Monday, September 23

Project 1 questions and overview

Project 1 testing

Functions defined in linked list macros

Project Requirements: Background

Spawn in Background

- Much of this is already in place in kthread.h
- But you have to change code to pass the "detached" argument.
- You'll also have to change other system calls as described in the spec.
- Don't change arguments to Spawn_Program and Spawn_With_Path.

Project Requirements: Kill

Kill Processes

- How do you terminate a process? Exit() provides a model, but the spec describes more issues.
- Use the function Detach_Thread in kthread.c. Set up the thread so the reaper will clean it up.
- When a process dies, who might have pointers to its kthread? What queues? (Next slide.)
- What is different when a process calls Kill on itself?
- There's not a lot of code for this part, but it may be where you spend the most time.

Project Requirements: Kill (cont.)

There is a list of all threads (s_allThreadList) and several thread queues.

A partial list of thread queues:

- s_runQueue
- each kthread's joinQueue
- wait queues for keyboard and block device requests
- s_graveyardQueue (threads that have already died) and s_reaperWaitQueue (reaper only)

You only have to handle cases where the thread being killed is on the run queue or a join queue.

Project Requirements: Process List

Process List

- How do you get a list of all processes?
- How do you determine process status? (And what is the status of the process that called Sys_PS?)
- Feel free to add fields to struct User_Context.

Linked List Macros

Defined in include/geekos/list.h.

- `DEFINE_LIST(listName, ofStruct)`
  creates struct listName; has pointers to the list head and tail
- `DEFINE_LINK(listName, ofStruct)`
  creates forward and backwards links within `struct ofStruct`
- `IMPLEMENT_LIST(listName, ofStruct)`
  defines a ton of list functions; names of the form `Function_Name_`*`listName`*

Linked List Functions

For a list My_List of struct My_Thing:

The list is declared like

```
struct My_List myList;
```

We'll use someThing to denote a **pointer** to a node of the list:

```
struct My_Thing *someThing;
```

Then you can traverse the list with

```
Get_Front_Of_My_List(&myList);
Get_Next_In_My_List(someThing);
```

Get_Next_In… returns NULL at the end of the list.

Linked List Notes

Some points that may be tricky:

- The macros define types and functions. You still have to declare the variable for the list itself:

```
DEFINE_LIST(My_List, My_Thing);
struct My_List myList;
```

- Each list type has its own functions. That is, you don't have a generic "get front of list" function; you have one for each type of list.
- A node can only be on one list of a given type, because there's only one set of forward and backward pointers for each type. In many cases, there is only one list of a type—for instance, All_Thread_List.

More Linked List Functions

You'll need to read the code in list.h for a complete list of functions. Here, myList and myList2 are My_List structs, and someThing and afterWhat are **pointers** to list nodes.

```
Clear_My_List(&myList);
Is_Member_Of_My_List(&myList, someThing);
Add_To_Front_Of_My_List(&myList, someThing);
Append_My_List(&myList, &myList2);
Remove_From_Front_Of_My_List(&myList);
Remove_From_My_List(&myList, someThing);
Insert_Into_My_List(&myList, afterWhat, someThing);
Is_My_List_Empty(&myList);
```

For each function that goes forward (front of list, go to next, etc.) there is a corresponding one that goes backward (back of list, go to previous, etc.).

Probably avoid Set_Next _In_My_List and Set_Prev_In_My_List.