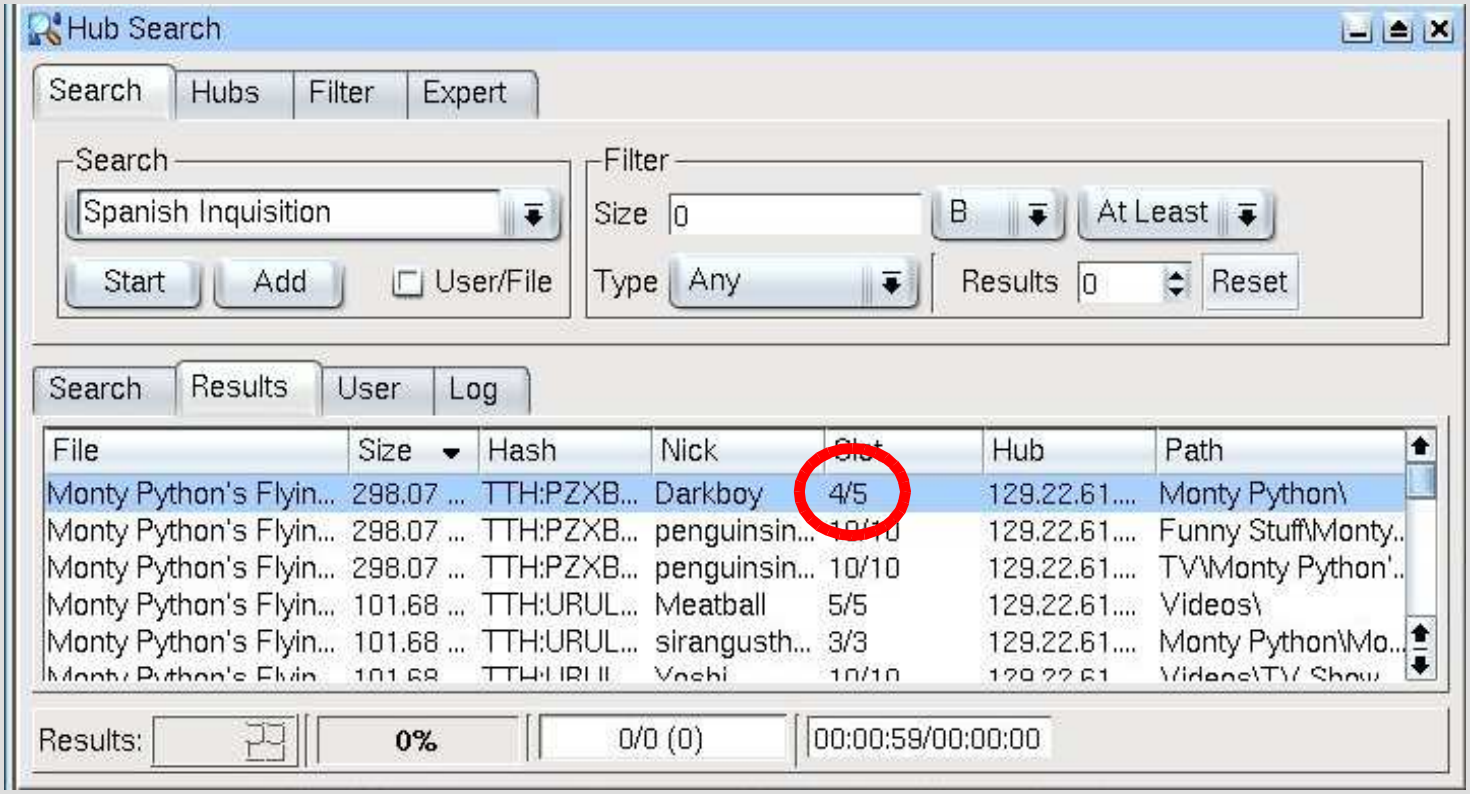
An Introduction to Semaphores

- Creating Semaphores
- Managing semaphores from the command line
- Removing Semaphores
- `union` Keyword
- The `semun` union.
- Initializing semaphores
- Finding group keys
- Using environment variables

(Unix semaphores are evil)

What is a Semaphore?

A semaphore is a counting structure that describes the number of resources remaining. This is remarkably similar to the number of available slots for users in Direct Connect.



