CSCV-452

CSCV 452 Principles of Operating Systems

Michael Duren

Week 10 – Phase III

Agenda

- Review Phase I concepts
- Lists
- Phase 3

Phase 3: Getting Started

- Download ***provided_phase3.zip***, which includes starter files and test cases for Phase 3.
- Create a working directory ***phase3*** in parallel with your *phases 1 and 2* folders, extract and copy the phase3 starter files and test cases
- After you create the directory ***phase3***, **cd** to phase3. In this phase3 directory, create a link to USLOSS:

  ln -s ../usloss/build usloss
- USLOSS is a library: *libusloss.a*. It is located in usloss/lib.

Phase 3: Getting Started

- Your phase3 code will be compiled into a library named: **libphase3.a** in your directory
- To execute a test case, you link the .o file of the test cases with the **libuser.o** and phases 1, 2, and 3 libraries.
  - The *libuser.o* is the object file resulting from *libuser.c*
  - Typing **make** will create the libphase3.a library in your directory.
  - Typing **make test00**, for example, will create an executable test case named 'test00' in your directory.
  - Typing make –B test00 will rebuild your libphase3.a library and the test program regardless of whether or not there were any changes

Phase 3: Phase 1 and 2 libraries

- In the provided Makefile, use the line:

  LIBS = -l452phase2 –l452phase1 -lusloss –l452phase1 –l452phase2 -lphase3

- Your phase3 will be graded using the provided lib452phase1.a and lib452phase2.a libraries
- Make sure you have the two libraries stored in the usloss/lib folder (my recommendation) or your working directory
- You can use your own libraries for development

Phase 3: Header Files for Phase 2

- usloss/include/phase[1-3].h function prototypes and constants to be used in this phase.
- usloss/include/usyscall.h syscall opcodes to be used as syscall numbers in phases 3 and 4
- usloss/include/libuser.h function prototypes for the system calls in this phase
- usloss/include/usloss.h contains function prototypes for USLOSS library functions, many useful constants.
- Your phase 3 data structures and constants will go to local .h file(s) that you will provide, or they may go in your .c file.

Phase 3: libuser.h (partial) in usloss/include

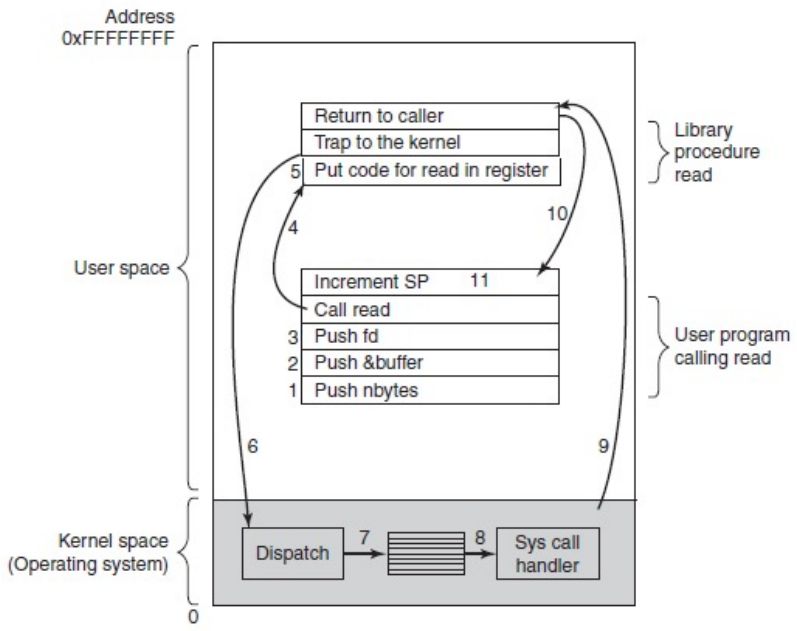
```
/* Phase 3 -- User Function Prototypes */

extern int  Spawn(char *name, int (*func)(char *), char *arg,
                  int stack_size, int priority, int *pid);

extern int  Wait(int *pid, int *status);

extern void Terminate(int status);

extern void GetTimeofDay(int *tod);

extern void CPUTime(int *cpu);

extern void GetPID(int *pid);

extern int  SemCreate(int value, int *semaphore);

extern int  SemP(int semaphore);

extern int  SemV(int semaphore);

extern int  SemFree(int semaphore);
```

