Course Title: Linux System Programming

Faculty: Rajesh Sola, Srinivas.K & Bharath.G

L&T Technology Services

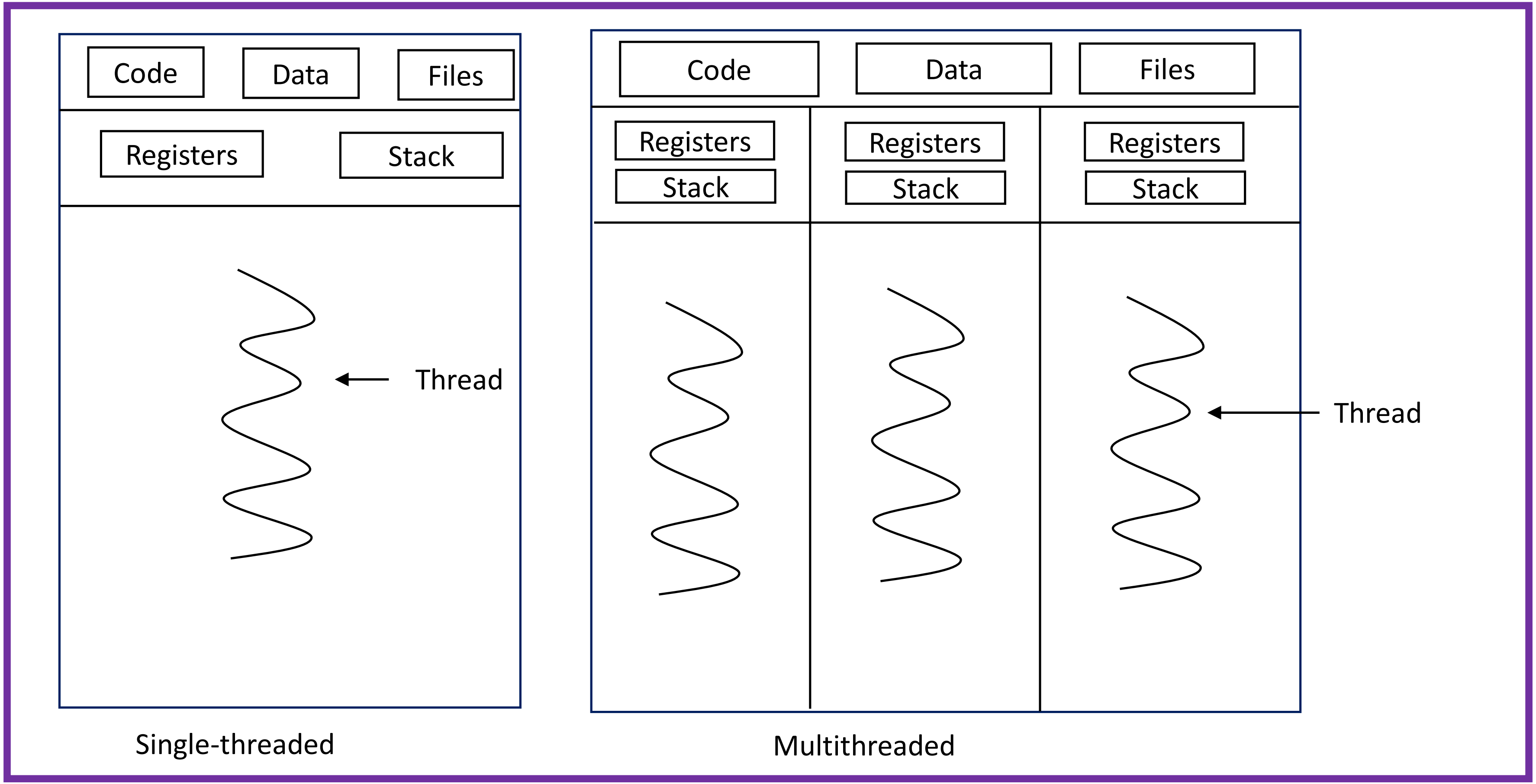
LTTS
GLOBAL ENGINEERING ACADEMY

Threads

Basics of Threads

- Path of execution within a process
- Various sub-activities within applications are referred as threads
- Referred as Light Weight Process (LWP)
- Significance of threads
  - Concurrent execution (parent – child process / multiple child process)
  - RESOURCE SHARING across threads
- Child process will have own resources, but threads will have shared resources
- Scheduled threads interchangeably use CPU based on time sharing
- Every process is run initially as a single thread, then multiple threads spawn
  - Firefox browser initially will be a single thread, on need basis multiple threads spawn
- Threads are faster than fork
- Common resources during execution run independently

Basics of Threads



Advantage of Thread over Process

- Concurrent execution and faster response, less time for context switch
- Effective use of multiprocessor system
- Resource sharing: code, global data, files can be shared among threads
  - PC, Stack and Registers is separate for each thread
  - Private / local data is not shared
- Easier communication between threads
- Enhanced throughput of the system
  - Number of jobs completed per unit time

Note:

If one thread makes a blocking call, whole process gets blocked.

Thread Models

- Types of threads
  - User threads
    - Threads used by application programmers, are above kernel and without kernel support
  - Kernel threads
    - Supported within kernel, perform multiple simultaneous tasks to serve multiple kernel system calls
- Models
  - Used to map user threads to kernel threads
  - Many to One model
    - Many user-level threads are mapped to single kernel thread, thread management is handled by thread library in user space
  - One to One model
    - Separate kernel thread is created to handle each and every thread, limitation is the count of threads that can be created
  - Many to Many
    - Many user-level threads are mapped to multiple kernel level threads

Commands

- ps –e –L –o pid,ppid,lwp,nlwp,stat,cmd
- ps –eLf

- To create threads, POSIX thread library is used
  - pthread_create
  - pthread_join
  - pthread_self
  - pthread_equal
  - pthread_yield
  - pthread_cancel

- gcc psample.c -lpthread

Inter process communication (IPC)

