INTER PROCESS COMMUNICATION (IPC)

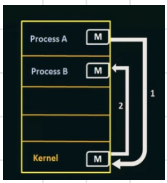
↳ Mechanisim for Processes executing concurrently to communicate and synchronise their actions

MESSAGE PASSING

↳ TWO OPERATIONS

↳ send message → fixed size message/variable

↳ receive message



In order to communicate using message passing

↳ establish a communication link b/w them

↳ Physical (shared memory/ hardware bus)

↳ logical (logical properties)

↳ exchange messages via send/receive

Priority Scheduling

↳ CPU allocated by priority

CON: Starvation → low priority may never execute

SOLUTION: Aging → increase priority as time progresses

turn around time: Completion – Arrival time

Waiting time: Total waiting time – ms process executed – Arrival time

↓ Preemptive

Waiting time: turn around time – burst time

↓ non preemptive

PREMPTIVE

Process ID	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

olution: Gantt Chart:

0	2	5	33	40	49	51	67
P1	P4	P2	P4	P1	P3	P5	

Waiting time: (40-2-0) + (5-0-5) + (49-0-12) + (33-3-2) + (51-0-9)

avg = 29/5

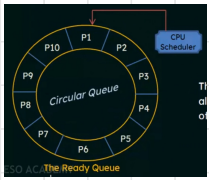
NON PREMPTIVE

→ best for time sharing

Round Robin (RR) Scheduling

↳ small time slices for each process (time quantum)

↳ FCFS but has premption



FIFO in ready queue

Tail → new proceses

Head → nex process to get CPU

2 Possibilities

↳ burst time < time quantum

↳ CPU released

↳ CPU scheduler gets next process in ready queue

↳ burst time > time quantum

↳ timer goess off

↳ interrupt to OS

↳ context switch, process moved to tail of ready queue

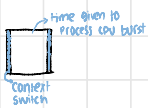
↳ CPU scheduler gets next process in ready queue

CONS

time quantum

↳ too small : too many context switches

↳ too big : starve for long time



Process ID	Burst Time
P1	24
P2	3
P3	3

Gantt Chart:

0	4	7	10	14	18	22	26	30
P1	P2	P3	P1	P1	P1	P1	P1	

Method 1

Turn Around time = Completion time - Arrival time

Waiting time = Turn Around time - Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	30 - 0 = 30	30 - 24 = 6
P2	7	7 - 0 = 7	7 - 3 = 4
P3	10	10 - 0 = 10	10 - 3 = 7

Average Turn Around time
= (30 + 7 + 10) / 3
= 47 / 3 = 15.66 ms

Average waiting time
= (6 + 4 + 7) / 3
= 17 / 3 = 5.66 ms

Method 2

Waiting time = Last Start Time - Arrival Time - (Preemption x Time Quantum)

Process ID	Waiting Time
P1	26 - 0 - (5x4) = 6
P2	4 - 0 - (0x4) = 4
P3	7 - 0 - (0x4) = 7

Average waiting time
= (6 + 4 + 7) / 3
= 17 / 3 = 5.66 ms

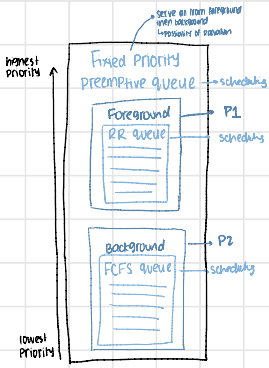
NO TALKING CLASS SAY BAHIR

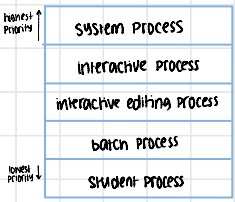
Multi-level Queue Scheduling

Scheduling algorithms for situations in which processes are classified in different groups

E.g

Foreground (interactive)	Background (batch)
↳ need to be quick	↳ doesn't need to be as fast
↓	↳ FCFS
Priority will be higher	
↳ Round Robin	





↳ Partions ready queue into several seperate queues

↳ Scheduling takes place within and among these queues

↳ Each process permamentaly assigned to 1 queue

based on memory size, Piority and process type

↳ Each queue has it's own scheduling algorithm (FCFS, SJF, RR...)