Phase 3: provided_prototypes.h

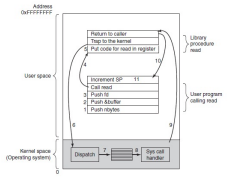
```
extern int  spawn_real(char *name, int (*func)(char *), char *arg,
                       int stack_size, int priority);

extern int  wait_real(int *status);

extern void terminate_real(int exit_code);

extern int  semcreate_real(int init_value);

extern int  semp_real(int semaphore);

extern int  semv_real(int semaphore);

extern int  semfree_real(int semaphore);

extern int  gettimeofday_real(int *time);

extern int  cputime_real(int *time);

extern int  getPID_real(int *pid);
```

Phase 3: Architecture

Recall:



Phase 3: Architecture



User Space

```
int Start3()
{
   int pid;
   GetPID(&pid);
}
```

Kernel Space

```
getpid();
```

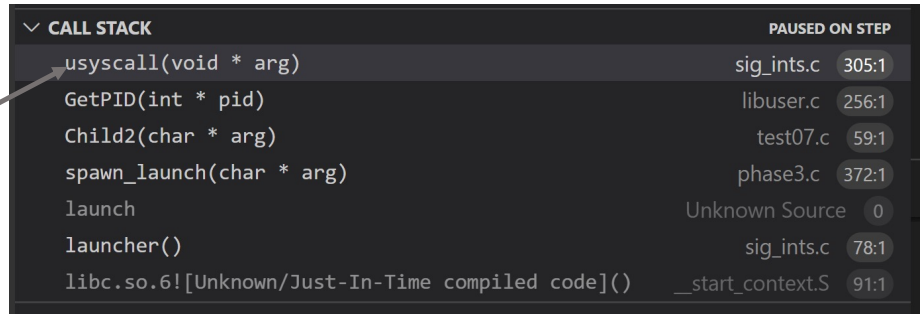
Phase 3: Architecture

User Space

```
int Start3()
{
   int pid;
   GetPID(&pid);
}

void GetPID(int *pid)
{
    sysargs sa;

    CHECKMODE;
    sa.number = SYS_GETPID;
    usyscall(&sa);
    *pid = (int) sa.arg1;
    return;
}
```

Kernel Space

```
getpid();
```



User Space process signals the kernel to perform some function.

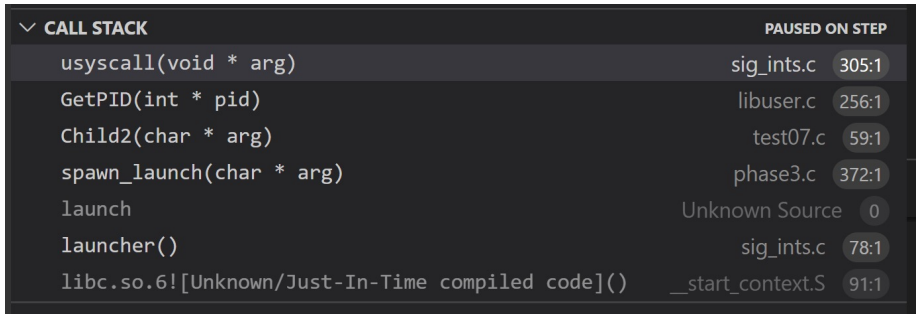
Phase 3: Architecture

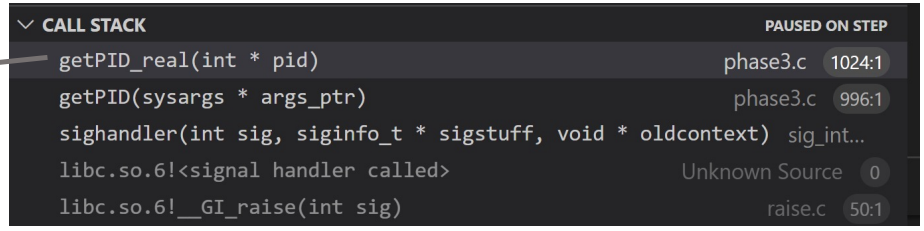
User Space

```
int Start3()
{
   int pid;
   GetPID(&pid);
}

void GetPID(int *pid)
{
    sysargs sa;

    CHECKMODE;
    sa.number = SYS_GETPID;
    usyscall(&sa);
    *pid = (int) sa.arg1;
    return;
}
```