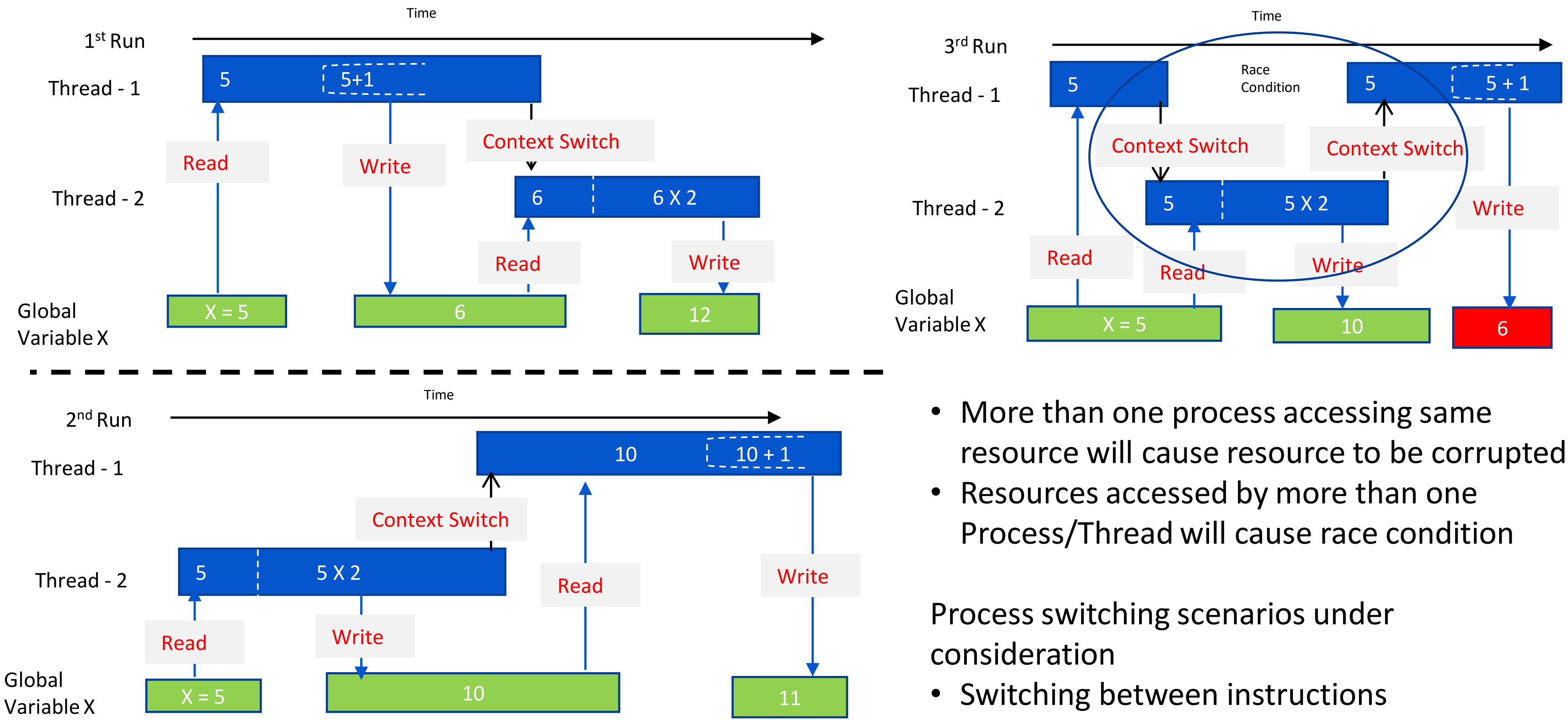
IPC

Requirement of IPC

- Data exchange
- Synchronization
  - Dependency / Sequencing
  - Mutual Exclusion
- Data exchange → shared memory, message queues, FIFOs/pipes
- Mutual exclusion → semaphore, mutex, spinlocks
- Dependency →semaphores, condition variables / event flags

Process that writes/updates data is **PRODUCER** and process that reads is **CONSUMER**

Race conditions



- More than one process accessing same resource will cause resource to be corrupted
- Resources accessed by more than one Process/Thread will cause race condition

Process switching scenarios under consideration

- Switching between instructions
- Switching before/after instructions

Critical section & Mutual Exclusion

Critical Section: Code/Instructions in a Process/Thread using shared resources

- During process execution in critical section, no switching should be allowed
- Only one Process/Thread can be in a related critical section at any given time.
- Should be as short as possible & no blocking calls

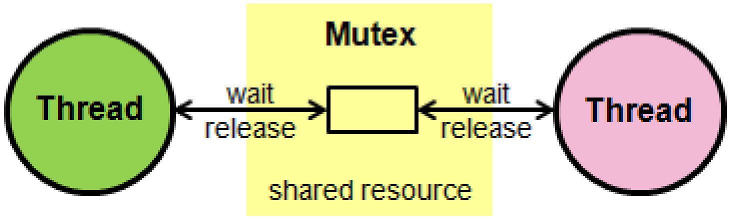
Mutual exclusion: Preventing simultaneous access to shared resources

- Disable interrupts (for very shorter duration)
        - - User space cannot have access
        - - For longer duration, inconsistency occurs
        - - Other CPU can access the resources
- Hardware support instructions
    - Atomic operation: Programming operations independent of other processes
        - - Resources can't be accessed by other process
    - Data bus locking techniques: CPU level bus locking techniques
- Above techniques have limitations and not scalable
- Software level solution for Mutual exclusion is semaphore & mutex

Semaphore & Mutex

Mutex

- Mutual Exclusion
- Only locked Process(es)/Threads can unlock the resources
- Any other Process/Threads trying to unlock is referred as “unauthorized operation”
- Unlocking twice or unlocking before locking is not allowed
- Strictly lock & unlock in the same thread only
- Mutex will have "ownership" as compared to semaphore



