typedef struct semaphore {
       struct proc *owner;                 /* process that created the semaphore */
    char name[MAX_NAME_LENGTH];        /* string name of semaphore */
    int count;                         /* control variable of semaphore */
    lock_data_t mutex;                 /* lock structure */
    SIMPLEQ_HEAD(,p_node) p_head;           /* list of processes waiting on semaphore
*/
    LIST_ENTRY(semaphore) s_next;      /* node in system wide list of semaphores */
} semaphore_t;

/*
 * Node of SIMPLEQ (defined above) used to keep track of waiting processes in
semaphore
 */
struct p_node {
  struct proc *p;                      /* pointer to process */
  SIMPLEQ_ENTRY(p_node) p_next;        /* link to next entry */
};

#endif

/***** END ADDITION by Dawit Woldegiorgis *************************************/
```

Subsection V: usr/src/sys/kern/kern_fork.c

```
/*     $OpenBSD: kern_fork.c,v 1.63 2003/09/23 20:26:18 millert Exp $     */
/*     $NetBSD: kern_fork.c,v 1.29 1996/02/09 18:59:34 christos Exp $     */

/*
 * Copyright (c) 1982, 1986, 1989, 1991, 1993
 *     The Regents of the University of California.  All rights reserved.
 * (c) UNIX System Laboratories, Inc.
 * All or some portions of this file are derived from material licensed
 * to the University of California by American Telephone and Telegraph
 * Co. or Unix System Laboratories, Inc. and are reproduced herein with
 * the permission of UNIX System Laboratories, Inc.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *     @(#)kern_fork.c     8.6 (Berkeley) 4/8/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/filedesc.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/mount.h>
#include <sys/proc.h>
#include <sys/resourcevar.h>
#include <sys/signalvar.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/acct.h>
#include <sys/ktrace.h>
#include <sys/sched.h>
#include <dev/rndvar.h>
#include <sys/pool.h>
#include <sys/mman.h>

#include <sys/syscallargs.h>

#include "systrace.h"
#include <dev/systrace.h>

#include <uvm/uvm_extern.h>
```

```
#include <uvm/uvm_map.h>

int    nprocs = 1;         /* process 0 */
int    randompid;          /* when set to 1, pid's go random */
pid_t  lastpid;
struct forkstat forkstat;

int pidtaken(pid_t);

/*ARGSUSED*/
int
sys_fork(struct proc *p, void *v, register_t *retval)
{
       return (fork1(p, SIGCHLD, FORK_FORK, NULL, 0, NULL, NULL, retval));
}

/*ARGSUSED*/
int
sys_vfork(struct proc *p, void *v, register_t *retval)
{
       return (fork1(p, SIGCHLD, FORK_VFORK|FORK_PPWAIT, NULL, 0, NULL,
           NULL, retval));
}

int
sys_rfork(struct proc *p, void *v, register_t *retval)
{
       struct sys_rfork_args /* {
              syscallarg(int) flags;
       } */ *uap = v;

       int rforkflags;
       int flags;

       flags = FORK_RFORK;
       rforkflags = SCARG(uap, flags);

       if ((rforkflags & RFPROC) == 0)
              return (EINVAL);

       switch(rforkflags & (RFFDG|RFCFDG)) {
       case (RFFDG|RFCFDG):
              return EINVAL;
       case RFCFDG:
              flags |= FORK_CLEANFILES;
              break;
       case RFFDG:
              break;
       default:
              flags |= FORK_SHAREFILES;
              break;
       }

       if (rforkflags & RFNOWAIT)
              flags |= FORK_NOZOMBIE;

       if (rforkflags & RFMEM)
              flags |= FORK_VMNOSTACK;

       return (fork1(p, SIGCHLD, flags, NULL, 0, NULL, NULL, retval));
}

/* print the 'table full' message once per 10 seconds */
```

```
struct timeval fork_tfmrate = { 10, 0 };

int
fork1(struct proc *p1, int exitsig, int flags, void *stack, size_t stacksize,
    void (*func)(void *), void *arg, register_t *retval)
{
	struct proc *p2;
	uid_t uid;
	struct vmspace *vm;
	int count;
	vaddr_t uaddr;
	int s;
	extern void endtsleep(void *);
	extern void realitexpire(void *);

	/*
	 * Although process entries are dynamically created, we still keep
	 * a global limit on the maximum number we will create. We reserve
	 * the last 5 processes to root. The variable nprocs is the current
	 * number of processes, maxproc is the limit.
	 */
	uid = p1->p_cred->p_ruid;
	if ((nprocs >= maxproc - 5 && uid != 0) || nprocs >= maxproc) {
		static struct timeval lasttfm;

		if (ratecheck(&lasttfm, &fork_tfmrate))
			tablefull("proc");
		return (EAGAIN);
	}
	nprocs++;

	/*
	 * Increment the count of procs running with this uid. Don't allow
	 * a nonprivileged user to exceed their current limit.
	 */
	count = chgproccnt(uid, 1);
	if (uid != 0 && count > p1->p_rlimit[RLIMIT_NPROC].rlim_cur) {
		(void)chgproccnt(uid, -1);
		nprocs--;
		return (EAGAIN);
	}

	/*
	 * Allocate a pcb and kernel stack for the process
	 */
	uaddr = uvm_km_valloc(kernel_map, USPACE);
	if (uaddr == 0) {
		chgproccnt(uid, -1);
		nprocs--;
		return (ENOMEM);
	}

	/*
	 * From now on, we're committed to the fork and cannot fail.
	 */

	/* Allocate new proc. */
	p2 = pool_get(&proc_pool, PR_WAITOK);

	p2->p_stat = SIDL;			/* protect against others */
	p2->p_exitsig = exitsig;
	p2->p_forw = p2->p_back = NULL;
```

```
	/*
	 * Make a proc table entry for the new process.
	 * Start by zeroing the section of proc that is zero-initialized,
	 * then copy the section that is copied directly from the parent.
	 */
	bzero(&p2->p_startzero,
	    (unsigned) ((caddr_t)&p2->p_endzero - (caddr_t)&p2->p_startzero));
	bcopy(&p1->p_startcopy, &p2->p_startcopy,
	    (unsigned) ((caddr_t)&p2->p_endcopy - (caddr_t)&p2->p_startcopy));

	/*
	 * Initialize the timeouts.
	 */
	timeout_set(&p2->p_sleep_to, endtsleep, p2);
	timeout_set(&p2->p_realit_to, realitexpire, p2);

