Phase 3 - Interprocess Communication

Due Dec 13 by 11:59pm **Points** 100

Project Repositories

Teams will use existing team project repositories for development and project submissions.

Project Phase Submissions

Submissions must be on the main branch (teams are welcome to create other branches for individual team purposes) but ONLY the main branch will be considered for submissions.

Submissions for phase 3 must be tagged as such:

- phase-3.0

You may re-tag any commit as a submission. If you have additional submissions PAST the due date with corrections to be made, you may tag them with a sub version, such as:

- phase-3.1
- phase-3.2
- …

Phase 3 Demo

A demo image can be obtained at the following URL:

https://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase3/DemOS.dli (https://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase3/DemOS.dli)

Phase 3 Goals and Requirements

In this project phase, we will extend our operating system's use of system calls and kernel services to allow processes to communicate with each other via Interprocess Communication (IPC). The two methods that will be implemented are semaphores to perform signaling, and message passing via the use of mailboxes.

File Layout/Organization

In addition to your current set of files, you will be adding one new file, outlined below.

Filename	Scope/Context	Description
ipc.h	User/Kernel	Definitions and data structures used in both the kernel and user contexts

Pre-requisites:

To begin phase 3, you will need to apply some changes to your repository from phase 2.

1. Within your repository, run the following to download/apply new files:

```
wget -O - http://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase3/setup.sh 2>/dev/null | b
ash
```

1. Confirm that no errors were encountered.
2. Review the git log to see the new additions/changes

```
git log
```

1. Commit the changes:

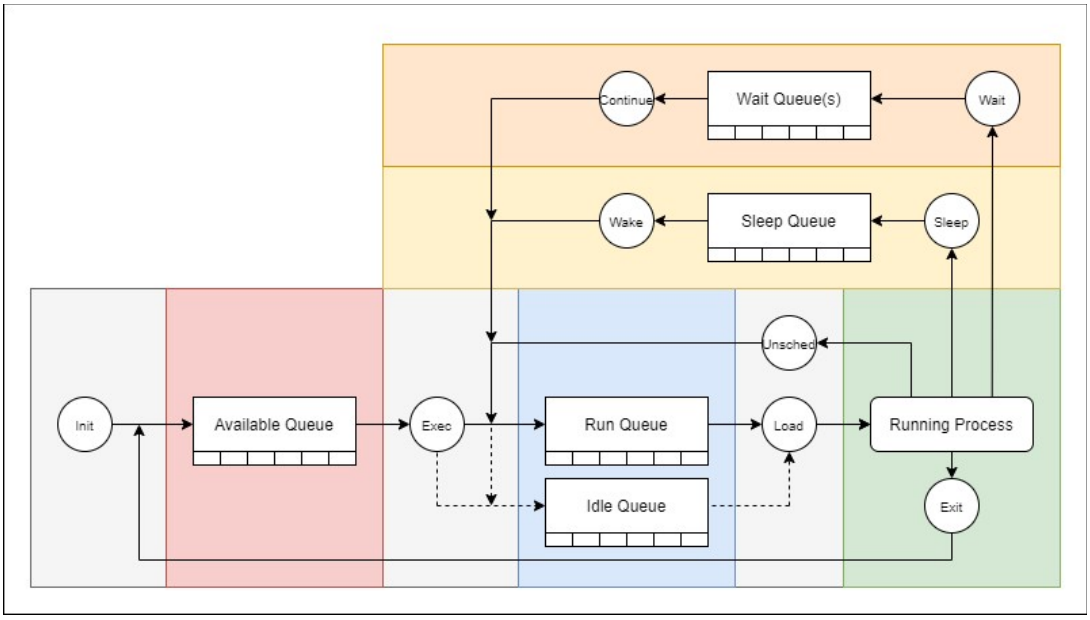
```
git commit -m "Phase 3 pre-requisites"
```

Project Phase 3 Implementation

New process life cycle

In this project phase, we will be implementing a new "WAITING" state for processes that are waiting on some trigger to be performed. Differing from previous phases where a single kernel queue (such as the sleep queue) was used, multiple wait queues (depending on the type of wait operation) will be utilized.

This will result in a new process lifecycle as depicted in the following diagram:



System Calls

A total of five new system calls will be implemented in the same manner that was implemented in phase 2.