Mutex API's

- #include <pthread.h>
- pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER (declare & initialize)
- pthread_mutex_init(&m1)
- pthread_mutex_lock(&m1) (lock)
- pthread_mutex_unlock(&m1) (unlock)
- pthread_mutex_destroy (&m1) (destroy)

Always check return value for Success or Failure

Semaphores

- Sequencing, Signaling mechanism, used for process/thread synchronization
- Manage and protect access to shared resources
- Kernel level data structure

Types of usage

- Binary Semaphore
  - Value of semaphore ranges between 0 & 1
  - Mutual Exclusion / Access to a single resource
- Counting Semaphore
  - Value of semaphore can be 0 (zero) & any positive value
  - Accessing/sharing multiple similar resources

Two (2) varieties of semaphores

- Traditional System V semaphores
- POSIX semaphores.

Two (2) types of POSIX semaphores

- Named
- Unnamed

Named Semaphore

Name is given to semaphore and can be access by parent & child or different processes

- Uses internal shared memory for resources access

POSIX API’s

#include <semaphore.h>

#include <errno.h>

- sem_t *ps; (declare a semaphore variable)
- ps = sem_open(“/s1", O-CREAT, 0666, 1) (internal shared memory)
- sem_wait(ps) (lock the semaphore)
- sem_post(ps) (unlock the semaphore)
- sem_close(ps) (close semaphore from process)
- sem_unlink(ps) (remove named semaphore)

All calls return 0 on success, -1 on error and ‘**errno**’ variable is set to error number

Unnamed Semaphores

No name is given to the Semaphore.

- Memory is allocated in the program address space

POSIX Unnamed Semaphore API’s

#include <semaphore.h>

#include <errno.h>

- sem_init(sem_t *sem, int pshared, unsigned int value) (Initialize unnamed semaphore)
- sem_wait(sem_t *sem) (Lock the semaphore)
  - Check sem_trywait & sem_timedwait
- sem_post(sem_t *sem) (Unlock the semaphore)
- sem_destroy(sem_t *sem) (Destroy the semaphore)

All calls return 0 on success, -1 on error and **‘errno’** variable is set to error number

Produce and Consumer Problem

Producer and Consumer scenario

- A Process/Thread will add data – Producer
- A Process/Thread will remove data – Consumer
- Common Buffer/Data Source
- Either Producer or Consumer only can access common data at a time (Shared resource)
- Consumer should block if buffer empty
- Producer should block if Buffer full

Deadlock

Two or more processes infinitely blocked (forever) due to circular dependency of resources

- Digital Copy – Printer(s1), Scanner(s2) Problem
- Arbitrary locking of multiple semaphores
- Parent & child - unlocking semaphore after waitpid
- Producer consumer problem - order of locking

Avoid deadlock

- If multiple locks are required, lock all of them at once (atomic locking)
- Don't apply mutual exclusion, before resolving dependency

Limitations of Semaphore and Mutex as a method of IPC

- Semaphores & Mutex can never carry data
- Processes / threads need to carry data or exchange the data

Data Exchange

Limitations of Semaphore and Mutex as a method of IPC

- Semaphores & Mutex can never carry data
- Processes / threads need to carry data or exchange the data

Need for other IPC Mechanisms

- Pipes/FIFO
- Message Queue
- Shared Memory

FIFO/Pipes

FIFO/Pipes

Pipe is a connection between two related processes

- Pipe is one-way communication only
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- For two way communication using pipes, two pipes should be used.
    - Process-1 writes to Pipe-1 & reads from Pipe-2
    - Process-2 reads from Pipe-1 & writes to Pipe-2

Named Pipe/ FIFO

- Connection between two unrelated processes

  int mkfifo(const char *pathname, mode_t mode)

- mkfifo mypipe, tail -f mypipe

Example

Pipes

System Calls related to pipe

#include <unistd.h>

- int pipe(int pipedes[2]) (Create unnamed pipe)
- ssize_t write(int fd, void *buf, size_t count) (Write to pipe)
- ssize_t read(int fd, void *buf, size_t count) (Read from pipe)
- int close(int fd) (Close pipe)