Multi-level Feedback Queue Scheduling

↳ allows processes to move between queues

↳ seperate processes according to their CPU bursts

↳ If a process uses too much CPU time

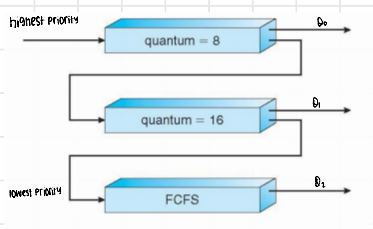
then moved to lower priority queue (so everybody gets a chance)

↳ I/O and interactive processes are in higher Priority queues

as they need quick responses

↳ A process in lower priority queue for too long → aging

may be moved to higher priority queue (prevents starvation)



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Parameters

↳ no. of queues

↳ scheduling algo for each queue

↳ method to upgrade process to higher priority

↳ method to demote process to lower priority

↳ method to determine queue when process needs service

↳ demote/upgrade

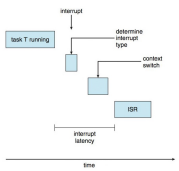
Multi-Processor Scheduling

NOTES MISSING

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity:** When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so known as **soft affinity**.
- **hard affinity:** allowing a process to specify a subset of processors on which it may run.

Real-Time CPU Scheduling

- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

LECTURE 19,20,21

Process Concept

↳ Process are executed programs that have

↳ Resource ownership — unit → process/task

↳ Process includes virtual space (to hold process image)

↳ OS prevents unwanted interference b/w processes

↳ Scheduling / Execution

- Process follows an execution path that may be interleaved with other processes
- Process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the operating system

- Today, the unit of dispatching is referred to as a **thread** or **lightweight process**

MOTIVATION

- Most modern applications are multithreaded
- Threads run within application (process)
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Thread control block

↳ information associated with each thread

↳ Program counter ↳ CPU registers

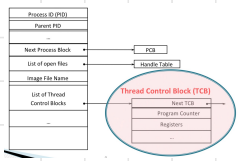
↳ CPU scheduling info ↳ Pending I/O info

Process control block (PCB)

↳ information associated with each process

↳ Memory management info

↳ Accounting info



Thread operations

↳ spawn : a thread within a process may spawn another thread

↳ provides new thread instruction pointer, arguments, register, stack

↳ block : a thread needs to wait for an event

↳ saves its user registers, program counter, stack pointers

↳ unblock: when the event for which the block occurs

↳ Finish: when thread completes

↳ its register context and stacks are dellocated

Thread states

↳ running

↳ ready

↳ blocked

Multithreaded Applications

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Linux Kernel is also multithreaded

Single vs Multithreaded Webserver

- How a web-server (as a single process) increase its response time?

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead?

Single vs Multithreaded Webserver

- Assume: 1000~ request per second
  - Performance of single threaded server
  - Working of a multi-threaded server.
  - What benefits do we get using a multithreaded server?

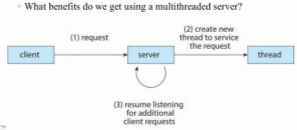


Figure 4.2 Multithreaded server architecture.

Process

↳ A program in execution

↳ can have 1 or more threads

Threads

↳ unit of execution within a process

↳ It compromises of

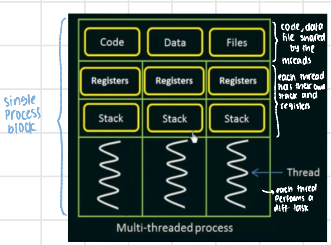
1. Thread ID
2. Program counter
3. Register set
4. Stack

↳ shares code section, data section, os resources with other threads of same processes

S.N.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

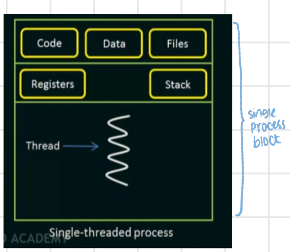
Multi Thread

↳ Performs multiple tasks at a time



Single Thread

↳ Performs one task at a time



MULTI-THREADING BENEFITS

↳ Responsiveness: may continue execution if process is blocked/performing lengthy operation
dedicated threads for handling user events

↳ Resource Sharing: threads share memory and recources of a process → allows diff threads activity within same address space
→ easier than shared memory/message passing