User-Space System Call API	Kernel System Call Handler	syscall_t Enumeration
sem_init	ksyscall_sem_init	SYSCALL_SEM_INIT
sem_wait	ksyscall_sem_wait	SYSCALL_SEM_WAIT
sem_post	ksyscall_sem_post	SYSCALL_SEM_POST
msg_send	ksyscall_msg_send	SYSCALL_MSG_SEND
msg_recv	ksyscall_msg_recv	SYSCALL_MSG_RECV

System Call Dispatching

- New system calls must be dispatched from the kernel system call interrupt handler

System Calls (User-Context)

In this project phase, new system calls are implemented to perform new functionality.

You must implement the following system call APIs. Make note of the function return types and

parameters when implementing.

- sem_init()
- sem_post()
- sem_wait()
- msg_send()
- msg_recv()

System Call Handlers (Kernel Context)

sem_init / ksyscall_sem_init System Call

Initializes the semaphore that is shared between one or more processes. This must be called before either sem_wait() or sem_post() is called.

The "semaphore" passed in is a pointer to a variable that will contain the semaphore id. By default, this variable should be set to "SEMAPHORE_UNINITIALIZED" (-1).

If a call to sem_init() is called on a semaphore that is not already initialized (i.e. the semaphore id is set to SEMAPHORE_UNINITIALIZED), it should:

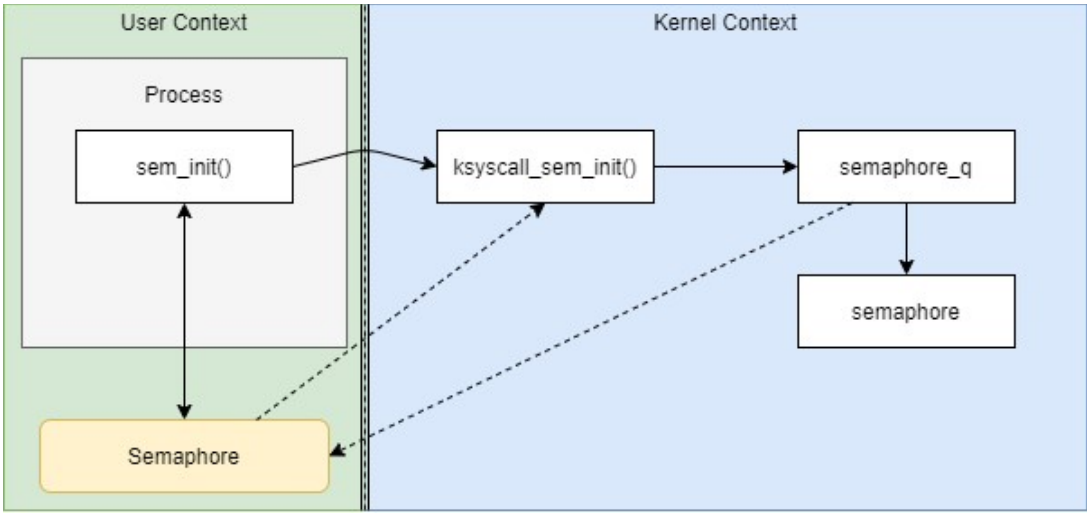
Check if the semaphore has indicated it is initialized in the kernel (via the semaphore init value). If not, you should:

- Dequeue a semaphore from the semaphore queue
- Set the semaphore id to the item that was dequeued
- Ensure that the semaphore count is set to 0
- Ensure that the semaphore init flag is set to SEMAPHORE_INITIALIZED

If a call to sem_init() is called on a semaphore that is already initialized (i.e. the semaphore id is NOT SEMAPHORE_UNINITIALIZED) and the semaphore init flag is set to SEMAPHORE_INITIALIZED it should:

- Obtain the semaphore id from the pointer passed in
- Ensure that the semaphore count is initialized to 0

The following diagram outlines the user context API and system call handler interactions:



sem_wait / ksyscall_sem_wait System Call

Waits on a semaphore to be posted.

For the passed in semaphore, determine if the semaphore id is valid. If it is not valid, panic.

If the semaphore count is > 0, then it means that at least one process is already waiting. In this case, the process should be unscheduled and moved into the wait queue for the given semaphore. The process state should be WAITING in this case.

The semaphore count should be incremented whenever a call to sem_wait is performed.

The same diagram for sem_init() applies to sem_wait().

sem_post / ksyscall_sem_post System Call

Posts a semaphore and releases the first process that was waiting on the semaphore.

