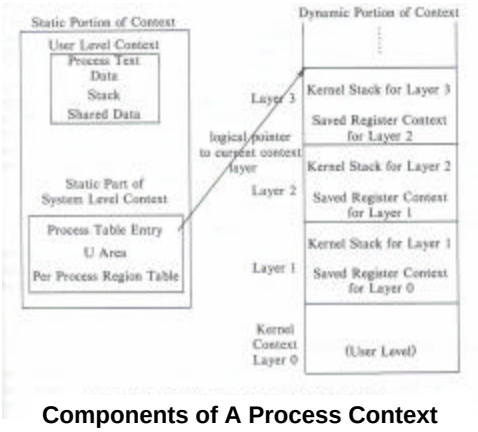
Process Context/Control

Last Time

- Context of A Process consists of 3 context layers. What are they?
  - User Level Context
    - Text, data, stack
  - System Level Context
    - U area, Proc table Entry, Pregion
    - Kernel Stack, set of layers
  - Machine Level Context
    - CPU registers (GP, PC, SP, Index regs, etc.)

Process Context

- Kernel Pushes and Pops the context when
  - Interrupts occur
  - System Call
  - Context switch
- A context is saved and another restored when?



Components of A Process Context

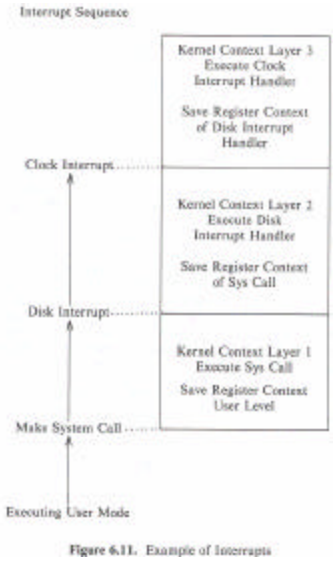


Figure 6.11. Example of Interrupts

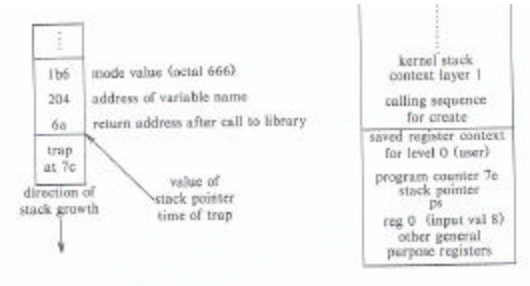
Example of Interrupts

```
Algorithm inthand
Input: none
Output: none
{
   save (push) current context layer;
   determine interrupt source;
   call interrupt handler;
   restore (pop) previous context layer;
}
```

```
algorithm syscall              /* algorithm for invocation of system call */
input:   system call number
output: result of system call
{
      find entry in system call table corresponding to system call number;
      determine number of parameters to system call;
      copy parameters from user address space to u area;
      save current context for abortive return (described in section 6.4.4);
      invoke system call code in kernel;
      if (error during execution of system call)
      {
            set register 0 in user saved register context to error number;
            turn on carry bit in PS register in user saved register context;
      }
      else
            set registers 0, 1 in user saved register context
                              to return values from system call;
}
```

Algorithm for System Calls

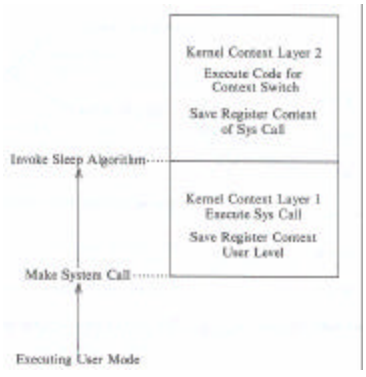
```
Char name[] = "file";
Main()
{
        int fd;
        fd = create(name, 0666);
}
```



Stack for Create Sys Call

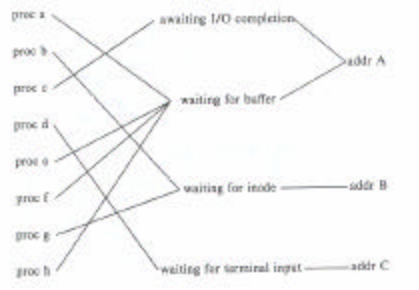
Sleeping Process

- When does a Process go to sleep?
  - System Call (awaiting an event)
  - Page Fault



Context Layer of Sleeping Proc

Sleeping Cont.