↳ Economy: cheaper than process creation as threads share recources
allocating memory/recources is costly
thread switching has lower overhead than context switching

↳ Scalibility: utilization of multiple cores for parallel execution
increases concurrency

Multicore Programing

PARALLELISM

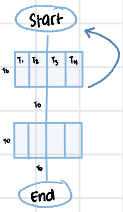
↳ act of managing multiple computations simultaneously

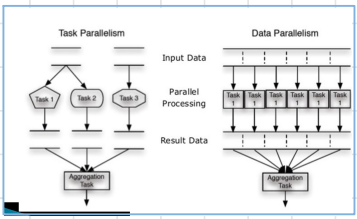
DATA PARALLELISM ← IMP → TASK PARALLISM

↳ focus on distributing data across diff parallel computing nodes

↳ focus on distributing threads across diff parallel computing nodes

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.





AMDAHL'S LAW

↳ speed up in latency of a task execution

$$\text{Speed up} \leq \frac{1}{S + \frac{1-S}{N}}$$

S → serial, N → no of cores, (1−S) → parallel

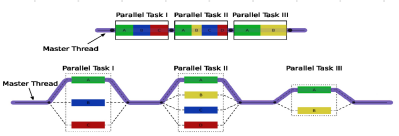
B) $S = 25\%$, $N = 2$

$$\text{Speed up} \leq \frac{1}{0.25 + \frac{0.75}{2}} = 2.28$$

CONCURRENCY

↳ act of managing multiple computations at the same time but not simultaneously → control is switched

Fork-Join model



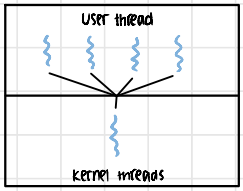

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve part
        join all subtasks spawned in previous loop
        combine results from subtasks
```

- Multicore systems putting pressure on programmers, challenges include
  - Dividing activities
    - What tasks can be separated to run on different processors
  - Balance
    - Balance work on all processors
  - Data splitting
    - Separate data to run with the tasks
  - Data dependency
    - Watch for dependences between tasks
  - Testing and debugging
    - Harder!!!!

MultiThreading → multiple threads at the same time

establish relationship between user and kernel thread

Many to one



PROS

↳ multiple user threads mapped to 1 kernel thread

↳ thread management is done by threaded library → so efficient

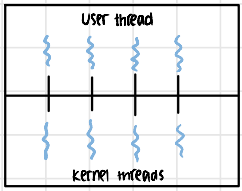
CONS

↳ if 1 thread makes a blocking system call entire process will be blocked

↳ only 1 thread can acess kernel at a time.

↳ multiple threads are unable to run in parallel on multiprocessors

One to one



PROS

↳ maps each user thread to kernel thread

↳ more concourrency → as allows another thread to run when a thread makes a blocking system call

↳ allows multiple threads to run parallel on multi processors

CONS

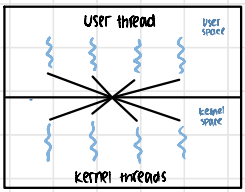
↳ each user thread requires a corresponding kernel thread

↳ overhead of creating kernel threads burdens performance

restricts no. of threads supported by the system

↳ e.g ↳ 4 core processor
↳ 5 threads
then only 4 threads work parallel
hence no of threds restricted

Many to Many



PROS

↳ kernel threads are ≤ to user threads

↳ more concorcuncy

↳ no limit to creating user threads

↳ kernel threads can run parallel on a multi processor

Hyper Threading / SMT (Simultaneous Multi Threading)

↳ more than 1 multithreading going on in the same system

e.g ↳ 4 core → virtually/logically divided into multiple processors
↳ 8 threads support

TYPES OF THREADS

user threads

↳ supported above kernel

↳ managed w/o kernel support

↳ thread management done by user level threads library

kernel threads

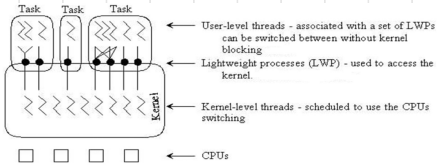
↳ supported and managed directly by the operating system

LIBRARIES

↳ POSIX PThreads

↳ Win 32 Threads

↳ Java Threads



- Linux kernel has processes and lightweight processes (LWP).
- Difference: LWPs share same address space and other resources like open files etc. As some resources are shared so these processes are considered to be light weight as compared to other normal processes and hence the name light weight processes.
- Therefore, effectively we can say that threads and light weight processes are same.
- Thread is the term used at the user level while LWP is a term used at kernel level.

- **POSIX Pthreads**
  - May be provided either as user-level or kernel-level
  - A POSIX standard (IEEE 1003.1c) API for thread creatio and synchronization
  - API specifies behavior of the thread library, implementation is up to development of the library
- **Win32**
  - Kernel-level library on Windows system
- **Java**
  - Java threads are managed by the JVM
  - Typically implemented using the threads model provided by underlying OS

Threaded Libraries

$$SUM = \sum_{i=1}^{N} i$$

Asynchronus Threading

↳ once parent creates child thread, parent resumes execution

↳ they execute concurrently and independently of one another

↳ Threads are independent → no dependency

↳ little data sharing

Synchronus Threading

↳ Once parent creates 1 or more children

↳ it waits all child to terminate before it resumes

↳ Threads by parent work cocurrently

↳ significant data sharing among threads

POSIX

POST LINUX COMPILATION

On Linux, programs that use the Pthreads API must be compiled with
-pthread or *-lpthread*

gcc -o thread -lpthread thread.c

THREAD CREATION

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);

- *thread* Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.
- *attr* Is the thread attribute object specifying the attributes for the thread that is being created. If *attr* is NULL, the thread is created with default attributes.
- *start* Is the main function for the thread; the thread begins executing user code at this address.
- *arg* Is the argument passed to *start*.

THREAD ID