Background and Theory

- Unix creates semaphores in groups
  - Allows programs to quickly access related semaphore sets.
  - Provides a convenient way to initialize several semaphores at the same time.
  - Creates an easy method to clean up system resources after a program crashes.
- Unix semaphores were designed by Satan
  - One function <--> 100,000 disjoint purposes
  - Syntax is usually confusing
  - Forces you to learn new keywords

Creating a New Semaphore Group:

```
int semget(IPC_PRIVATE, int numsems, IPC_CREAT | 0666)
```

This will return the key for the newly-created semaphore group (`semid`)

Informs the kernel that you want to create a new, unique semaphore group.

The number of semaphores that you want to create

`IPC_CREAT`: Informs the kernel that it is alright to create the semaphore group if it does not already exist.

`0666`: These are the permissions of the semaphore group. In this instance, everyone is granted read and write permissions.

Using a Previously Created Semaphore Group:

```
int semget(int semid, 0, 0666)
```

Returns the value of `semid`, or -1 on failure.

This is the key of the semaphore group that you want to gain access to. It will most likely be the return value of a previous call to `semget()`.

Since the group was already created, you don't need to specify the number of semaphores i the group.

The permissions for the semaphore group. Notice that `IPC_CREAT` was not set this time because we are not creating a new group.

Managing Semaphores on the Command Line

The following functions can be used on the command prompt to view and removed shared resources:

`ipcs`: Shows the the interprocess communication resources currently allocated to the system.

`ipcrm:` Allows you to remove IPC resources that had been allocated but were never freed.

```
Options: -s: Removes semaphore resource
         -m: Removes shared memory resource
```

Removing Semaphore Groups

```
int semctl(int semid, 0, IPC_RMID)
```

On failure, `semctl` will return -1 and set `errno` to the appropriate value.

This is the id of the semaphore group that was returned by the `semget` command.

Ignore this for now

This command is instructing `semctl` to remove the semaphore group.

Now Things Get More Complicated :-(

It is common knowledge that there are some programmers who spend their days in dark computer labs designing hateful data structures intended to make computer science majors suffer. This data structure was devised in the deepest and darkest of all computer labs.

`union` creates a datatype that can be of type `type_1` OR type `type_2` OR ... OR type `type_n`. It is NOT a structure containing all of the types; rather, it is a datatype that can be treated as any of its given types at a particular moment in time.

```
union UnionName
{
   type_1 Name_1;
   type_2 Name_2;
   ...
   type_n Name_n;
};
```

Another Reason that Solaris is Painful to work with:

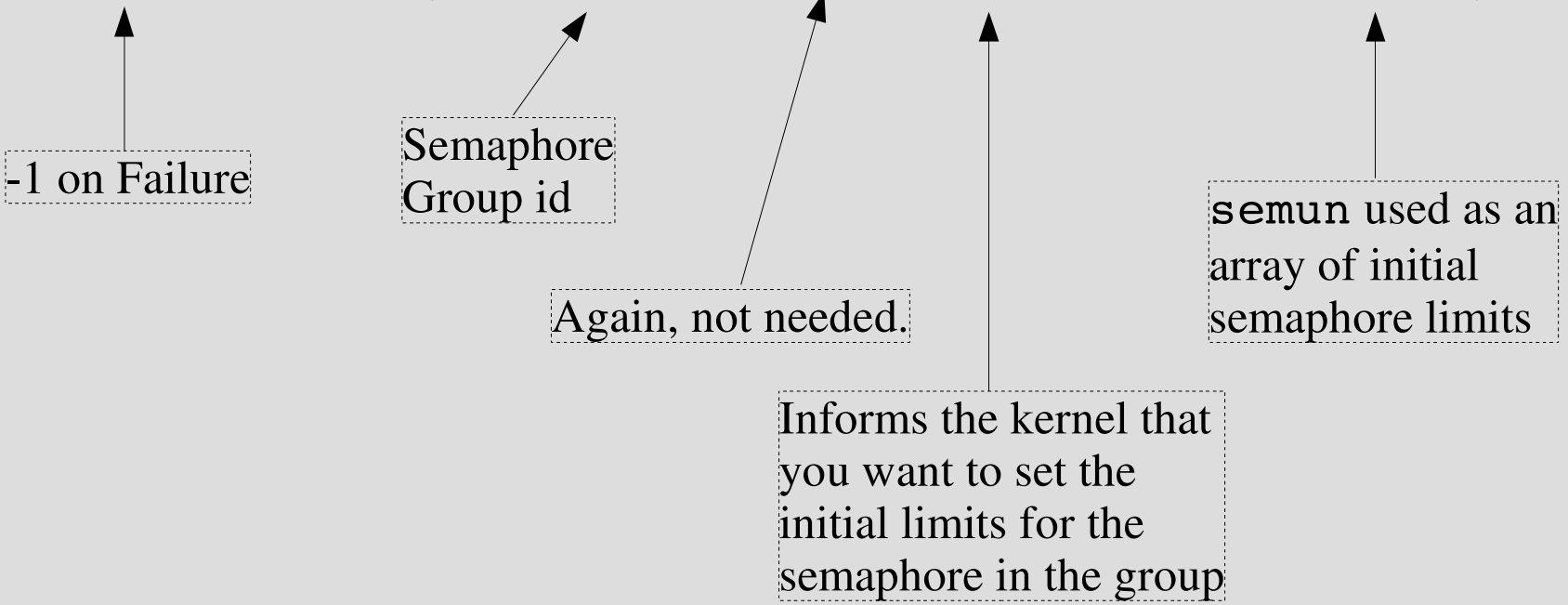
Even though many of the semaphore commands require this structure, it is not defined anywhere in the Solaris system header files. You will need to define this within your programs:

