SASTRA UNIVERSITY
ENGINEERING . MANAGEMENT . LAW . SCIENCES . HUMANITIES . EDUCATION

BCSCCS/BICCIC/BITCIT 505R03

OPERATING SYSTEMS LAB

June 2017

B.TECH CSE / IT / ICT
SCHOOL OF COMPUTING
SASTRA UNIVERSITY

Course Objective

To enable the students acquire knowledge on the various concepts of operating systems by doing hands on experiments on *process management*, *concurrency, synchronization, inter-process communication*, *file management*, *processor management* and *memory management*

LIST OF EXPERIMENTS

Process Creation and Management

1) Creation of a child process using fork system call and communication using pipe.

Inter-Process Communication

2) a). IPC using Shared Memory System Call.
   b). Working with Message Queues.

Processor Scheduling

3) Simulation of uni-processor algorithms and comparing their performances.
4) Simulation of thread scheduling approaches.

Concurrency and Synchronization

5) Implementing producer-consumer system.
6) Implementing reader-writer problem.

Deadlock handling

7) Banker's algorithm for deadlock avoidance
8) Implementation of deadlock detection algorithm
9) Implementing solution for dining philosopher's problem.

Memory Allocation

10) Simulate dynamic partitioning and buddy system

Paging

11) Simulate page replacement algorithms.
12) Simulate address translation under paging

Disk Scheduling

13) Disk Scheduling Techniques

Kernel Level Programming

14) Adding a new system call to Linux kernel

1. Creation of a child process and communication

Objective:

To create a child process using fork system call and use pipe for interaction between parent and child

Pre-requisite:

Knowledge of parent-child process, fork and commands.

Procedure:

- ❑ Develop the parent process with code for calls to fork and pipe
- ❑ OS generates child process as a result of fork() call.
- ❑ Parent suspended to invoke child
- ❑ Child process writes a message into pipe and suspends itself.
- ❑Parent process wakes up and read the message from the pipe.

Pre-Lab:

Practice on getpid, getppid commands

Additional Exercises:

File sharing, Creation of multiple children

2. IPC using Shared Memory and Message Queue

Objective:

To implement IPC using shared memory concept and message queues with the help of the Unix functions available.

Prerequisite:

Knowledge of IPC, Shared memory, message queues, syntax and functionalities of the built-in functions for them

Procedure:

- Create the sender process and receiver process
- Create either a shared memory or message queue by including the appropriate header file and making use of the respective functions
- Sender pushes its message into shared memoty/message queue.
- Receiver retrieves the message and presents it to the user

Pre-Lab

Practicing shmat, shmget, Msgget, Msgsnd, Msgrcv

Additional Exercise:

IPC based on chatting application, secured communication

3. Simulate CPU Scheduling algorithms

Objective:

Simulation of CPU Scheduling algorithms

Prerequisite:

Knowledge of scheduling algorithms

Procedure:

- ❑ Input the number of processes to be scheduled
- ❑ Input the CPU burst time of each of the processes.
- ❑ Calculate the "waiting time" and the "average waiting time" of all the processes based on the following scheduling methods:
    - First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin, Priority based, Shortest Remaining Time
- ❑ Compare the mean turn around time and find the algorithm providing the best result.

Pre-Lab:

Waiting Time, Burst Time and Average Waiting Time calculation

Additional Exercise: SRT, Feedback Queue, Real time scheduling

4. Simulate Thread Scheduling

Objective:

To Simulate thread Scheduling algorithms

Prerequisite:

Knowledge of thread scheduling algorithms

Procedure:

- **Load sharing** – Maintain a global queue of processes and whenever a processor is free send a process from the queue.
- **Gang scheduling** – Identify related threads and schedule them simultaneously on all the processors
- **Dedicated processor assignment** – each program is allocated a number of processors equivalent to the number of threads
- Identify the merits and demerits of the above schemes by applying different sample test cases.

Pre-Lab:

Multi-threading, Multi-processors

5. Simulating Producer-Consumer problem

Objective:

Implementation of Producer-Consumer problem using bounded and unbounded variations

Pre-requisite

Knowledge of Concurrency, Mutual exclusion, Synchronization and producer-consumer problem

Procedure:

- Implement producer-consumer program with producers and consumers simulated as threads.
- Employ necessary semaphores for bounded and unbounded implementations
- Run the program to allow the producer and consumer share the buffer by synchronizing themselves through mutual exclusion

Pre-Lab:

Simple programs using Semaphore functions

Additional Exercise:

Multiple producers and consumers

6. Simulating Reader-Writer problem

Objective:

To write a code to solve the readers writers problem based on reader priority and writer priority solution

Pre-requisite

Knowledge of Concurrency, Mutual exclusion, Synchronization and Reader writer problem

Procedure:

- Create a reader process
- Create a writer process
- Implement necessary semaphores
- Implement the programs giving reader priority and writer priority

Pre-Lab :

Semaphore, multi-processes

Additional Exercise:

multiple readers and writers, solution based on message passing

7. Banker's Algorithm for Dead Lock Avoidance

Objective:

Simulate bankers algorithm for dead lock avoidance

Procedure:

- ❑ Get the number of processes and resources
- ❑ Create the following data structures:
- ❑ **Available** – Number of available resources of each types.
- ❑ **Max** – Maximum demand of each process.
- ❑ **Allocation** – Number of resources of each type currently allocated to each process.
- ❑ **Need** – Remaining resource need of each process. (Max-Allocation)
- ❑ Use **Safety algorithm** and **Resource-Request algorithm**.