Kernel Space

```
getpid();
```





Kernel level function is performed

Phase 3: Debugging Notes

Review section 5 of USLOSS manual

SIGUSR1 signal used in implementation of USLOSS interrupts

To get around the problem, add the below line in your .gdbinit file either in the current working directory or in your home directory

```
handle SIGUSR1 nostop noprint
```

Phase 3: Debugging Notes on lldb

The process handle command can be used to allow SIGUSR1 be passed to your process without lldb getting in the way to stop it.

Assuming you have test00 created,

```
lldb test00
```

And then run the process and see what you get.

What should you do then?

```
b main
```

```
pro hand –p true –s false SIGUSR1
```

Then you do your debugging activities

Phase 3: start2 Function

Recall: the phase 2 library uses **fork1** to create a kernel-level process at **priority 1** that will execute the **start2** code that you provide in phase 3.

Thus, **start2** is the entry point for phase 3. When the code in **start2** starts executing, there will be three processes already created: **sentinel** and **start1** and **start2**.

Initialize your phase 3 data structures; in particular, the **phase 3 process table** and the **semaphore table**.

Phase 3: Process Table for Phase 3

You will need a process table for phase 3. This table will be used to manage **User-Level Processes**! You **cannot** modify/extend the phase 1 or 2 process tables or data structures.

You can use **MAXPROC** for the size of the phase 3 process table.

Note: when using the provided lib452phase1.a, you can use **getpid()%MAXPROC** to determine which slot in your phase 3 process table to use. (In phase 2, you were recommended same approach to locate an entry in your phase 2 process table.)

Phase 3: start2 Function cont’

Initialize the **sys_vec** array to point to the system call functions that you are creating in phase 3.

Make the unused parts of **sys_vec** point to a **nullsys3** function.

This function is similar to the **nullsys** function used in phase 2 with the difference being that **nullsys3** terminates the offending process, rather than halting the simulator.

Phase 3: Using usyscall.h

You can check **usyscall.h** that is stored in the same folder with usloss.h and phase1.h and other USLOSS system header files

Partial code in the header file

```
#define SYS_SPAWN                    3

#define SYS_WAIT                     4

#define SYS_TERMINATE                5
```

Use these to assign the correct handler in the sys_vec table, such as:

```
sys_vec[SYS_SPAWN]          = spawn;

sys_vec[SYS_WAIT]           = wait1;
```

Phase 3: Test Processes

Test processes in phase 3 run in **user-mode**, not kernel mode

System calls that create and manage user mode processes: **Spawn, Wait,** and **Terminate**.

Errors cause termination of the offending process, rather than halting USLOSS.

Terminate definition leads to call **quit** after all the descendants of the process have quit.

Phase 3: start2 Function cont’

Spawn the phase3 test process: **start3**.

Note that **start3** is a **user-mode** process.