	/*
	 * Duplicate sub-structures as needed.
	 * Increase reference counts on shared objects.
	 * The p_stats and p_sigacts substructs are set in vm_fork.
	 */
	p2->p_flag = P_INMEM;
	p2->p_emul = p1->p_emul;
	if (p1->p_flag & P_PROFIL)
		startprofclock(p2);
	p2->p_flag |= (p1->p_flag & (P_SUGID | P_SUGIDEXEC));
	p2->p_cred = pool_get(&pcred_pool, PR_WAITOK);
	bcopy(p1->p_cred, p2->p_cred, sizeof(*p2->p_cred));
	p2->p_cred->p_refcnt = 1;
	crhold(p1->p_ucred);

	/* bump references to the text vnode (for procfs) */
	p2->p_textvp = p1->p_textvp;
	if (p2->p_textvp)
		VREF(p2->p_textvp);

	if (flags & FORK_CLEANFILES)
		p2->p_fd = fdinit(p1);
	else if (flags & FORK_SHAREFILES)
		p2->p_fd = fdshare(p1);
	else
		p2->p_fd = fdcopy(p1);

	/*
	 * If p_limit is still copy-on-write, bump refcnt,
	 * otherwise get a copy that won't be modified.
	 * (If PL_SHAREMOD is clear, the structure is shared
	 * copy-on-write.)
	 */
	if (p1->p_limit->p_lflags & PL_SHAREMOD)
		p2->p_limit = limcopy(p1->p_limit);
	else {
		p2->p_limit = p1->p_limit;
		p2->p_limit->p_refcnt++;
	}

	if (p1->p_session->s_ttyvp != NULL && p1->p_flag & P_CONTROLT)
		p2->p_flag |= P_CONTROLT;
	if (flags & FORK_PPWAIT)
		p2->p_flag |= P_PPWAIT;
	LIST_INSERT_AFTER(p1, p2, p_pglist);
	p2->p_pptr = p1;
	if (flags & FORK_NOZOMBIE)
```

```
                p2->p_flag |= P_NOZOMBIE;
        LIST_INSERT_HEAD(&p1->p_children, p2, p_sibling);
        LIST_INIT(&p2->p_children);

        /***** BEGIN ADDITION by Dawit Woldegiorgis
**********************************/

        LIST_INIT(&p2->semaphores);
        if (LIST_EMPTY(&p1->semaphores))  /* nothing to inherit */
                p2->inherited = 0;
        else                                            /* child should inherit
parent's semaphores */
                p2->inherited = 1;

        /***** END ADDITION by Dawit Woldegiorgis
**********************************/

#ifdef KTRACE
        /*
         * Copy traceflag and tracefile if enabled.
         * If not inherited, these were zeroed above.
         */
        if (p1->p_traceflag & KTRFAC_INHERIT) {
                p2->p_traceflag = p1->p_traceflag;
                if ((p2->p_tracep = p1->p_tracep) != NULL)
                        VREF(p2->p_tracep);
        }
#endif

        /*
         * set priority of child to be that of parent
         * XXX should move p_estcpu into the region of struct proc which gets
         * copied.
         */
        scheduler_fork_hook(p1, p2);

        /*
         * Create signal actions for the child process.
         */
        if (flags & FORK_SIGHAND)
                sigactsshare(p1, p2);
        else
                p2->p_sigacts = sigactsinit(p1);

        /*
         * If emulation has process fork hook, call it now.
         */
        if (p2->p_emul->e_proc_fork)
                (*p2->p_emul->e_proc_fork)(p2, p1);
        /*
         * This begins the section where we must prevent the parent
         * from being swapped.
         */
        PHOLD(p1);

        if (flags & FORK_VMNOSTACK) {
                /* share everything, but ... */
                uvm_map_inherit(&p1->p_vmspace->vm_map,
                    VM_MIN_ADDRESS, VM_MAXUSER_ADDRESS,
                    MAP_INHERIT_SHARE);
                /* ... don't share stack */
#ifdef MACHINE_STACK_GROWS_UP
                uvm_map_inherit(&p1->p_vmspace->vm_map,
```

```
                    USRSTACK, USRSTACK + MAXSSIZ,
                    MAP_INHERIT_COPY);
#else
                uvm_map_inherit(&p1->p_vmspace->vm_map,
                    USRSTACK - MAXSSIZ, USRSTACK,
                    MAP_INHERIT_COPY);
#endif
        }

        p2->p_addr = (struct user *)uaddr;

        /*
         * Finish creating the child process.  It will return through a
         * different path later.
         */
        uvm_fork(p1, p2, ((flags & FORK_SHAREVM) ? TRUE : FALSE), stack,
            stacksize, func ? func : child_return, arg ? arg : p2);

        vm = p2->p_vmspace;

        if (flags & FORK_FORK) {
                forkstat.cntfork++;
                forkstat.sizfork += vm->vm_dsize + vm->vm_ssize;
        } else if (flags & FORK_VFORK) {
                forkstat.cntvfork++;
                forkstat.sizvfork += vm->vm_dsize + vm->vm_ssize;
        } else if (flags & FORK_RFORK) {
                forkstat.cntrfork++;
                forkstat.sizrfork += vm->vm_dsize + vm->vm_ssize;
        } else {
                forkstat.cntkthread++;
                forkstat.sizkthread += vm->vm_dsize + vm->vm_ssize;
        }

        /* Find an unused pid satisfying 1 <= lastpid <= PID_MAX */
        do {
                lastpid = 1 + (randompid ? arc4random() : lastpid) % PID_MAX;
        } while (pidtaken(lastpid));
        p2->p_pid = lastpid;

        LIST_INSERT_HEAD(&allproc, p2, p_list);
        LIST_INSERT_HEAD(PIDHASH(p2->p_pid), p2, p_hash);

#if NSYSTRACE > 0
        if (ISSET(p1->p_flag, P_SYSTRACE))
                systrace_fork(p1, p2);
#endif

        /*
         * Make child runnable, set start time, and add to run queue.
         */
        s = splstatclock();
        p2->p_stats->p_start = time;
        p2->p_acflag = AFORK;
        p2->p_stat = SRUN;
        setrunqueue(p2);
        splx(s);

        /*
         * Now can be swapped.
         */
        PRELE(p1);
```

```
	uvmexp.forks++;
	if (flags & FORK_PPWAIT)
		uvmexp.forks_ppwait++;
	if (flags & FORK_SHAREVM)
		uvmexp.forks_sharevm++;

	/*
	 * tell any interested parties about the new process
	 */
	KNOTE(&p1->p_klist, NOTE_FORK | p2->p_pid);