```
#include <pthread.h>

pthread_t pthread_self()
```
returns : ID of current (this) thread

WAIT FOR THREAD COMPLETION

```
#include <pthread.h>

pthread_join (thread, NULL)
```
returns : 0 on success, some error code on failure.

THREAD TERMINATION

```
#include <pthread.h>

Void pthread_exit (return_value)
```

Threads terminate in one of the following ways:

- The thread's start functions performs a return specifying a return value for the thread.
- Thread receives a request asking it to terminate using pthread_cancel()
- Thread initiates termination pthread_exit()
- Main process terminates

```
int main()
{
    pthread_t thread1, thread2;  /* thread variables */
    thdata data1, data2;         /* structs to be passed to threads */

    /* initialize data to pass to thread 1 */
    data1.thread_no = 1;
    strcpy(data1.message, "Hello!");

    /* initialize data to pass to thread 2 */
    data2.thread_no = 2;
    strcpy(data2.message, "Hi!");

    /* create threads 1 and 2 */
    pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &data1);
    pthread_create (&thread2, NULL, (void *) &print_message_function, (void *) &data2);

    /* Main block now waits for both threads to terminate, before it exits
       If main block exits, both threads exit, even if the threads have not
       finished their work */
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    exit(0);
}
```

Example code but not complete

Race condition

↳ when several processes manipulate/acess the same data concurrently and outcome depends on order in which acess take place

results in inconsistency

SOLUTION

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Process Synchronization

The orderly execution of cooperating process that share an address space

Producer and consumer run cocurrently using

BUFFER

unbounded buffer

↳ has a size on buffer

↳ consumer waits for new items

↳ producer can always produce new items

bounded buffer

↳ fixed buffer size

↳ consumer must wait if buffer empty

↳ producer must wait till buffer full

↳ use counter variable (++ produced, -- consumed)

Critical section problem

↳ a system with n processes

↳ each process has a segment of code called critical section

↳ where process may be changing common variables
updating table
writing a file

Three requirements

↳ **mutual exclusion:**

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

↳ **Progress:**

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

↳ **bounded waiting:**

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's solution → humble

↳ software solution to the critical section problem

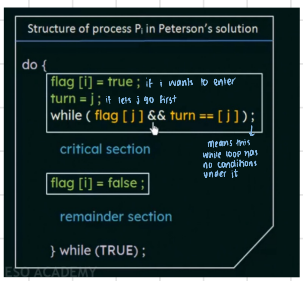
↳ 2 processes that alternate execution → b/w critical section, remainder section

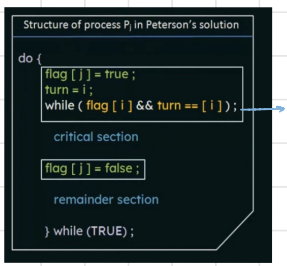
↳ the 2 processes share 2 variables

↳ int turn ; → indicates whose turn to enter critical section

↳ boolean flag[2]; → indicates if process ready to enter critical section

↳ satisfies all 3 critical section requirements





→ satisfies mutual exclusion

Test and set Lock

↳ hardware solution to the critical section problem

↳ processes share shared lock variable = 0 → unlocked, 1 → lock } helps satisfy mutual exclusion

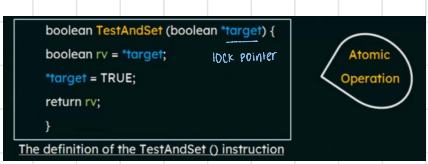
* lock = 0 in begining

