Overview of s_osk

- **s_osk** simulates a simple OS running on a single CPU
- Simulated system calls (called by user processes)
  - `s_begin`, `s_end` User task startup/termination
  - `s_spawn`, `s_wait` Task creation/wait
  - `s_yield` Voluntary context switch
  - `s_getusage` Get cpu usage (virtual time) of all user processes
- Unix mechanisms used in implementation
  - Process creation, termination, synchronization
  - Inter-process communication (FIFOs)
  - Signals and signal handlers

Conceptual Model

- A **task** is a simulated process
  - One system task (`s_osk`)
  - Zero or more user tasks
- System task (`s_osk`)
  - Controls creation of all user tasks
  - Tracks task CPU usage
- The system is **non-preemptive**
  - A task retains control until it makes a system call
  - One Exception: A **timeout alarm** prevents runaway task
  - `s_osk` terminates when the timeout alarm has expired or when all user tasks have terminated
- **Invariant**: There is at most one active task at a time (even though there may be many ready tasks)

Example 1 (1)

- **user1.c User Process**
  - Initialize by calling `s_begin()`
  - Terminate by calling `s_end()`
  - `s_begin()` and `s_end()` are defined in `s_runlib.c`
    - Compile and link with `user1.o`

- **s_osk Simulated OS**
  - Start execution: `s_osk -v 5 user1`
    - Set verbose level to 5
    - Fork/exec initial user task 'user1'
    - Initial task may create other tasks through `s_spawn()`
  - After fork/exec, wait for a user task (user1) to return control to `s_osk`
Example 1 (2)

- Control Structure
  » Control passes back and forth between s_osk and user1 processes through messages

```
user1
s_begin()
{ write msg;  
  read msg;  
  ...  
  Compute ...  
  s_end()
  { write msg;  

s_osk
  Initialize:  
  Update procTbl;  
  fork/exec user1;  
  read msg:  
  Update procTbl;  
  write msg;  
  read msg:  
  Update procTbl;  
  ...  
```

Initialization Example

```
user1:  
pid = 385  
tid = 1  

syscallFifo

```

Created by s_osk

```
s_osk
  "syscallFifo"

```

Created by user1

```
s_begin:

1) Open "syscallFifo" WO
2) Get pid
3) Create replyFifo
4) Send msg with pid
5) Open replyFifo RO
6) Recv msg with tid
7) Read syscall from "syscallFifo"

```

Deathlock Startup

```

s_osk
Create "syscallFifo";
Open "syscallFifo" RO;
Fork/exec user1;

Deadlock

s_begin()
{ write msg;
  NNNNN = Get pid;
  Create replyFifoNNNNNN;  
  Open "replyFifoNNNNNN" RO;
  Open "syscallFifo" WO;
  Write STARTUP msg to "syscallFifo";
  } 

User Task

```

```

```

SysCall Message Formats

- Basic Message (Suggested)
  » typedef struct timeval timeval_t
  » vtime
    » Is accumulated user + system time
    » Obtained by calling getrusage(2)
      » (u->ru_utime + u->ru_stime) where u is a ptr to rusage

```

```
struct requestBase {
  u_int16_t bodySz;  // #bytes in msg body
  tid_t tid;  // task id, pid if from s_begin
  u_int16_t syscall;  // system call number
  timeval_t vtime;  // accumulated virtual time
};
```
**SysCall Reply Message Formats**

```c
struct replyBase {
    u_int16_t bodySz; // #bytes in msg body
tid_t tid; // original task id
u_int16_t sysCall; // system call number
int status; // return status
};
```

- **s_begin**: Task id
- **s_end** and **s_spawn**: Return code
- **s_wait**: Return status of terminated task
- **s_yield**: None
- **get_rusage**: Need an extended message (i.e., bodySz > 0)

**Example 2**

**Control Structure**

```
user2
s_begin()
{ write msg;
read msg;
}
... Compute ...
s_yield()
{ write msg;
read msg;
}
s_end()
{ write msg;

s_osk
Initialize:
... Update procTbl;
... fork/exec user1;
... read msg. Update procTbl;
... write msg to user2;
... read msg. Update procTbl;
... user1 is at head of run queue
... write msg to user2;
... read msg;
```

**Example 3**

```
s_osk
Initialize:
... Update procTbl;
... fork/exec user1;
... read msg. Update procTbl;
... write msg to user3;
... read msg. Update procTbl;
... fork/exec new task T;
... read s_begin msg from T;
... write msg to T;
... read s_yield msg from T;
... write msg to user2;
... read msg;
... Task T at head of run queue etc.
```

```
s_begin()
{ write msg;
read msg;
}
tid = s_spawn(...)
{ write msg;
read msg;
}
... Compute ...
s_yield()
{ write msg;
read msg;
}
s_end()
{ write msg;

user3
```

**Example 5**

```
s_osk -v 5 user5 50000 60000
```

```
s_begin()

user5.c
```

```
int calcPrime(int n) { ... Some code ... }
int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    s_begin();
    calcPrime(n);
    s_end();
}
```

```
user5a.c
```

```
int calcPrime(int n) { ... Some code ... }
int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    s_begin();
    calcPrime(n);
    s_end();
}  
```