- Set of events map to a set of addresses
- Multiple processes can map to same event
- All process waiting on event are awaken

Process Control

- What is the mechanism to create new processes?
- Hint First process is hand crafted at initial boot of OS.
- Fork() syscall
  - pid = fork();

Fork

- Creates an “exact” image of parent process.
- Common
  - U area, text, data, inodes, fd, etc.
- Not Common
  - System level context
  - PID

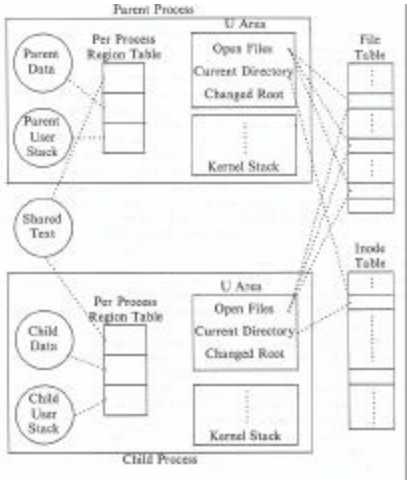
General Outline of Fork

- Reserve virtual memory
- Allocate proc table entry and fill it
- Copy Parent’s process group, credentials, FD, limits and signal actions to child
- Allocate new u area and copy data from parents
- Dup Parent’s Pregion
- Child returns 0, parent returns PID of Child

Fork Alg.

```
algorithm fork
input:  none
output: to parent process, child PID number
        to child process, 0
{
    check for available kernel resources;
    get free proc table slot, unique PID number;
    check that user not running too many processes;
    mark child state "being created;"
    copy data from parent proc table slot to new child slot;
    increment counts on current directory inode and changed root (if applicable);
    increment open file counts in file table;
    make copy of parent context (u area, text, data, stack) in memory;
    push dummy system level context layer onto child system level context;
            dummy context contains data allowing child process
            to recognize itself, and start running from here
            when scheduled;
    if (executing process is parent process)
    {
        change child state to "ready to run;"
        return(child ID);       /* from system to user */
    }
    else    /* executing process is the child process */
    {
        initialize u area timing fields;
        return(0);      /* to user */
    }
}
```

Process Control -- Fork



New Processes

- This time we want to create a process that is different then the parent. How do we accomplish this task?
- Exec() system call
- Typically used with a fork() system call

Tasks left to Exec

- Parse pathname of executable
- Verify execute permissions
- Read/check header of executable
- Adjust Set UID/GID bits
- Copy envir. Variables into Kernel space
- Allocate memory regions
- Release old space address

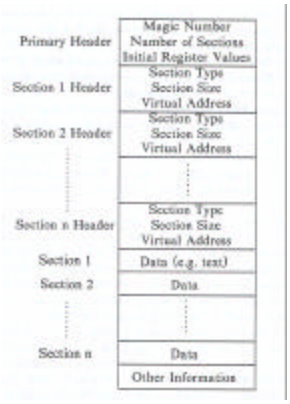
Tasks left to Exec

- Set up memory space (e.g. Shared text or not)
- Copy environment variable back to user space
- Reset signal handlers
- Initialize hardware context

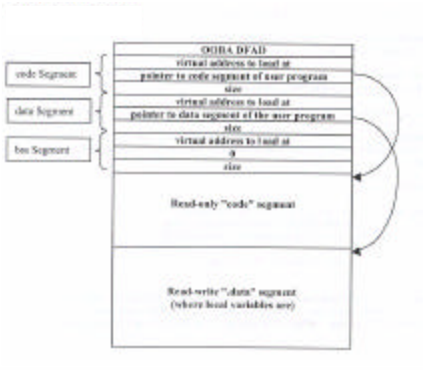
```
algorithm exec
input:   (1) file name
         (2) parameter list
         (3) environment variables list
output: none
{
        get file inode (algorithm namei);
        verify file executable, user has permission to execute;
        read file headers, check that it is a load module;
        copy exec parameters from old address space to system space;
        for (every region attached to process)
                detach all old regions (algorithm detach);
        for (every region specified in load module)
        {
                allocate new regions (algorithm allocreg);
                attach the regions (algorithm attachreg);
                load region into memory if appropriate (algorithm loadreg);
        }
        copy exec parameters into new user stack region;
        special processing for setuid programs, tracing;
        initialize user register save area for return to user mode;
        release inode of file (algorithm iput);
}
```

Exec Algorithm

Executable File Formats



General Unix Format



NACHOS Format