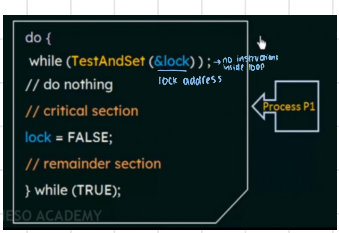
↳ if lock = 1, then wait till it becomes free

↳ if lock = 0, then execute critical section and lock = 1

↳ satisfies mutual exclusion

↳ does not satisfy bounded-waiting





when while = 0 → F
= 1 → T

Atomic Operation → hardware instruction

↳ single, uninterrupted operation

eg

TestAndSet() is run as a single operation which can not be interrupted

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Compare and Swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

- If two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

   /* critical section */

   lock = 0;

   /* remainder section */
}
```

On Intel_x86 architectures

```
lock cmpxchg <destination operand>, <source operand>
```

Mutex locks/spin lock

↳ to protect critical section/prevent race conditions

↳ aquire() } both calls must be atomic

↳ release() }

↳ requires busy waiting

while process in critical section any other process that tries to enter critical section must loop continuously to check if condition true

↳ wastes CPU cycles → that some other process could use productively

-
```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```
-
```
release() {
   available = true;
}
```
-
```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Pthreads Synchronization – Mutex Lock

#include <pthread.h>
Pthread_mutex_t mutex;

/* create the mutex lock */
Pthread_mutex_ init(&mutex,NULL);

1st Arg: Mutex initiliazer
2nd Arg: Attributes, NULL means no error checks will be performed

Pthreads Synchronization – Mutex Lock

- The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions.
- If the mutex lock is unavailable when pthread_ mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_ unlock().

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
```

```
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization

Semaphore

↳ Software solution to the critical section problem

↳ technique to manage concurrent processes

↳ an int variable S (always +ve) shared b/w threads
(S: semaphore)

↳ S can only be accessed through 2 atomic operations

↳ wait() → to decrement
S<=0 → some process already in critical section (so cant use shared resource) → wait
S>0 → can use shared resource → dont wait

↳ signal() → to increment → tell other processes that shared resource is free to be used

Definition of wait ():

```
P (Semaphore S) {
    while (S <= 0)
      ; // no operation
    S-- ; → when S>0
}
```

Definition of signal ():

```
V (Semaphore S) {
    S++ ;
}
```

* S=1 in begining

indivisibility

when 1 process modifies semaphore

no other process can simultaneously modify that semaphore value

TWO TYPES OF SEMAPHORES

Binary Semaphore

↳ S can only have 2 values
0 → some process already executing so not free to use shared resource
1 → free to use shared resource

↳ behave similarly to mutex locks (Provide mutual exclusion)

↳ can implement a counting semaphore S as a binary semaphore

Counting semaphore

↳ S can multiple values

↳ used to control acess to a resource that has multiple instances

↳ S = no. of instances of a resource

Semaphore Disadvantage

↳ requires **busy waiting** → while process in critical section any other process that tries to enter critical section must loop continuously to check if condition true

