1 What is a context?

A processer context is a clean way of representing the state of the machine at any time. Contexts are the underlying layer upon which threads are built. Every context, represented by a ucontext structure, has the following components:

- its own stack
- its own set of signal flags
- all of the registers

Essentially, it allows a user to isolate some portion of code execution within a context and switch between them. Contexts can be created, suspended, copied and destroyed. They are similar to threads but come with no pre-set switching mechanism - i.e. if you create two threads by default they will both run but two contexts don’t actually do anything unless you make them.

2 Details

First, a ucontext structure must be initialized with a call to getcontext(2). The structure will have the following fields (and more not shown):

```c
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t uc_sigmask;
    stack_t uc_stack;
    ...
} ucontext_t;
```

You still need to set the structure values above:

- `uc_link` is a the next context to run when this context completes;
- `uc_sigmask` is a signal mask for blocking signals in this context;
- `uc_stack` is the execution stack used by this context. For a description of the `uc_stack` structure reference the manual for sigaltstack(2). Beware not to make the stack too small, or it will be overrun and very bad things will happen.

Setting these values is still insufficient to executed the context, and you still need to set up the function that will be called when the context is set or swapped. This done by a call to makecontext(2), and it is well described in the manual. A context is switched in using either setcontext(2) or swapcontext(2), which either directly sets the context, or sets and also saves the state of the running context, respectively.
3 Useful Man Pages

- ucontext
- getcontext
- makecontext
- setcontext
- swapcontext

4 Sample program and Exercises

Found in `~dxu/handouts/cs355/` is a context-helloworld program `context-hw.c` that is a good place to start. Try editing the program and adding new features. Here are some mini-exercises you might want to try:

1. What happens if you want to switch back to the main program after printing Hello World?
2. Can you write a program that alternates between two functions indefinitely?
3. What happens when a signal is delivered? How do signals affect the execution of a context?
4. Can you write a program that alternates between two functions at a set interval? Recall the kernel timer lab we did.
5. How do you track the current context? If a function is running in multiple contexts at once, how do you know which one is which?

4.1 ucontext Hello World

```c
#include <ucontext.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define STACKSIZE 4096

void f()
{
    printf("Hello World\n");
}

int main(int argc, char * argv[]){
    ucontext_t uc;
    void * stack;
    getcontext(&uc);
    stack = malloc(STACKSIZE);
    uc.uc_stack.ss_sp = stack;
    uc.uc_stack.ss_size = STACKSIZE;
    f();
    return 0;
}
```
uc.uc_stack.ss_flags = SS_DISABLE;
sigemptyset(&(uc.uc_sigmask));
uc.uc_link = NULL;
makecontext(&uc, f, 0);
setcontext(&uc);
perror("setcontext"); //setcontext() doesn’t return on success
return 0;

5 Spinning the CPU

You will find yourself creating work to keep threads running long enough, such as loops that look like this:

while(global_flag==0) {
    //do nothing
}

This works, but drives up the load on the underlying computer and makes things slower for everyone. Unfortunately, there is no way to give up CPU time nicely, as sleep or usleep because they trample SIGALRM. To prevent burning CPU cycles unnecessarily use poll(NULL, 0, 100); This is a hack to prevent spinning. It calls the poll(2) function which has a timeout (3rd argument). The timeout doesn’t work properly with the ucontext switching, but does allow other programs to use the CPU.

while(global_flag==0) {
    //do nothing nicely
    poll(NULL, 0, 100);
}