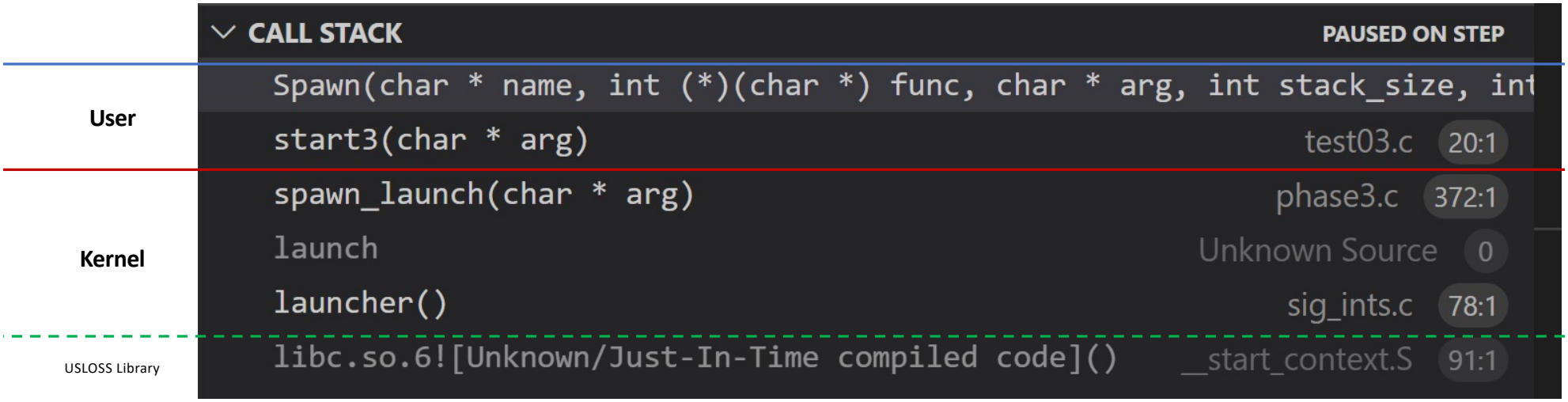
It must have:

Priority 3

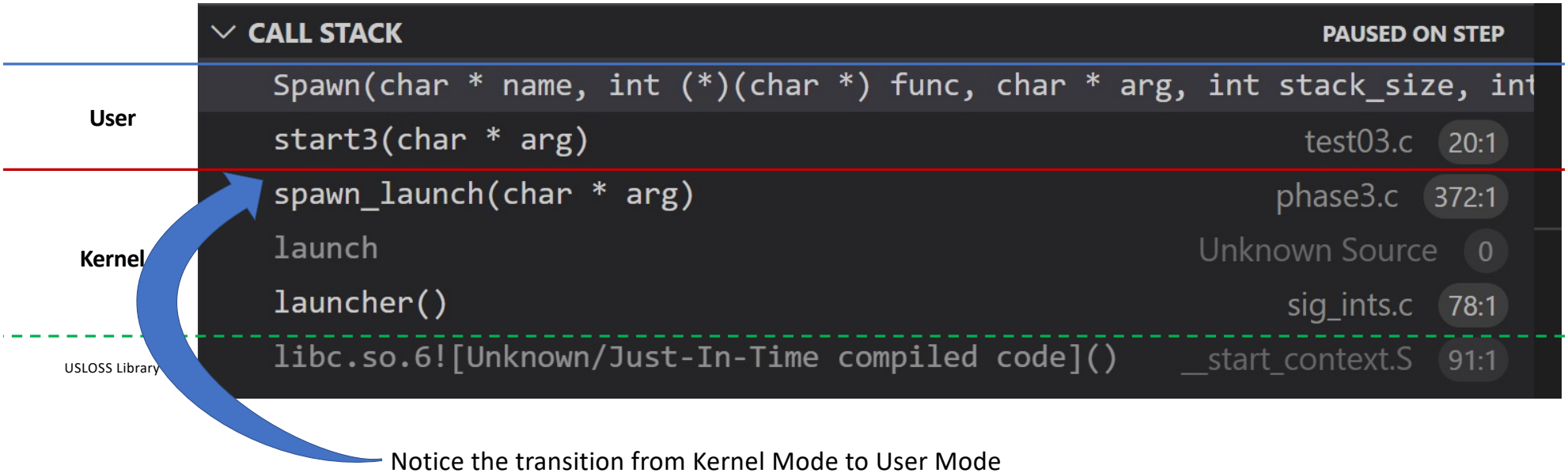
Stack size = 4 * USLOSS_MIN_STACK.

start2 should then wait for **start3** to terminate, after which **start2** calls **quit(0)**.