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

Creates a data structure that can be used as an integer, an array of `semid_ds` structures, or an array or unsigned integers.

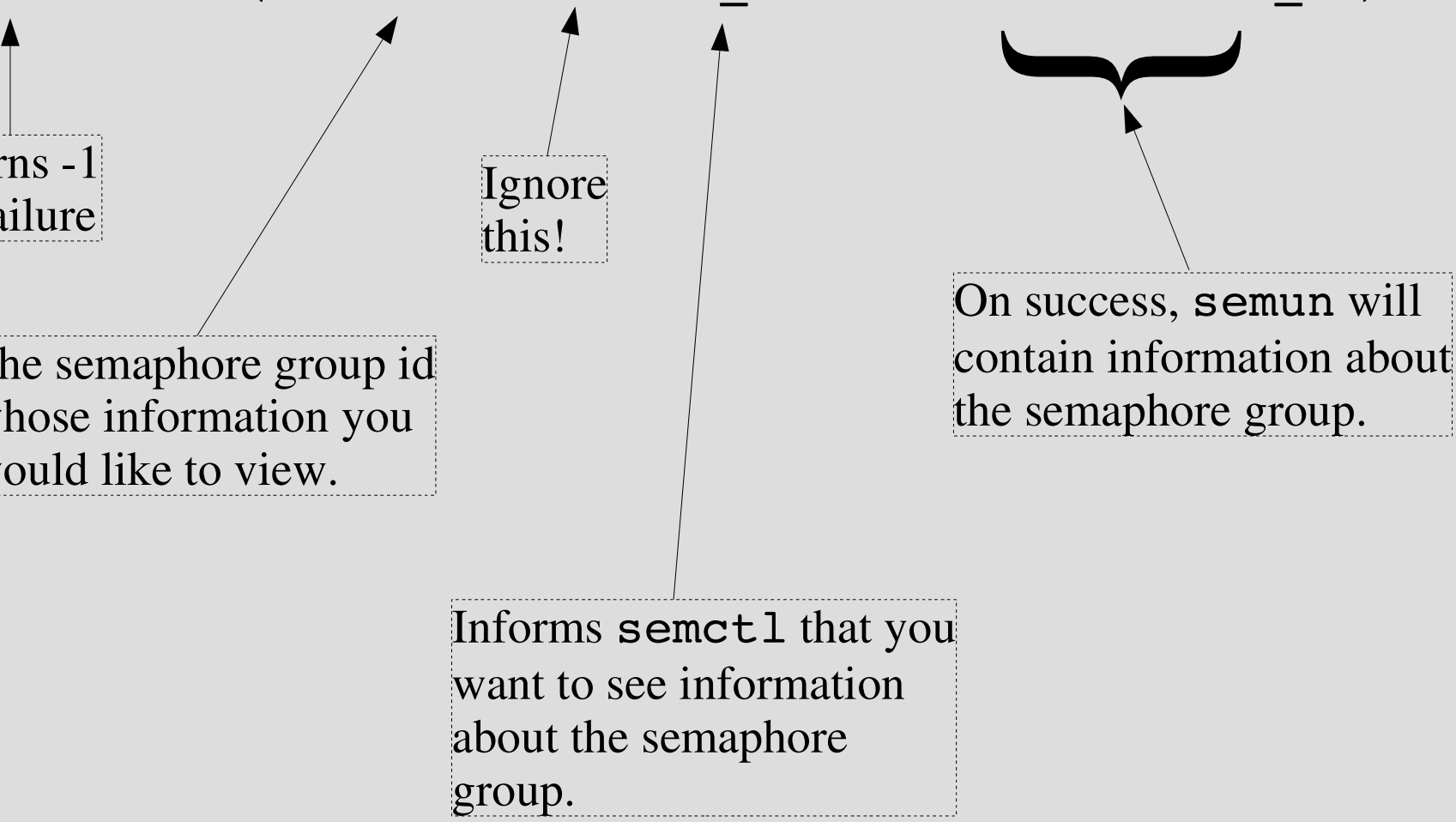
Initializing Semaphore Counters

```
union     semun SemUnion;
unsigned int    counters[] = {3, 4, 2, 1};

SemUnion.array = counters;

int semctl(int semid, 0, SETALL, SemUnion)
```