For the passed in semaphore, determine if the semaphore id is valid. If it is not valid, panic.

If the semaphore has a process that is waiting, move the process from the semaphore wait queue to the kernel run queue. Ensure that when this happens, the process state is set to READY.

If the semaphore count is > 0, it should be decremented.

The same diagram for sem_init() applies to sem_post().

msg_send / ksyscall_msg_send System Call

Sends a message to the specified mailbox. This is a non-blocking operation. The calling process will

proceed once the message is "sent" to the mailbox.

A message is sent by enqueueing it to the specified mailbox.

If the mailbox is full (message queue is full) the system call handler should panic.

If the mailbox has a process in it's wait queue, it should:

- Dequeue from the mailbox wait queue and move the process to the kernel run queue
- Ensure that the state is set to READY
- Obtain the message pointer from the receiving process' trapframe
- Dequeue the message from the mailbox to the receiving process

See the section on Mailbox Message Queues below for details on mailbox message enqueuing and dequeueing.

msg_recv / ksyscall_msg_recv System Call

Receives a message from the specified mailbox. This is a blocking operation - if the mailbox is empty, the process will not proceed - it should wait. If the mailbox has a message, it can be "received" immediately and the calling process can proceed.

If the mailbox has a message

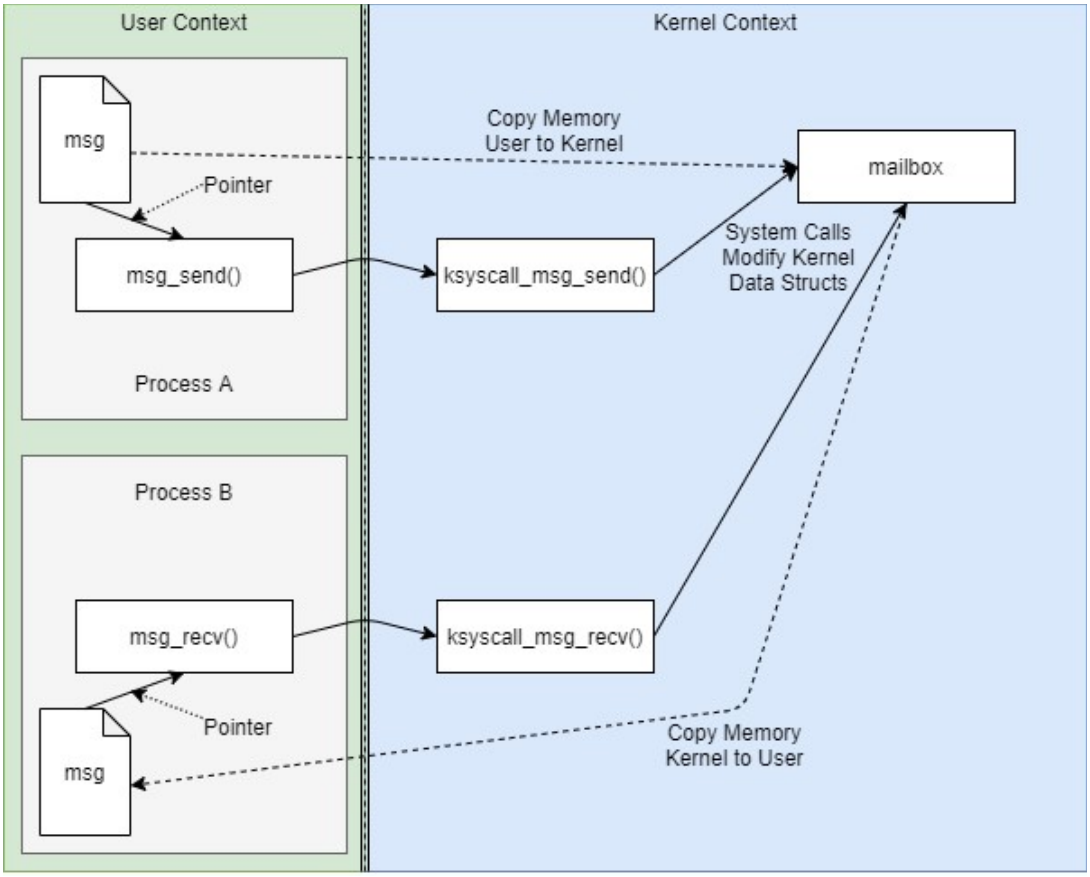
- Dequeue it to the message pointer via the running process' trapframe
- Panic if a message cannot be dequeued

If there is no message in the mailbox

- Move the process to the specified mailbox wait queue
- Set the state to WAITING
- Clear the run pid so another process can be scheduled

See the section on Mailbox Message Queues below for details on mailbox message enqueuing and dequeueing.