↳ wastes CPU cycles → that some other process could use productively

Solution

↳ modify wait() and signal()

to block() → Places process in waiting queue (waiting state)

to wake up() → move process from waiting to ready queue

↳ instead of busy waiting, block itself

↳ transfers control to CPU scheduler (which selects another process to execute)

hence no wastage of CPU time

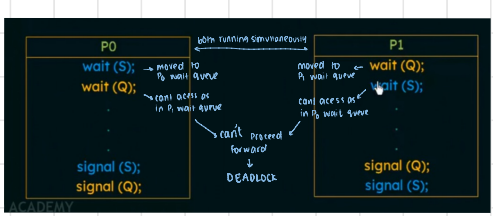
PROBLEM

↳ Deadlock

↳ set of blocked processes, each holding a resource and waiting to aquire a resource, held by another process

↳ Starvation

↳ when a process is postponed because it requires a resource to run, that is never allocated to this process



Semaphore

↳ Software solution to the critical section problem

↳ technique to manage concurrent processes

↳ an int variable S (always +ve) shared b/w threads
(S → semaphore)

↳ S can only be accessed through 2 atomic operations

↳ wait() → to decrement
S<=0 → some process already in critical section so can't use shared resource → wait
S>0 → can use shared resource → don't wait

↳ signal() → to increment → tell other processes that shared resource is free to be used

```
Definition of wait ():
P (Semaphore S) {
    while (S <= 0)
    ; // no operation
    S-- ; → when S>0
}
```

```
Definition of signal ():
V (Semaphore S) {
    S++ ;
}
```

* S=1 in begining

TWO TYPES OF SEMAPHORES

Binary Semaphore

↳ S can only have 2 values
0 → some process already executing so not free to use shared resource
1 → free to use shared resource

↳ behave similarly to mutex locks (provide mutual exclusion)

↳ can implement a counting semaphore S as a binary semaphore

Counting Semaphore

↳ S can multiple values

↳ used to control acess to a resource that has multiple instances

↳ S = no. of instances of a resource

SEMAPHORE DISADVANTAGE

↳ requires **BUSY WAITING**

↳ while a process is in critical section any other process that tries to enter critical section must loop continously to check if condition true

↳ wastes CPU cycles → that some other process could use productively

* This type of semaphore also known a spinlock (as process 'spins' waiting for lock)

SOLUTION TO BUSY WAITING

↳ modify wait() and signal()

to block() → Places process (waiting state) in waiting queue

to wake up() → move process from waiting to ready queue

↳ instead of busy waiting, block itself

transfers control to CPU scheduler (which selects another process to execute)

hence no wastage of CPU time

PROBLEM

↳ **Deadlock**

↳ set of blocked processes, each holding a resource and waiting to aquire a resource, held by another process

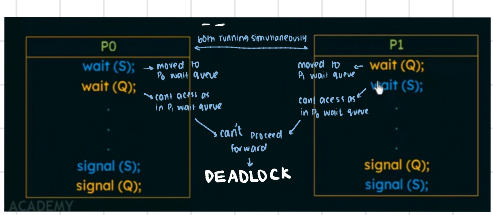
↳ **Starvation**

↳ when a process is postponed because it requires a resource to run, that is never allocated to this process

indivisibility

when 1 process modifies semaphore

no other process can simultaneously modify that semaphore value

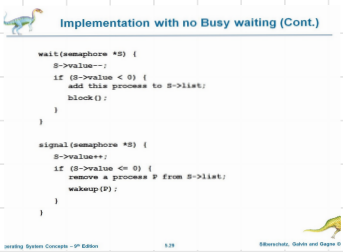


DEADLOCK VS STARVATION

↳ process stuck in circular waiting for the resources

↳ process waits for a resource indefinitely

deadlock implies starvation BUT starvation does not impy deadlock



POSIX - Semaphores

#include <semaphore.h>
Sem_t sem;
/* Create the semaphore and initialize it to 1 */
Sem_init(&sem, 0, 1);
The sem_init() function is passed three parameters:
1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

POSIX - Semaphores

/* acquire the semaphore */
Sem_wait(&sem);

/* critical section */

/* release the semaphore */
Sem_post(&sem);

Priority Inversion

↳ scheduling problem when lower priority process holds a lock needed by a higher priority process