Phase 3: Architecture



Phase 3: Architecture



Notice the transition from Kernel Mode to User Mode

Phase 3: System Call Handling

- Protect OS by not allowing user process to execute system code
- Review: Each system call is handled as a software interrupt
  - On USLOSS, this is an interrupt of type SYSCALL_INT
  - usyscall changes process from user to kernel mode.
  - This is the only way to do this! (Cannot use psr_set to change from user to kernel mode.)
  - int_vec[SYSCALL_INT] = syscall_handler;
- When a system call happens, the syscall_handler is invoked. It will call a particular function that can handle the system call based on the syscall number.

Phase 3: Skeleton Code for sys_handler (phase 2)

```
void syscall_handler(int dev, void *unit)
{
      sysargs *sys_ptr;
      sys_ptr = (sysargs *)unit;

      /* Sanity check: if the interrupt is not SYSCALL_INT, halt(1)*/


      //more checking


      /* Now it is time to call the appropriate system call handler*/
      sys_vec[sys_ptr->number](sys_ptr);

}
```

Phase 3: sysarg

```
sysarg data structure is defined in phase2.h

/* The sysargs structure */

typedef struct sysargs
{
      int number;
      void *arg1;
      void *arg2;
      void *arg3;
      void *arg4;
      void *arg5;
} sysargs;
```

Phase 3: System Call Handling cont'

- You can no longer disable (or enable) interrupts. All the code in phase 3 runs with interrupts enabled.
- This allows greater concurrency and time-slicing is now possible within the implementation code.
- Must change back to **user-mode** before exiting system-call handling code. Note that you can use **psr_set** to do this, since **psr_set** is callable when in kernel mode.

Phase 3: System Call Handler

- The array **sys_vec[]** is declared in phase2.h

```
extern void (*sys vec[])(sysargs *args);
```

- The macro **MAXSYSCALLS** is defined in phase1.h

```
#define MAXSYSCALLS 50
```

- In your phase2.c, you should have defined the variable such as

```
/* system call array of function pointers */
void (*sys_vec[MAXSYSCALLS])(sysargs *args);
```

Phase 3: System Call Handling cont’

- In your start2 function definition in phase3.c, you need to initialize the system call handlers

```
for (i = 0; i < MAXSYSCALLS; i++)
//initialize every system call handler as nullsys3;
```

- A reference definition of nullsys3

```
static void nullsys3(sysargs *args_ptr)
{
   printf("nullsys3(): Invalid syscall %d\n", args_ptr->number);
   printf("nullsys3(): process %d terminating\n", getpid());
   terminate_real(1);
} /* nullsys3 */
```

- You need to fill in the entries pointing to the valid system callers you implement in phase 3 such as

```
sys_vec[SYS_SPAWN]        = spawn;
```

Phase 3: Mutual Exclusion

- You cannot disable interrupts as you did in phases 1 and 2
- Use mailboxes
- one-slot mailbox
- Recall mailboxes used in problem examples such as producer-consumer or reader-writer problems.

Phase 3: How To Bock

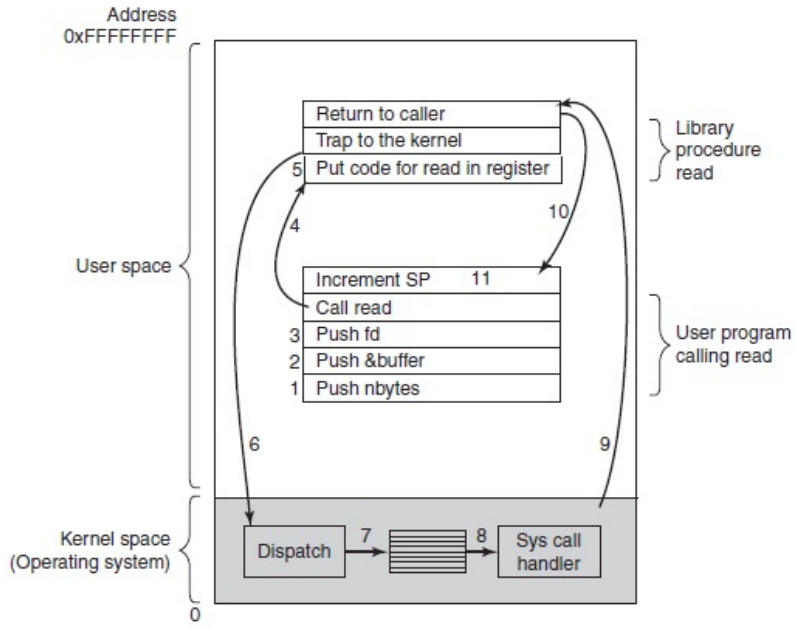
- Cannot use block_me and unblock_process from phase 1
- Use a private mailbox
- Each process has a private mailbox
- This is a zero-slot mailbox
- How to block a running process?
- Only the process ever does is a MboxReceive on its private mailbox
- Some other process will do the corresponding MboxSend or MboxCondSend when it is time to wake up the process