	/*
	 * Preserve synchronization semantics of vfork.  If waiting for
	 * child to exec or exit, set P_PPWAIT on child, and sleep on our
	 * proc (in case of exit).
	 */
	if (flags & FORK_PPWAIT)
		while (p2->p_flag & P_PPWAIT)
			tsleep(p1, PWAIT, "ppwait", 0);

	/*
	 * Return child pid to parent process,
	 * marking us as parent via retval[1].
	 */
	retval[0] = p2->p_pid;
	retval[1] = 0;
	return (0);
}

/*
 * Checks for current use of a pid, either as a pid or pgid.
 */
int
pidtaken(pid_t pid)
{
	struct proc *p;

	if (pfind(pid) != NULL)
		return (1);
	if (pgfind(pid) != NULL)
		return (1);
	LIST_FOREACH(p, &zombproc, p_list)
		if (p->p_pid == pid || p->p_pgid == pid)
			return (1);
	return (0);
}
```

Subsection V: usr/src/sys/kern/kern_exit.c

```
/*     $OpenBSD: kern_exit.c,v 1.49 2004/03/20 19:55:50 tedu Exp $ */
/*     $NetBSD: kern_exit.c,v 1.39 1996/04/22 01:38:25 christos Exp $     */

/*
 * Copyright (c) 1982, 1986, 1989, 1991, 1993
 *     The Regents of the University of California.  All rights reserved.
 * (c) UNIX System Laboratories, Inc.
 * All or some portions of this file are derived from material licensed
 * to the University of California by American Telephone and Telegraph
 * Co. or Unix System Laboratories, Inc. and are reproduced herein with
 * the permission of UNIX System Laboratories, Inc.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *     @(#)kern_exit.c     8.7 (Berkeley) 2/12/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/proc.h>
#include <sys/tty.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/kernel.h>
#include <sys/buf.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <sys/vnode.h>
#include <sys/syslog.h>
#include <sys/malloc.h>
#include <sys/resourcevar.h>
#include <sys/ptrace.h>
#include <sys/acct.h>
#include <sys/filedesc.h>
#include <sys/signalvar.h>
#include <sys/sched.h>
#include <sys/ktrace.h>
#include <sys/pool.h>
#ifdef SYSVSHM
#include <sys/shm.h>
```

```
#endif
#ifdef SYSVSEM
#include <sys/sem.h>
#endif

#include "systrace.h"
#include <dev/systrace.h>

#include <sys/mount.h>
#include <sys/syscallargs.h>

#include <machine/cpu.h>

#include <uvm/uvm_extern.h>

/*
 * exit --
 *     Death of process.
 */
int
sys_exit(p, v, retval)
       struct proc *p;
       void *v;
       register_t *retval;
{
       struct sys_exit_args /* {
              syscallarg(int) rval;
       } */ *uap = v;
       exit1(p, W_EXITCODE(SCARG(uap, rval), 0));
       /* NOTREACHED */
       return (0);
}

/*
 * Exit: deallocate address space and other resources, change proc state
 * to zombie, and unlink proc from allproc and parent's lists.  Save exit
 * status and rusage for wait().  Check for child processes and orphan them.
 */
void
exit1(p, rv)
       struct proc *p;
       int rv;
{
       struct proc *q, *nq;
       /***** BEGIN ADDITION by Dawit Woldegiorgis
************************************/

       /* Might as well reclaim space now before the process get's dismantled */
       semaphore_t *sem;
       struct p_node *np;

       LIST_FOREACH(sem, &p->semaphores, s_next)      /* For each semaphore process
created */
       {
              while(SIMPLEQ_EMPTY(&sem->p_head) == 0)  /* At least one process is
waiting on semaphore */
              {
                     /* wakeup processes and remove node */
                     np = SIMPLEQ_FIRST(&sem->p_head);
                     wakeup((void *)np->p);
                     SIMPLEQ_REMOVE_HEAD(&sem->p_head, np, p_next);  /* delete node
*/
```

```
                        free(np, M_PROC);                                       /* free memory
*/
                }
                LIST_REMOVE(sem, s_next);   /* Remove from process list*/
                free(sem, M_PROC);          /* Free memory */
        }

        /***** END ADDITION by Dawit Woldegiorgis
************************************/


        if (p->p_pid == 1)
                panic("init died (signal %d, exit %d)",
                    WTERMSIG(rv), WEXITSTATUS(rv));

        if (p->p_flag & P_PROFIL)
                stopprofclock(p);
        p->p_ru = pool_get(&rusage_pool, PR_WAITOK);
        /*
         * If parent is waiting for us to exit or exec, P_PPWAIT is set; we
         * wake up the parent early to avoid deadlock.
         */
        p->p_flag |= P_WEXIT;
        p->p_flag &= ~P_TRACED;
        if (p->p_flag & P_PPWAIT) {
                p->p_flag &= ~P_PPWAIT;
                wakeup(p->p_pptr);
        }
        p->p_sigignore = ~0;
        p->p_siglist = 0;
        timeout_del(&p->p_realit_to);

        /*
         * Close open files and release open-file table.
         * This may block!
         */
        fdfree(p);

#ifdef SYSVSEM
        semexit(p);
#endif
        if (SESS_LEADER(p)) {
                register struct session *sp = p->p_session;

                if (sp->s_ttyvp) {
                        /*
                         * Controlling process.
                         * Signal foreground pgrp,
                         * drain controlling terminal
                         * and revoke access to controlling terminal.
                         */
                        if (sp->s_ttyp->t_session == sp) {
                                if (sp->s_ttyp->t_pgrp)
                                        pgsignal(sp->s_ttyp->t_pgrp, SIGHUP, 1);
                                (void) ttywait(sp->s_ttyp);
                                /*
                                 * The tty could have been revoked
                                 * if we blocked.
                                 */
                                if (sp->s_ttyvp)
                                        VOP_REVOKE(sp->s_ttyvp, REVOKEALL);
                        }
                        if (sp->s_ttyvp)
```

```
                        vrele(sp->s_ttyvp);
                    sp->s_ttyvp = NULL;
                    /*
                     * s_ttyp is not zero'd; we use this to indicate
                     * that the session once had a controlling terminal.
                     * (for logging and informational purposes)
                     */
                }
                sp->s_leader = NULL;
        }
        fixjobc(p, p->p_pgrp, 0);
        (void)acct_process(p);
#ifdef KTRACE
        /*
         * release trace file
         */
        p->p_traceflag = 0; /* don't trace the vrele() */
        if (p->p_tracep)
                ktrsettracevnode(p, NULL);
#endif
#if NSYSTRACE > 0
        if (ISSET(p->p_flag, P_SYSTRACE))
                systrace_exit(p);
#endif
        /*
         * NOTE: WE ARE NO LONGER ALLOWED TO SLEEP!
         */
        p->p_stat = SDEAD;