SOLUTION →

Priority Inheritance Protocol

↳ when a job blocks a higher priority job it ignores the current lower priority job executes higher priority job's critical section then releases lock and returns to the current lower priority job

CLASSICAL PROBLEMS OF SYNCHRONIZATION

1. Bounded buffer Problem

↳ shared buffer

↳ buffer of n slots, each capable of storing one unit of data

↳ Producer process
- insert data in empty slot of buffer
- not insert data when buffer is full

↳ consumer process
- remove data from filled slot of buffer
- not remove data when buffer is empty

Solution using semaphores

↳ n buffers → each can hold one item

↳ m(mutex) → binary semaphore
- aquire lock
- release lock

↳ empty → counting semaphore → initial value = no of slots in buffer since initially all slots are empty

↳ full → counting semaphore → initial value =

Producer

```
do {
  wait (empty); // wait until empty>0
                   and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full); // increment 'full'
} while(TRUE)
```

Notes: wait → always decrements; empty>0 → empty slot available; acquire lock → consumer can't make any change; release lock → free to use buffer now; signal → always increments

Consumer

```
do {
  wait (full); // wait until full>0 and
                   then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty); // increment 'empty'
} while(TRUE)
```

Note: full>0 → there are slots that are filled

2. The Readers Writers Problem

↳ shared database

↳ Readers → only want to read

↳ Writers → want to read and write

↳ if a writer and a writer/reader acess database simultaneously → PROBLEM

↳ give exclusive acess to database → SOLUTION

Solution using semaphores

↳ mutex → a semaphore → initialize to 1

↳ wrt → a semaphore → initialized to 1

↳ readcount → counts how many readers reading same data → initialized to 0
- ↳ whenever modified wait(mutex) to be called

Writer Process

```
do {
/* writer requests for critical
section */
    wait(wrt);
  /* performs the write */
   // leaves the critical section
    signal(wrt);
} while(true);
```

Notes: wait(wrt) → no one other than wrt can make changes; if wrt free then give it acess; signal(wrt) → free to use now

Reader Process

```
do {
  wait (mutex);
 readcnt++; // The number of readers has now increased by 1
 if (readcnt==1)
    wait (wrt); // this ensure no writer can enter if there is even one reader
   signal (mutex); // other readers can enter while this current reader is
                      inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
     signal (wrt);        // writers can enter
     signal (mutex); // reader leaves
} while(true);
```

used in OS resource allocation → Processes → resources

3. The Dinning-Philosophers Problem

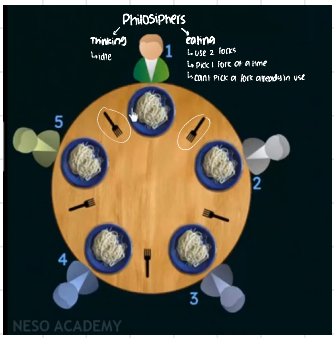
↳ 5 Philosophers, 5 forks

↳ Eating → use 2 forks for eating; Pick 1 fork at a time; can't pick if fork already in use — can't eat unless has 2 forks

↳ Thinking → idle

↳ when a Philosophers eat he uses 2 forks

↳ NO 2 adjacent Philosophers try to eat at the same time (as forks are limited) → SOLUTION



SOLUTION USING SEMAPHORES

↳ fork[5] (as 5 forks) → a binary semaphore array (an element initialized to 1; 1 → free, 0 → in use)

↳ grab fork → wait() → used on 2 forks

↳ release fork → signal()

```
The structure of philosopher i
do {
wait (chopstick [i] ); first chopstick
                       (for current)
wait(chopstick [ (i + 1) % 5] ) ; second chopstick
                       (for adjacent chopstick)
. . . .
// eat
signal(chopstick [i]);
signal(chopstick [(i + 1) % 5]);
// think
}while (TRUE);
```

can lead to Deadlock

all 5 hungry and grab left fork

all elements of fork = 0

when grabbing right fork, they will all be delayed forever

solutions to Deadlock

1. allow at most 4 philosophers (no of forks will remain 5)
2. allow to pick forks ONLY if both forks available
3. use asymmetric solution
   - ↳ an odd P first picks right fork then left fork
   - ↳ an even P first picks left fork then right fork

deadlock implies starvation

BUT

starvation does not imply deadlock