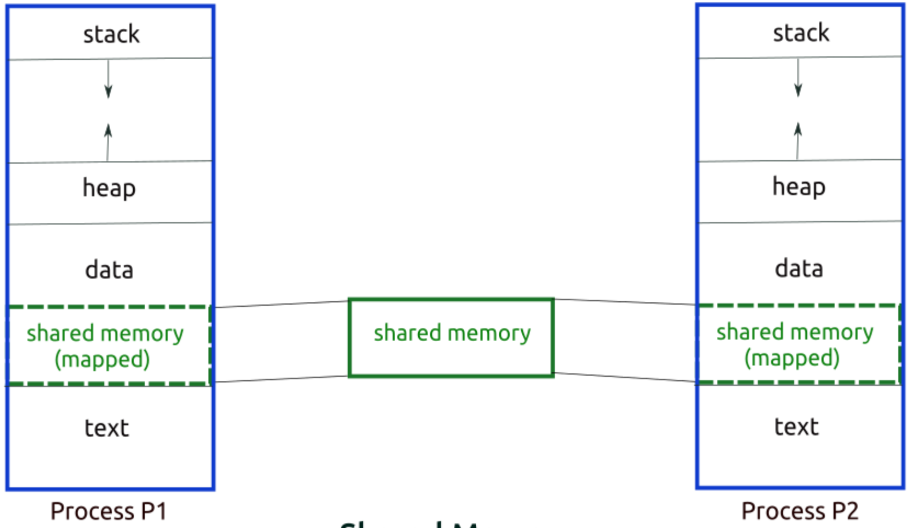
- Advantages and Disadvantages

Shared Memory

Shared Memory

Memory Segment is created by the kernel and mapped to the data segment of the address space of a requesting process

Can be used like a global variable in address space



Shared Memory

Shared Memory

- int shm_open (const char *name, int oflag, mode_t mode);

Create, or gain access to, a shared memory object.

- void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Map a shared memory object into its address space.

Do operations on shared memory (read, write, update).

- int munmap (void *addr, size_t length);

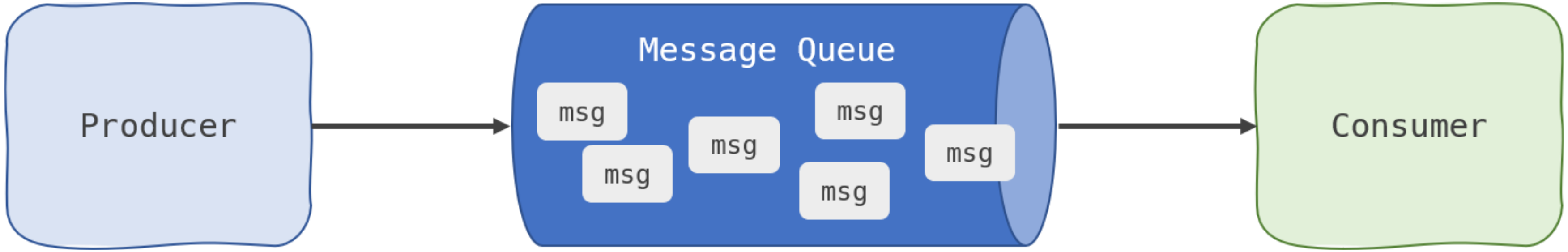
Delete mappings of the shared memory object.

- int shm_unlink (const char *name);

Destroy a shared memory object when no references to it remain open.

Message queues

Message Queues



The messages from Producer are stored on queue & provided on-demand to Consumer

- Typically FIFO based, can also be priority based
- Messages with same priority are read in FIFO order

Synchronization

- On read, if queue is empty, the receiver is blocked
- On write, if the queue is full, sender will be blocked
- Messages are discrete

Message Queues

```
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <mqueue.h>
```

- mqd_t mq_open(const char *name, int oflag)
- mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr)
- int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio)
- ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)
- int mq_close(mqd_t mqdes)
- int mq_unlink(const char *name)

Link with -lrt

Queries?

Thank You !

L&T Technology Services