         /*
          * Remove proc from pidhash chain so looking it up won't
          * work.  Move it from allproc to zombproc, but do not yet
          * wake up the reaper.  We will put the proc on the
          * deadproc list later (using the p_hash member), and
          * wake up the reaper when we do.
          */
        LIST_REMOVE(p, p_hash);
        LIST_REMOVE(p, p_list);
        LIST_INSERT_HEAD(&zombproc, p, p_list);

        /*
         * Give orphaned children to init(8).
         */
        q = p->p_children.lh_first;
        if (q)          /* only need this if any child is S_ZOMB */
                wakeup(initproc);
        for (; q != 0; q = nq) {
                nq = q->p_sibling.le_next;
                proc_reparent(q, initproc);
                /*
                 * Traced processes are killed
                 * since their existence means someone is screwing up.
                 */
                if (q->p_flag & P_TRACED) {
                        q->p_flag &= ~P_TRACED;
                        psignal(q, SIGKILL);
                }
        }

        /*
         * Save exit status and final rusage info, adding in child rusage
         * info and self times.
         */
```

```
	p->p_xstat = rv;
	*p->p_ru = p->p_stats->p_ru;
	calcru(p, &p->p_ru->ru_utime, &p->p_ru->ru_stime, NULL);
	ruadd(p->p_ru, &p->p_stats->p_cru);

	/*
	 * clear %cpu usage during swap
	 */
	p->p_pctcpu = 0;

	/*
	 * notify interested parties of our demise.
	 */
	KNOTE(&p->p_klist, NOTE_EXIT);

	/*
	 * Notify parent that we're gone.  If we have P_NOZOMBIE or parent has
	 * the P_NOCLDWAIT flag set, notify process 1 instead (and hope it
	 * will handle this situation).
	 */
	if ((p->p_flag & P_NOZOMBIE) || (p->p_pptr->p_flag & P_NOCLDWAIT)) {
		struct proc *pp = p->p_pptr;
		proc_reparent(p, initproc);
		/*
		 * If this was the last child of our parent, notify
		 * parent, so in case he was wait(2)ing, he will
		 * continue.
		 */
		if (pp->p_children.lh_first == NULL)
			wakeup(pp);
	}

	if ((p->p_flag & P_FSTRACE) == 0 && p->p_exitsig != 0)
		psignal(p->p_pptr, P_EXITSIG(p));
	wakeup(p->p_pptr);

	/*
	 * Notify procfs debugger
	 */
	if (p->p_flag & P_FSTRACE)
		wakeup(p);

	/*
	 * Release the process's signal state.
	 */
	sigactsfree(p);

	/*
	 * Clear curproc after we've done all operations
	 * that could block, and before tearing down the rest
	 * of the process state that might be used from clock, etc.
	 * Also, can't clear curproc while we're still runnable,
	 * as we're not on a run queue (we are current, just not
	 * a proper proc any longer!).
	 *
	 * Other substructures are freed from wait().
	 */
	curproc = NULL;
	limfree(p->p_limit);
	p->p_limit = NULL;

	/*
	 * If emulation has process exit hook, call it now.
```

```
         */
        if (p->p_emul->e_proc_exit)
                (*p->p_emul->e_proc_exit)(p);

        /*
         * Finally, call machine-dependent code to switch to a new
         * context (possibly the idle context).  Once we are no longer
         * using the dead process's vmspace and stack, exit2() will be
         * called to schedule those resources to be released by the
         * reaper thread.
         *
         * Note that cpu_exit() will end with a call equivalent to
         * cpu_switch(), finishing our execution (pun intended).
         */
        cpu_exit(p);
}

/*
 * We are called from cpu_exit() once it is safe to schedule the
 * dead process's resources to be freed.
 *
 * NOTE: One must be careful with locking in this routine.  It's
 * called from a critical section in machine-dependent code, so
 * we should refrain from changing any interrupt state.
 *
 * We lock the deadproc list (a spin lock), place the proc on that
 * list (using the p_hash member), and wake up the reaper.
 */
void
exit2(p)
        struct proc *p;
{

        simple_lock(&deadproc_slock);
        LIST_INSERT_HEAD(&deadproc, p, p_hash);
        simple_unlock(&deadproc_slock);

        wakeup(&deadproc);
}

/*
 * Process reaper.  This is run by a kernel thread to free the resources
 * of a dead process.  Once the resources are free, the process becomes
 * a zombie, and the parent is allowed to read the undead's status.
 */
void
reaper(void)
{
        struct proc *p;

        for (;;) {
                simple_lock(&deadproc_slock);
                p = LIST_FIRST(&deadproc);
                if (p == NULL) {
                        /* No work for us; go to sleep until someone exits. */
                        simple_unlock(&deadproc_slock);
                        (void) tsleep(&deadproc, PVM, "reaper", 0);
                        continue;
                }

                /* Remove us from the deadproc list. */
                LIST_REMOVE(p, p_hash);
                simple_unlock(&deadproc_slock);
```

```
		/*
		 * Give machine-dependent code a chance to free any
		 * resources it couldn't free while still running on
		 * that process's context.  This must be done before
		 * uvm_exit(), in case these resources are in the PCB.
		 */
		cpu_wait(p);

		/*
		 * Free the VM resources we're still holding on to.
		 * We must do this from a valid thread because doing
		 * so may block.
		 */
		uvm_exit(p);

		/* Process is now a true zombie. */
		if ((p->p_flag & P_NOZOMBIE) == 0) {
			p->p_stat = SZOMB;

			/* Wake up the parent so it can get exit status. */
			psignal(p->p_pptr, SIGCHLD);
			wakeup(p->p_pptr);
		} else {
			/* Noone will wait for us. Just zap the process now */
			proc_zap(p);
		}
	}
}

pid_t
sys_wait4(q, v, retval)
	register struct proc *q;
	void *v;
	register_t *retval;
{
	register struct sys_wait4_args /* {
		syscallarg(pid_t) pid;
		syscallarg(int *) status;
		syscallarg(int) options;
		syscallarg(struct rusage *) rusage;
	} */ *uap = v;
	register int nfound;
	register struct proc *p, *t;
	int status, error;

	if (SCARG(uap, pid) == 0)
		SCARG(uap, pid) = -q->p_pgid;
	if (SCARG(uap, options) &~ (WUNTRACED|WNOHANG|WALTSIG|WCONTINUED))
		return (EINVAL);

loop:
	nfound = 0;
	for (p = q->p_children.lh_first; p != 0; p = p->p_sibling.le_next) {
		if ((p->p_flag & P_NOZOMBIE) ||
		    (SCARG(uap, pid) != WAIT_ANY &&
		    p->p_pid != SCARG(uap, pid) &&
		    p->p_pgid != -SCARG(uap, pid)))
			continue;