Pre-Lab:

Prior knowledge of deadlocks and all deadlock avoidance methods.

Additional Exercise:

Deadlock prevention - circular wait, Deadlock recovery, Finding cycle in resource allocation graph

8. Deadlock Detection Algorithm

Objective:

To implement the deadlock detection algorithm

Procedure

- ❑ Construct the Allocation and Available matrices
- ❑ Follow the following steps
- ✓ Mark each process that has a row in the allocation matrix of all zeros
- ✓ Initialize a temporary vector W to equal to the available vector
- ✓ Find an index i such that process i is currently unmarked and the ith row of Q is less than or equal to W. If no such row found terminate algorithm
- ✓ If row found, mark process i and add the corresponding row of the allocation matrix to W.
- ✓ Return to step 3
- ❑ A deadlock exist is there are unmarked processes

Pre-Lab:

Prior knowledge of deadlocks and all three deadlock strategies.

9. Implementing solution for Dining Philosopher's problem

Objective:

To write a code to solve the Dining philosopher problem.

Prerequisite

Knowledge of Concurrency, Deadlock and Starvation

Procedure

- ❑ Create philosopher process
- ❑ Declare semaphore for mutual exclusion and left & right forks
- ❑ Implement function for obtaining fork - takefork
- ❑ Implement function for releasing fork - putfork.
- ❑Implement function for testing blocked philosophers

Pre-Lab:

Semaphore, multi-processes

Additional Exercise:

Solution using monitors, Sleeping barber problem

10. Memory allocation using dynamic partitioning and buddy systems

Objective:

Simulation of memory allocation through dynamic partitioning.

Simulation of buddy systems.

Prerequisite:

Knowledge on the concepts of memory partitioning techniques

Procedure:

- ❑ Input size of the process to be allocated memory
- ❑Allocate memory to the process using the placement algorithm. If memory of that size available then raise an exception.
- ❑When a request for terminating a process received, release the memory of that process and designate it as free memory.
- ❑**Placement Algorithms:**
    - ➢ Best-fit
    - ➢ First-fit
    - ➢ Next-fir

- Compute the external fragmentation under the different placement algorithms.

Buddy Systems

- Keep the total memory as one large partition of size 2^U initially.
- If a request of size s such that s^{U-1} is made then the entire block is allocated
- Otherwise the block is split into two equal buddies
- If one of the buddies is equal to the request then it is allocated else the buddy is split into two equal sized buddies.
- This process is repeated until the smallest buddy equal to or greater than the request s is obtained

Additional Experiments

- Fixed partitioning, Compaction

11. Simulate Page Replacement algorithms

Objective:

Simulate page replacement algorithms.

Prerequisite:

Knowledge of Paging concepts, replacement algorithms

Procedure:

- Input the number of memory frames and the page reference string
- Select a page to replace based on the following approaches:
  - Least Frequently Used
  - Least Recently Used
  - Optimal page replacement

Pre-Lab :

Paging

Additional Exercise:

FIFO, MRU, LFU, Second-chance

12. Simulate Address Translation in Paging

Objective:

To simulate the address translation from logical to physical address under paging

Prerequisite:

Knowledge of Pages, Frames, Memory partitioning

Procedure:

- Get the range of physical and logical addresses.
- Get the page size.
- Get the page number of the data.
- Construct page table by mapping logical address to physical address.
- Search page number in page table and locate the base address.
- Calculate the physical address of the data.

Pre-Lab:

Paging, Page replacement, Address calculation methods

Additional Exercise:

Simulation of thrashing, Implementation of segmentation

13. Disk Scheduling Techniques

Objective:

Simulation of disk scheduling techniques.

Prerequisite:

Knowledge of disk scheduling algorithms.

Procedure:

Implement scheduling algorithms listed below:

- **First-in-First-Out**
- **Shortest Seek Time First**
- **Scan**

Pre-Lab:

Calculation of Seek time, transfer time etc.

Additional Exercise:

C-SCAN, Look

14. Adding a new system call to Linux kernel

Objective:

Adding a our own developed system call to Linux kernel.

Prerequisite:

Knowledge of linux kernel, System calls

Procedure:

1)Download the latest version of the 2.6 Linux kernel from www.kernel.org.
2)Unzip and untar the kernel directory into /usr/src/.
3)In /usr/src/Linux-x.x.x/kernel/, Create a new file myservice.c to define your system call.
4) In /usr/src/Linux-x.x.x/include/asm/unistd.h, define an index for your system call. Your index should be the number after the last system call defined in the list.
5) Also, you should increment the system call count.
6) In /usr/src/Linux-x.x.x/arch/i386/kernel/entry.S, you should define a pointer to hold a reference to your system call routine.Add your object after the other kernel objects have been declared.

7) Add your system call to the Makefile in /usr/src/Linux-x.x.x/kernel/Makefile

8) Make your system from /usr/src/Linux-x.x.x

9) Add a new boot image to Lilo, by editing /etc/lilo.conf. Your lilo configuration will vary slightly.

10) Making a user test file. You also need to copy your edited unistd.h from /usr/src/Linux-x.x.x/include/asm/ to /usr/include/kernel/ because it contains your

system call's index.

11) Reboot into your new kernel and compile your user test program to try out your system call. You will know if it worked if you see a kernel message in /var/log/kernel/warnings announcing that your service is running.

Pre-Lab:

shell commands