Phase 3: Spawn in test00.c (partial code)

```
int start3(char *arg)
{
    int pid;
    int status;


   printf("start3(): started.  Calling Spawn for Child1\n");
   Spawn("Child1", Child1, NULL, USLOSS_MIN_STACK, 5, &pid);
   printf("start3(): fork %d\n", pid);
   Wait(&pid, &status);
   printf("start3(): result of wait, pid = %d, status = %d\n", pid, status);
   printf("start3(): Parent done. Calling Terminate.\n");
   Terminate(8);


   return 0;
} /* start3 */
```

Phase 3: Spawn in libuser.c

```
int Spawn(char *name, int (*func)(char *), char *arg, int stack_size,
        int priority, int *pid)
{
    sysargs sa;

    CHECKMODE;
    sa.number = SYS_SPAWN;
    sa.arg1 = (void *) func;
    sa.arg2 = arg;
    sa.arg3 = (void *) stack_size;
    sa.arg4 = (void *) priority;
    sa.arg5 = name;
    usyscall(&sa);
    *pid = (int) sa.arg1;
    return (int) sa.arg4;
} /* end of Spawn */
```



Phase 3: In your phase3.c

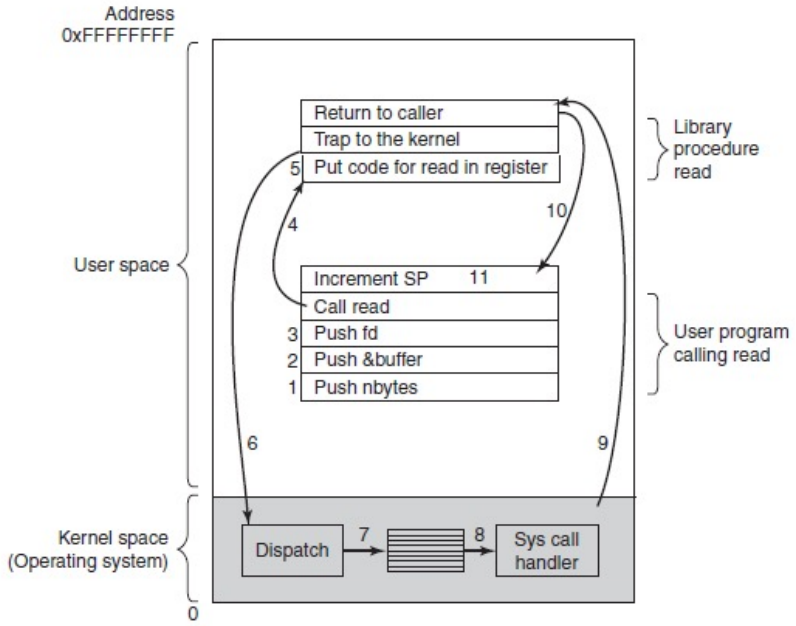
The definition of start2 definition has the below code:

```
for (i = 0; i < MAXSYSCALLS; i++)
   sys_vec[i] = nullsys3;

sys_vec[SYS_SPAWN]        = spawn;
```

Phase 3: In your phase3.c

The definition of st … below code:

```
for (i = 0; i < M

    sys_vec[i] = r
```