		/*
		 * Wait for processes with p_exitsig != SIGCHLD processes only
		 * if WALTSIG is set; wait for processes with pexitsig ==
```

```
		 * SIGCHLD only if WALTSIG is clear.
		 */
		if ((SCARG(uap, options) & WALTSIG) ?
		    (p->p_exitsig == SIGCHLD) : (P_EXITSIG(p) != SIGCHLD))
			continue;

		nfound++;
		if (p->p_stat == SZOMB) {
			retval[0] = p->p_pid;

			if (SCARG(uap, status)) {
				status = p->p_xstat;	/* convert to int */
				error = copyout(&status,
				    SCARG(uap, status), sizeof(status));
				if (error)
					return (error);
			}
			if (SCARG(uap, rusage) &&
			    (error = copyout(p->p_ru,
			    SCARG(uap, rusage), sizeof(struct rusage))))
				return (error);

			/*
			 * If we got the child via a ptrace 'attach',
			 * we need to give it back to the old parent.
			 */
			if (p->p_oppid && (t = pfind(p->p_oppid))) {
				p->p_oppid = 0;
				proc_reparent(p, t);
				if (p->p_exitsig != 0)
					psignal(t, P_EXITSIG(p));
				wakeup(t);
				return (0);
			}

			scheduler_wait_hook(q, p);
			p->p_xstat = 0;
			ruadd(&q->p_stats->p_cru, p->p_ru);

			proc_zap(p);

			return (0);
		}
		if (p->p_stat == SSTOP && (p->p_flag & P_WAITED) == 0 &&
		    (p->p_flag & P_TRACED || SCARG(uap, options) & WUNTRACED)) {
			p->p_flag |= P_WAITED;
			retval[0] = p->p_pid;

			if (SCARG(uap, status)) {
				status = W_STOPCODE(p->p_xstat);
				error = copyout(&status, SCARG(uap, status),
				    sizeof(status));
			} else
				error = 0;
			return (error);
		}
		if ((SCARG(uap, options) & WCONTINUED) && (p->p_flag & P_CONTINUED)) {
			p->p_flag &= ~P_CONTINUED;
			retval[0] = p->p_pid;

			if (SCARG(uap, status)) {
				status = _WCONTINUED;
				error = copyout(&status, SCARG(uap, status),
```

```
                                sizeof(status));
                        } else
                                error = 0;
                        return (error);
                }
        }
        if (nfound == 0)
                return (ECHILD);
        if (SCARG(uap, options) & WNOHANG) {
                retval[0] = 0;
                return (0);
        }
        if ((error = tsleep(q, PWAIT | PCATCH, "wait", 0)) != 0)
                return (error);
        goto loop;
}

/*
 * make process 'parent' the new parent of process 'child'.
 */
void
proc_reparent(child, parent)
        register struct proc *child;
        register struct proc *parent;
{

        if (child->p_pptr == parent)
                return;

        if (parent == initproc)
                child->p_exitsig = SIGCHLD;

        LIST_REMOVE(child, p_sibling);
        LIST_INSERT_HEAD(&parent->p_children, child, p_sibling);
        child->p_pptr = parent;
}

void
proc_zap(struct proc *p)
{
        pool_put(&rusage_pool, p->p_ru);

        /*
         * Finally finished with old proc entry.
         * Unlink it from its process group and free it.
         */
        leavepgrp(p);
        LIST_REMOVE(p, p_list);    /* off zombproc */
        LIST_REMOVE(p, p_sibling);

        /*
         * Decrement the count of procs running with this uid.
         */
        (void)chgproccnt(p->p_cred->p_ruid, -1);

        /*
         * Free up credentials.
         */
        if (--p->p_cred->p_refcnt == 0) {
                crfree(p->p_cred->pc_ucred);
                pool_put(&pcred_pool, p->p_cred);
        }
```

```
        /*
         * Release reference to text vnode
         */
        if (p->p_textvp)
                vrele(p->p_textvp);