The following diagram outlines the message send / message receive interactions between the user system call APIs and the kernel system call handlers.



Data types and structures

In addition to the existing data types and structures from phase 2, you will be adding some new items and modifying or extending additional items.

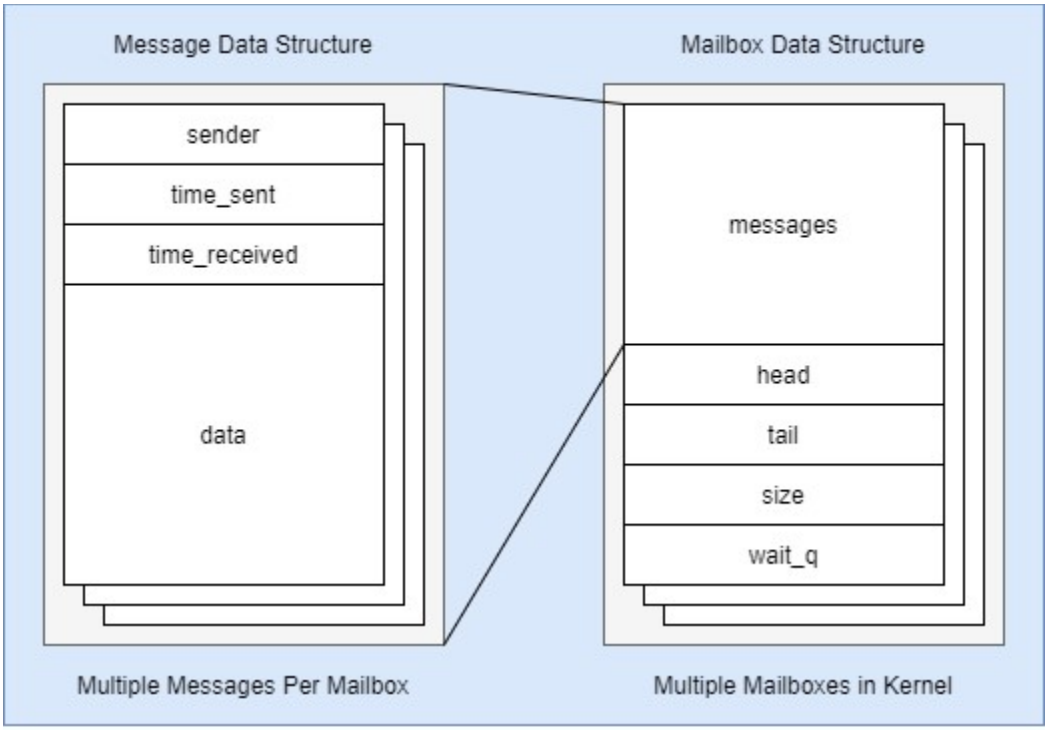
Enumerations

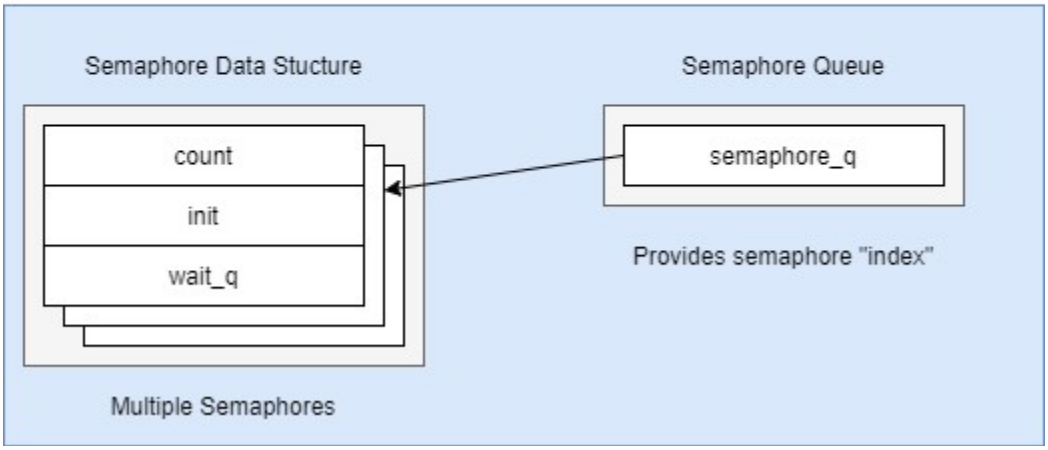
The process state enumeration will be extended to include a new "WAITING" state:

```
// Process states
typedef enum {
    AVAILABLE,
    READY,
    RUNNING,
    SLEEPING,
    WAITING
} state_t;
```