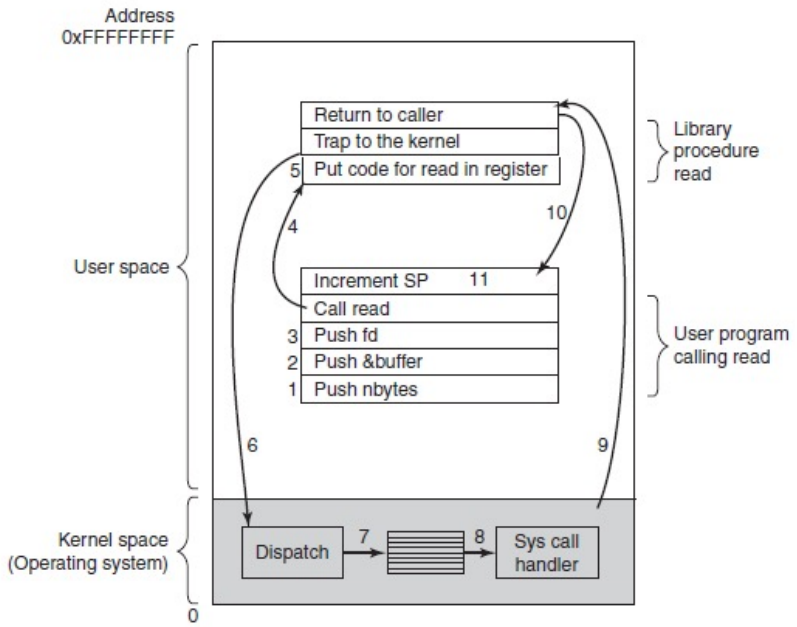

```
sys_vec[SYS_SPAWN]          = spawn;
```

Phase 3: Revisit phase2.c interrupt handling

```
// this is the function defined in liblxuphase2.a
static void syscall_handler(int dev, void * punit).

{

   sysargs *sys_ptr;

   sys_ptr = (sysargs *) punit;

   //code deleted

   sys_vec[sys_ptr->number](sys_ptr);


} /* syscall_handler */
```

Phase 3: spawn

```
static void spawn(sysargs *args_ptr){
   int (*func)(char *);
   char *arg;
   int stack_size;
  //more local variables
   if (is_zapped() ) { //should terminate the process }

   func = args_ptr->arg1;
   arg = args_ptr->arg2;
   stack_size = (int) args_ptr->arg3;
   //more code to extract system call arguments as well as exceptional handling

   // call another function to modularize the code better
   kid_pid = spawn_real(name, func, arg, stack_size, priority);
    args_ptr->arg1 = (void *) kid_pid;   //packing to return back to the caller
    args_ptr->arg4 = (void *) 0;

   if (is_zapped() ) { //should terminate the process }
   /* set to user mode */  // more code to write. Hint: call psr_set to do this
    return ;
}
```



Phase 3: spawn_real

- Call fork1 to create a process that runs a start function
- The process runs at user mode
- Maintain the parent-child relationship at phase 3 process table
- Hint: provide a launch function: spawn_launch()

Phase 3: spawn_real

```
int spawn_real(char *name, int (*func)(char *), char *arg,
               int stack_size, int priority){
   int kidpid;
   int my_location;  /* parent's location in process table */
   int kid_location; /* child's  location in process table */
   int result;
   u_proc_ptr kidptr, prevptr;

   my_location = getpid() % MAXPROC;

   /* create our child */
   kidpid = fork1(name, spawn_launch, NULL, stack_size, priority);
   //more to check the kidpid and put the new process data to the process table
   //Then synchronize with the child using a mailbox
      result = MboxSend(ProcTable[kid_location].start_mbox,
                        &my_location, sizeof(int));
  //more to add
   return kidpid;
}
```

Phase 3: spawn_launch

```
static int spawn_launch(char *arg)
{
   int parent_location = 0;
   int my_location;
   int result;
   int (* start_func) (char *);
  // more to add if you see necessary

   my_location = getpid() % MAXPROC;

   /* Sanity Check */
  /* Maintain the process table entry, you can add more */
   ProcTable[my_location].status = ITEM_IN_USE;

  //You should synchronize with the parent here,
  //which function to call?

  //Then get the start function and its argument
```