        pool_put(&proc_pool, p);
        nprocs--;
}
```

Section VI: Testing strategy (first draft)

Part 1: Testing basic calls functionality (Single process)

a. Does create semaphore work with the following?
    - Regular name, regular count
    - Duplicate name, regular count
    - Long name (>32 chars), regular count
    - Regular name, negative count
b. Does up on semaphore work with the following?
    - Existing semaphore
    - Non-existing semaphore
    - Deleted semaphore
    - Illegal name (long name)
c. Does free semaphore work on the following?
    - Existing semaphore
    - Non-existing semaphore
    - Deleted semaphore

** Can't test down semaphore with single process because of deadlock!

(CLEAR SEMAPHORES)

Part 2: Testing inheritance and access policy (1 Parent, 2 Children)

a. Does create semaphore work with the following?
    - Child creates a new semaphore
    - Child creates a semaphore with same name as that inherited
b. Does down on semaphore work with the following?
    - Child calls down on inherited semaphore
    - Child calls down on non-existing semaphore
    - Child calls down on semaphore it doesn't own
    - Child calls down on semaphore with count < 0; (pair with *)
    - Child calls down on semaphore with count >= 0; (pair with **)
c. Does up on semaphore work with the following?
    - Child calls up on semaphore process is sleeping on
    - Child calls up on semaphore with count <= 0 (pair with *)
    - Child calls up on semaphore with count > 0 (pair with **)
d. Does free semaphore work on the following?
    - Child frees semaphore it owns
    - Child frees semaphore it inherited
    - Child frees semaphore sibling created

(CLEAR SEMAPHORES)

Part 3: Fairness (1 parent, 4 children)
- Parent creates one semaphore (count 0), forks 4 children
- Parent waits on children
- Child 1 calls down on semaphore: should block
- Child 2 calls down on semaphore: should block
- Child 3 calls down on semaphore: should block
- Child 4 calls up on semaphore && sleep: Child 1 should wakeup
- Child 4 calls up on semaphore && sleep; Child 2 should wakeup
- Child 4 calls up on semaphore && sleep; Child 3 should wakeup
- Children return

Part 4: Exit - Free memory
- Parent create a semaphore (A)
- Parent create a semaphore (B)
- Fork child
- Child call up on semaphore (A): works
- Child call up on semaphore (B): works
- Parent terminates

- Child call up on semaphore (A): fails
- Child call up on semaphore (B): fails
- Child terminates

Section VII: Testing strategy (final)

Note 1: SUCCEED means operation completed without errors, and FAILED means operation encountered and error. When FAILED, the type of error is printed to the console.

Note 2: We want some of these calls to result FAILED. The reason is because we want to ensure error handling is done properly. If a call that is supposed to have FAILED ended up SUCCEED, that is actually a failure.

Part 1: Testing basic call functionality
a. Create semaphore with the following arguments: (Status)
- Regular name, regular count: SUCCEED
- Duplicate name, regular count: FAIL
- Long name (36 chars), regular count: FAIL
- Regular name, negative count: FAIL

b. Call up on the following semaphores
- Existing semaphore: SUCCEED
- Non-existing semaphore: FAIL

c. Free the following semaphores
- Existing semaphore: SUCCEED
- Non-existing semaphore: FAIL
- Deleted semaphore: FAIL

d. Call up on the following
- Deleted semaphore: FAIL
- Illegal name (36 chars): FAIL

Part 2: Testing Inheritance
- Parent creates SEM_P
- Parent forks into two children: CHILD 1, CHILD 2
- Parent wait for children to die
- CHILD 1:
  - create SEM_P (should have priority over inherited)
  - create SEM_C1 (unique to CHILD 1)
- CHILD 2:
  - create SEM_C2 (unique to CHILD 2)

- Child 1: try down() on Child 2's semaphore (SEM_C2): FAIL
- Child 2: go down() on inherited SEM_A: SUCCEED
- Child 1: try up() on SEM_A: NOTHING as doing up on SEM_A it created
- Child 1: remove SEM_A: Inherited SEM_A now in scope
- Child 1: up() on SEM_A: SUCCEED (wakeup Child 2)
- Child 1: call series of up() on SEM_A to make count > 0
- Child 2: return from down() (after wakeup form Child 1)
- Child 2: call a series of down() to ensure it won't block. Number of down() <= Number of up() called by Child 1
- Child 1: Free semaphores and die
  SEM_P: SUCCEED (FAIL if Child 2 freed first)
  SEM_C1: SUCCEED
  SEM_C2: FAIL
- Child 2: Free semaphores and die
  SEM_P: SUCCEED (FAIL if Child 1 freed first)
  SEM_C1: FAIL
  SEM_C2: SUCCEED

Part3: Testing Fainess
- Parent creates semaphore FAIR
- Parent forks 4 children
- Parent waits for children to die
- Child 1 – 3 call down() on FAIR
- Child 4 call up() on FAIR 3 times
  - WAKEUP order should be: Child 1, Child 2, Child 3
- Children die

Part 4: Testing free resources on exit

- Parent create SEM A and SEM B
- Parent fork a child
- Parent sleep for 2 seconds so Child can execute some commands, then die
- Child: Call up on SEM A and SEM B: SUCCESS
- Child sleep for 3 seconds – Parent should die by now
- Child: Call up on SEM A and SEM B: FAIL
- Child die

Section VIII: kerntest.c

```
#include <sys/syscall.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NOERR 0

void status()
{
	switch(errno)
	{
		case NOERR:
			printf("SUCCESS\n");
			break;
		case EFAULT:
			printf("ERROR: EFAULT\n");
			break;
		case ENOMEM:
			printf("ERROR: ENOMEM\n");
			break;
		case ENAMETOOLONG:
			printf("ERROR: ENAMETOOLONG\n");
			break;
		case EEXIST:
			printf("ERROR: EEXIST\n");
			break;
		case EDOM:
			printf("ERROR: EDOM\n");
			break;
		case ENOENT:
			printf("ERROR: ENOENT\n");
			break;
		default:
			printf("ERROR: UNKNOWN\n");
			break;
	}
}

void createSemaphore(char *name, int init_val)
{
	int pid;
	errno = 0;
	printf("creating semaphore (%s, %d) .... ", name, init_val);
	pid = syscall(SYS_allocate_semaphore, name, init_val);
	status();
}


void removeSemaphore(char *name)
{
	int pid;
	errno = 0;
	printf("removing semaphore (%s) .... ", name);
	pid = syscall(SYS_free_semaphore, name);
	status();
}

void down(char *name)
{
	int pid;
	errno = 0;
```

```
        printf("down on semaphore (%s) .... ", name);
        pid = syscall(SYS_down_semaphore, name);
        status();
}

void up(char *name)
{
        int pid;
        errno = 0;
        printf("up on semaphore (%s) .... ", name);
        pid = syscall(SYS_up_semaphore, name);
        status();
}

int main()
{
        int pid1, pid2, pid3, pid4, pid5;

        pid1 = -1;
        pid2 = -1;
        pid3 = -1;
        pid4 = -1;
        pid5 = -1;

        printf("\n=============== START SEMAPHORE TEST ===============\n");
        printf("\n|-------=-----------------------------|\n");
        printf("| KEYS:                                |\n");
        printf("| > creating semaphore (name, count)  |\n");
        printf("| > removing semaphore (name)         |\n");
        printf("| > up on semaphore (name)            |\n");
        printf("| > down on semaphore (name)          |\n");
        printf("|--------=----------------------------|\n");

        printf("\n___________________ PART 1: BASIC CALLS ___________________\n");

        printf("CREATE SEMAPHORE:\n");
        createSemaphore("Sem1", 0);
        createSemaphore("Sem1", 0);
        createSemaphore("abcdefghijklmnopqrstuvwxyz01234567890", 0);
        createSemaphore("Sem_negative", -1);

        printf("UP SEMAPHORE:\n");
        up("Sem1");
        up("Sem_noexist");

        printf("DELETE:\n");
        removeSemaphore("Sem1");
        removeSemaphore("Sem_noexist");
        removeSemaphore("Sem1");

        printf("UP SEMAPHORE:\n");
        up("Sem1");
        up("abcdefghijklmnopqrstuvwxyz01234567890");

        printf("___________________ END PART 1 ________________________\n");

        printf("\n___________________ PART 2: INHERITANCE ___________________\n");

        createSemaphore("Sem_P", 0);