Data Types and Data Structures

The following diagram outlines the data structures to be implemented in this phase:





In this project phase, new data types are to be created:

ipc.h

- Declares a semaphore state enumeration
- Declares a type definition for a semaphore
- Defines the size (in bytes) for message data
- Declares the message data structures

```
// Semaphore definitions
typedef enum {
    SEMAPHORE_UNINITIALIZED = -1,
```

```
    SEMAPHORE_INITIALIZED = 1
} sem_state_e;

typedef int sem_t;

// Message definitions
#define MSG_SIZE 256

typedef struct msg_t {
    int sender;                     // Sending PID
    int time_sent;                  // Time sent
    int time_received;              // Time Received
    unsigned char data[MSG_SIZE];   // Message data
} msg_t;
```

kernel.h

New kernel definitions and data structures for Semaphores:

```
// Maximum number of semaphores
#define SEMAPHORE_MAX PROC_MAX

// Semaphore data structure
typedef struct {
    int count;                      // Semaphore count
    int init;                       // Indicates if initialized
    queue_t wait_q;                 // Wait queue for the semaphore
} semaphore_t;


// Semaphore Data Stuctures
extern semaphore_t semaphores[SEMAPHORE_MAX];
extern queue_t semaphore_q;
```

New kernel definitions and data structures for mailboxes:

```
// Maximum number of mailboxes
#define MBOX_MAX PROC_MAX

// Size of each mailbox
#define MBOX_SIZE PROC_MAX

// Mailbox data structures
typedef struct {
    msg_t messages[MBOX_SIZE]; // Incoming messages
    int head;                       // First message
    int tail;                       // Last message
    int size;                       // Total messages
```

```
    queue_t wait_q;                 // Processes waiting for messages
} mailbox_t;


// Mailbox Data Structures
extern mailbox_t mailboxes[MBOX_MAX];
```

Kernel Initialization

The following must be implemented during initialization:

- During kernel initialization, all new data structures need to be initialized
- Any kernel data structures that need to be pre-populated should be populated during initialization

Kernel/Operating System Initiated Processes

- In addition to the current idle task that is executed at the kernel/operating system startup, the following processes should be started:
  - dispatcher_proc
  - printer_proc

Mailbox Message Queues

In ksyscall.c, you will implement two "helper" functions to handle the enqueueing / dequeueing of messages from a given mailbox.

Mailbox Enqueue:

```
int mbox_enqueue(msg_t *msg, int mbox_num);
```

The mailbox enqueue function will behave similar to your normal queue, except that it will use an array of messages versus an array of integers for the items within your queue.

When enqueueing an item, you should copy the message to the specified mailbox message using the source message pointer.

Mailbox Dequeue

```
int mbox_dequeue(msg_t *msg, int mbox_num);
```

The mailbox enqueue function will behave similar to your normal queue, except that it will use an

array of messages versus an array of integers for the items within your queue.

When dequeuing an item, you should copy the message from the specified mailbox message using the destination message pointer.

User Processes

Three user processes have been provided to you. These user processes consist of:

- A "Dispatcher" process
- A "Printer" process
- A "User" process

Dispatcher process (dispatcher_proc)

The dispatcher process should be instantiated by the kernel after the idle task has been started.

The dispatcher process will:

1. receive messages from a mailbox
2. print details about the message to the console
3. wait on a semaphore
4. update shared memory (between it and the printer process)
5. post a semaphore.

Printer process (printer_proc)

The printer process simply displays the value of the shared memory when it changes.

It will:

1. wait on a semaphore
2. print the shared memory value.
3. post the semaphore

User process (user_proc)

This process will be instantiated whenever the 'n' key is pressed in the SPEDE Target console.

This process is an extension of the previous user process function that prints less information during

it's run loop and will send a message to a mailbox just before it exits.