Phase 3: spawn_launch

```
//…..following previous slide, add more code
  if ( !is_zapped() ) {
     //more code if you see necessary
    //Then set up use mode
     psr_set(psr_get() & ~PSR_CURRENT_MODE);
     result = (start_func)(start_arg);
     Terminate(result);
  }
  else {
     terminate_real(0);
  }
  printf("spawn_launch(): should not see this message following Terminate!\n");

  return 0;
} /* spawn_launch */
```

Phase 3: Terminate

- Terminate is expected to terminate the invoking process, which is a user-level process, and all of its children
- To terminate the children processes, your system-call handling function, assuming that you name it terminate, should call zap() to "zap" each child.
- Recall that zap was implemented in phase 1.
- The invoking process blocks when zapping another process and is unblocked when the zapped process quits.

Phase 3: Terminate

- Your **user-level** process terminates via system call handling that calls **quit()**.
- Should do this at the end of **terminate_real** definition.
- You need to clean your phase 3 table entries when handling terminating processes
- You also need to adjust parent's children list on the process table

Phase 3: semcreate

- Is the function pointed by **sys_vec[SYS_SEMCREATE]**?
- Extract the initial semaphore value from the sysargs structure and makes sure it is not negative.
  - Calls semcreate_real

    ```
    int semacreate_real(int init_value)
    ```

  - Which then handles the work of semaphore creation
- **Semcreate_real** returns the result of the creation to **semcreate**;
- **semcreate** then puts this result back into the **sysargs** structure, changes to user mode, then returns.

Phase 3: Flow of Execution

User process (a phase 3 test case test20.c):

```
int sem1;
int start3(char *arg)
{
    int result;
    int pid;
    int status;

    printf("start3(): started\n");
    result = SemCreate(3, &sem1);
    SemP(sem1);
    //...
    Terminate(8);

    return 0;
} /* start3 */
```

Phase 3: SemCreate

User process (a phase 3 test case)

result = SemCreate(3, &sem1)

```
int SemCreate(int value, int *semaphore)
{
    sysargs sa;

    CHECKMODE;
    sa.number = SYS_SEMCREATE;
    sa.arg1 = (void *) value;
    usyscall(&sa);
    *semaphore = (int) sa.arg1;
    return (int) sa.arg4;
} /* end of SemCreate */
```

Phase 2 code

```
int_vec[SYSCALL_INT]   = syscall_handler;
```

```
static void syscall_handler(int dev, void * punit)
{
  sysargs *sys_ptr;

  sys_ptr = (sysargs *) punit;

  //code deleted

  sys_vec[sys_ptr->number](sys_ptr);

} /* syscall_handler */
```

Phase 3: SemV

- What if the semaphore value >0
- What if the semaphore value ==0
  - Is there any process blocked on the semaphore because of P operation?
  - **MboxCondSend** can be used to check the semaphore's private mailbox used for blocking
  - No process is blocked on it

Phase 3: SemP

- What if the semaphore value >0
- Otherwise
  - MboxReceive used to block on the private mailbox of the semaphore
  - After unblocked
  - if the semaphore is being freed, need to synchronize with the process that is freeing the semaphore
  - Hint: use another zero-slot mailbox

Phase 3: SemFree

- TWO conditions to consider
  - Where there is no process blocked on the semaphore
  - When there are processes blocked on the semaphore

Phase 3: Processes blocked on the semaphore

```
//The below code is needed when someone is blocked on the semaphore
  SemTable[semaphore].status = FREEING;
  strcpy(free_buf, "FREE");
  process_count = 0;
  while (MboxCondSend(SemTable[semaphore].priv_mbox, free_buf,
                  strlen(free_buf) + 1) == 0) {
    process_count++;
  }
  /* Each process that we just released will do a send to the free_mbox.
   * We will receive process_count times to get each of those messages.
   * After we get all of the messages, the semaphore can be marked as
   * available for future allocation.
   */
  for (i = 0; i < process_count; i++)
    MboxReceive(SemTable[semaphore].free_mbox, NULL, 0);
//MORE CODE is needed to conclude function
```