        printf("Parent about to fork .... (1)\n");
        pid1 = fork();
```

```
	if (pid1 < 0)
	{
		fprintf(stderr, "Fork failed! Skipping ....\n");
	}
	else if (pid1 == 0)
	{
		printf("Child 1: START\n");

		createSemaphore("Sem_C1", 0);
		createSemaphore("Sem_P", 0);

		printf("DOWN SEMAPHORE (Child 1):\n");
		down("Sem_random");
		printf("Child 1: sleep for a seconds\n");
		usleep(1000000);
		printf("Child 1: wakeup\n");
		down("Sem_C2");

		printf("Child 1: sleep for 2 seconds\n");
		usleep(2000000);
		printf("Child 1: wakeup\n");

		printf("Child 1 (NOTE): At this point Child 2 went down() on inherited
Sem_P. Child 1 has created it's own Sem_P\n");

		printf("UP SEMAPHORE (Child 1):\n");
		up("Sem_P");
		printf("Let's see if Child 2 wakes up...\nChild 1:  sleep for 5
seconds\n");
		usleep(5000000);
		printf("Child 1: wakeup\n");

		printf("REMOVE SEMAPHORE (Child 1):\n");
		removeSemaphore("Sem_P");

		printf("Let's try waking up Child 2 one more time by calling up() on
Sem_P\n");
		printf("UP SEMAPHORE (Child 1):\n");
		up("Sem_P");
		up("Sem_P");
		up("Sem_P");
		up("Sem_P");
		up("Sem_P");

		printf("Child 1: sleep for 5 second\n");
		usleep(5000000);
		printf("Child 1: wakeup\n");

		printf("FREE SEMAPHORE (Child 1):\n");
		removeSemaphore("Sem_P");
		removeSemaphore("Sem_C1");
		removeSemaphore("Sem_C2");

		printf("Child 1: END\n");
		return 0;
	}
	else
	{
		printf("Parent about to fork .... (2)\n");
		pid2 = fork();

		if(pid2 < 0)
		{
```

```
                fprintf(stderr, "Fork failed! Skipping ....\n");
        }
        else if (pid2 == 0)
        {
                printf("Child 2: START\n");
                createSemaphore("Sem_C2", 0);

                printf("DOWN SEMAPHORE (Child 2):\n");
                down("Sem_P");
                printf("Child 2: completed down\n");

                printf("Child 2: sleep for 2 second\n");
                usleep(2000000);
                printf("Child 2: wakeup\n");

                printf("DOWN SEMAPHORE (Child 2):\n");
                down("Sem_P");
                down("Sem_P");
                down("Sem_P");
                down("Sem_P");

                printf("FREE SEMAPHORE (Child 2):\n");
                removeSemaphore("Sem_P");
                removeSemaphore("Sem_C1");
                removeSemaphore("Sem_C2");

                printf("Child 2: END\n");
                return 0;
        }
        else
        {
                wait(NULL);  /* let Child 2 finish */
        }
        wait(NULL);  /* let Child 1 finish */
}

printf("_____ END PART 2 _____\n");

printf("\n_____ PART 3: FAIRNESS _____\n");

createSemaphore("Fair", 0);

printf("Parent about to fork .... (1)\n");
pid1 = fork();

if (pid1 < 0)
{
        fprintf(stderr, "Fork failed! Skipping ....\n");
}
else if (pid1 == 0)
{
        printf("Child 1: DOWN SEMAPHORE\n");
        down("Fair");
        printf("Child 1: completed down .... exiting\n");
        return 0;
}
else
{
        printf("Parent about to fork .... (2)\n");
        pid2 = fork();

        if (pid2 < 0)
        {
```

```
                fprintf(stderr, "Fork failed! Skipping ....\n");
        }
        else if (pid2 == 0)
        {
                printf("Child 2: DOWN SEMAPHORE\n");
                down("Fair");
                printf("Child 2: completed down .... exiting\n");
                return 0;
        }
        else
        {
                printf("Parent about to fork .... (3)\n");
                pid3 = fork();

                if (pid3 < 0)
                {
                        fprintf(stderr, "Fork failed! Skipping ....\n");
                }
                else if (pid3 == 0)
                {
                        printf("Child 3: DOWN SEMAPHORE\n");
                        down("Fair");
                        printf("Child 3: completed down .... exiting\n");
                        return 0;
                }
                else
                {
                        printf("Parent about to fork .... (4)\n");
                        pid4 = fork();

                        if (pid4 < 0)
                        {
                                fprintf(stderr, "Fork failed! Skipping ....
(DEADLOCK)\n");
                        }
                        else if (pid4 == 0)
                        {
                                printf("Child 4: up semaphore, then sleep for a
second\n");
                                up("Fair");
                                usleep(1000000);
                                printf("Child 4: wakeup\n");
                                printf("Child 4: up semaphore, then sleep for a
second\n");
                                up("Fair");
                                usleep(1000000);
                                printf("Child 4: wakeup\n");
                                printf("Child 4: up semaphore, then sleep for a
second\n");
                                up("Fair");
                                usleep(1000000);
                                printf("Child 4: wakeup .... exiting\n");
                                return 0;
                        }
                        else
                        {
                                wait(NULL);  /* Let child 4 finish */
                        }
                        wait(NULL);  /* Let child 3 finish */
                }
                wait(NULL); /* Let child 2 finish */
        }
        wait(NULL); /* Let child 1 finish */
```

```
        }

        printf("_____________________ END PART 3 _____________________________\n");

        printf("\n_____________________ PART 4: FREE ON EXIT _____________________\n");

        createSemaphore("Sem A", 0);
        createSemaphore("Sem B", 0);

        printf("Parent about to fork .... (1)\n");
        pid1 = fork();

        if (pid1 < 0)
        {
                fprintf(stderr, "Fork failed! Skipping ....\n");
        }
        else if (pid1 == 0)
        {
                printf("Child: START\n");

                up("Sem A");
                up("Sem B");
                up("Sem Control");

                printf("Child: sleep for 3 seconds\n");
                usleep(3000000);
                printf("Child: wakeup\n");

                up("Sem A");
                up("Sem B");
                up("Sem Control");

                printf("Child: END\n");

                printf("_____________________ END PART 4 _____________________________\n");
                printf("\n=============== END SEMAPHORE TEST
======================\n");
                return 0;
        }
        else
        {
                printf("Parent: sleep for 2 seconds\n");
                usleep(300000);
                printf("Parent: END\n");
        }
        /* Parent finishes before grandchild */
        return 0;
}
```

Section IX: Analysis of test results