Obtaining Semaphore Group Information

```
int semctl(int semid, 0, IPC_STAT, semun AS semid_ds)
```



Finding a Group ID Key

```
struct semid_ds
{
  struct ipc_perm sem_perm;

  time_t          sem_otime;

  time_t          sem_ctime;

  unsigned short  sem_nsems;
};
```

```
struct ipc_perm
{
  key_t key;

  uid_t uid;

  gid_t gid;

  uid_t cuid;

  gid_t cgid;

  unsigned short mode;

  unsigned short seq;
};
```

Using the `semun` structure as a `semid_ds`, we can discover the key that was used to generate the semaphore group. This could be useful if you forgot to store the return value from `semget()`.

The Problem with exec()

If we write a simple multi-process program that only fork()s children, then all of the children will know the semaphore group id because they all have access to the same variables:

```
if(fork() == 0)
{
  ...
  printf("%i", semGID);
  ...
}
else
{
  ...
  printf("%i", semGID);
  ...
}
```

However, if one of the children calls exec(), then its process information will be overwritten with another image, so it will lose access to its parents variables. How can this child process gain access to the semaphore group information?

```
if(vfork() == 0)
{
  execlp("bob", "bob", NULL);
}
else
{
  ...
  printf("%i", semGID);
  ...
}
```

Solution: Passing Group Keys through Environment Variables

Environment variables are always passed on to child processes (well, almost always, but we're not going to go into that). Therefore, why not pass the semaphore group key as an environment variable?

```
char  S_Env[32];
key_t S_Grp;
```

Parent:

```
S_Grp = Error(semget(IPC_PRIVATE, 3, IPC_CREAT | 0666));

snprintf(S_Env, 31, "SEM_GROUP=%i", S_Grp);
Error(putenv(S_Env));
```

```
key_t S_Grp;
```

Child:

```
S_Grp = atoi(getenv("SEM_GROUP"));

Error(semget(S_Grp, 3, IPC_CREAT | 0666));
```

Performing Semaphore Operations

```
int semop(int semid, struct sembuf *sops, size_t nsops)
```

-1 on Failure

The semaphore group key that all of the semaphores in `sops` belong to

Array

The length of the `sops` array.

```
struct sembuf
{
  unsigned short semnum;   // Semaphore index
                           // in group.

  short          semop;    // semop < 0 --> Wait
                           // semop > 0 --> Signal

  short          sem_flg; // Set this to 0
}
```

semop Example:

```
// Create an enumerated type to help me remember
// which semaphore is which.
enum eSEMAPHORES {sTRAIN = 0, sCAR, sLIGHT};


// Create a sembuf structure that can be used to
// wait on the train semaphore.
struct sembuf Wait_On_Train = {sTRAIN, -1, 0};


// This will cause the process to wait on the
// sTRAIN semaphore (ie, semaphore 0) within
// the semaphore group S_Grp
Error(semop(S_Grp, &Wait_On_Train, 1));
```