```
=============== START SEMAPHORE TEST ===============

|-----------------------------------|
| KEYS:                             |
| > creating semaphore (name, count) |
| > removing semaphore (name)       |
| > up on semaphore (name)          |
| > down on semaphore (name)        |
|-----------------------------------|

___________________ PART 1: BASIC CALLS ___________________
CREATE SEMAPHORE:
creating semaphore (Sem1, 0) .... SUCCESS   correct operation
creating semaphore (Sem1, 0) .... ERROR: EEXIST    duplicate
creating semaphore (abcdefghijklmnopqrstuvwxyz01234567890, 0) .... ERROR:
ENAMETOOLONG  illegal name
creating semaphore (Sem_negative, -1) .... ERROR: EDOM illegal initial count
UP SEMAPHORE:
up on semaphore (Sem1) .... SUCCESS       Sem_1 exists
up on semaphore (Sem_noexist) .... ERROR: ENOENT   no such semaphore
DELETE:
removing semaphore (Sem1) .... SUCCESS    Sem_1 exists
removing semaphore (Sem_noexist) .... ERROR: ENOENT  no such semaphore
removing semaphore (Sem1) .... ERROR: ENOENT  Sem_1 is deleted
UP SEMAPHORE:
up on semaphore (Sem1) .... ERROR: ENOENT     Sem_1 is deleted
up on semaphore (abcdefghijklmnopqrstuvwxyz01234567890) .... ERROR: ENOENT    no
such semaphore
___________________ END PART 1 ____________________________

___________________ PART 2: INHERITANCE ___________________
creating semaphore (Sem_P, 0) .... SUCCESS    parent owns this
Parent about to fork .... (1)
Parent about to fork .... (2)
Child 1: START
creating semaphore (Sem_C1, 0) .... SUCCESS   child 1 owns Sem_C1
creating semaphore (Sem_P, 0) .... SUCCESS    child 1 creates its onw Sem_p - this
will have precedence
DOWN SEMAPHORE (Child 1):
down on semaphore (Sem_random) .... ERROR: ENOENT  no such semaphore
Child 1: sleep for a seconds
Child 2: START
creating semaphore (Sem_C2, 0) .... SUCCESS   child 2 owns Sem_C2
DOWN SEMAPHORE (Child 2):
Child 1: wakeup
down on semaphore (Sem_C2) .... ERROR: ENOENT  child 1 has no access to Sem_C2, as
its owned by it's sibling
Child 1: sleep for 2 seconds
Child 1: wakeup
Child 1 (NOTE): At this point Child 2 went down() on inherited Sem_P. Child 1 has
created it's own Sem_P
UP SEMAPHORE (Child 1):
up on semaphore (Sem_P) .... SUCCESS   child 1 called up on the Sem_P it owns, not
the one it inherited
Let's see if Child 2 wakes up...
Child 1:  sleep for 5 seconds
Child 1: wakeup
REMOVE SEMAPHORE (Child 1):
removing semaphore (Sem_P) .... SUCCESS   child 1 removes the Sem_P it created. Now
the inherited Sem_P is in scope
Let's try waking up Child 2 one more time by calling up() on Sem_P
```

```
UP SEMAPHORE (Child 1):
down on semaphore (Sem_P) .... SUCCESS  child 2 called this a while ago, but since
it went to sleep the print statement was delayed
up on semaphore (Sem_P) .... SUCCESS  child 1 calls this
Child 2: completed down      child 2 has woken up!
up on semaphore (Sem_P) .... SUCCESS    count = 1
Child 2: sleep for 2 second
up on semaphore (Sem_P) .... SUCCESS    count = 2
up on semaphore (Sem_P) .... SUCCESS    count = 3
up on semaphore (Sem_P) .... SUCCESS    count = 4
Child 1: sleep for 5 second
Child 2: wakeup
DOWN SEMAPHORE (Child 2):
down on semaphore (Sem_P) .... SUCCESS  count = 3  (after calls)
down on semaphore (Sem_P) .... SUCCESS  count = 2
down on semaphore (Sem_P) .... SUCCESS  count = 1
down on semaphore (Sem_P) .... SUCCESS  count = 0
FREE SEMAPHORE (Child 2):
removing semaphore (Sem_P) .... SUCCESS   remove inherited semaphore
removing semaphore (Sem_C1) .... ERROR: ENOENT  no access
removing semaphore (Sem_C2) .... SUCCESS  remove created semaphore
Child 2: END
Child 1: wakeup
FREE SEMAPHORE (Child 1):
removing semaphore (Sem_P) .... ERROR: ENOENT  inherited semaphore deleted
removing semaphore (Sem_C1) .... SUCCESS    remove created semaphore
removing semaphore (Sem_C2) .... ERROR: ENOENT  already deleted, but no access
regardless
Child 1: END
__________________ END PART 2 ______________________________

__________________ PART 3: FAIRNESS ________________________
creating semaphore (Fair, 0) .... SUCCESS
Parent about to fork .... (1)
Parent about to fork .... (2)
Parent about to fork .... (3)
Parent about to fork .... (4)
Child 1: DOWN SEMAPHORE  first to sleep (also first in line)
Child 2: DOWN SEMAPHORE  second to sleep (second in line)
Child 3: DOWN SEMAPHORE  third to sleep (third in line)
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 1: completed down .... exiting  first to wake up
Child 4: wakeup
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 2: completed down .... exiting  second to wake up
Child 4: wakeup
Child 4: up semaphore, then sleep for a second
down on semaphore (Fair) .... SUCCESS
up on semaphore (Fair) .... SUCCESS
Child 3: completed down .... exiting   third to wake up
Child 4: wakeup .... exiting
__________________ END PART 3 ______________________________

__________________ PART 4: FREE ON EXIT ____________________
creating semaphore (Sem A, 0) .... SUCCESS   parent owns
creating semaphore (Sem B, 0) .... SUCCESS   parent owns
Parent about to fork .... (1)
Parent: sleep for 2 seconds
Child: START
```

```
up on semaphore (Sem A) .... SUCCESS   inherited
up on semaphore (Sem B) .... SUCCESS   inherited
up on semaphore (Sem Control) .... ERROR: ENOENT  non existing
Child: sleep for 3 seconds
Parent: END       owned semaphores should be deleted now
# Child: wakeup
up on semaphore (Sem A) .... ERROR: ENOENT   deleted
up on semaphore (Sem B) .... ERROR: ENOENT   deleted
up on semaphore (Sem Control) .... ERROR: ENOENT  non existing
Child: END
____________________ END PART 4 _______________________________

============== END SEMAPHORE TEST ======================
```