CPU Scheduling Algorithms

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called CPU scheduling

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources

- Maximum utilization of CPU so that we can keep the CPU as busy as possible.
- Throughput means the number of processes which are completing their execution in per unit time. There must be maximum throughput.
- Turnaround time means that the time taken by the processes to finish their implementation. It must be a minimum.
- Waiting time is that time for which the process remains in the ready queue. It must be a minimum.
- There must be fare allocation of CPU.
- Response time is the time when the process gives its first response. It must be a minimum.

Terminology

- Arrival time is the time at which the process arrives in the ready queue for execution, and it is given in our table when we need to calculate the average waiting time.
- Completion time is the time at which the process finishes its execution.
- Turnaround time is the difference between completion time and arrival time, i.e. turnaround time = Completion time- arrival time
- Burst time is the time required by the process for execution of the process by CPU.
- Waiting time (W.T) is the difference between turnaround time and burst time, i.e. waiting time= Turnaround time – Burst time

Types of Scheduling Algorithms

First Come First Serve (FCFS) Scheduling

In this scheduling, the process which arrives first in front of CPU will be executed first by the CPU. It is a non-preemptive type of scheduling algorithm, i.e. in this scheduling algorithm priority of processes does not matter, or you can say that whatever the priority of the process is, the process will be executed in the manner they arrived in front of the CPU.

Shortest Job First(SJF) scheduling algorithm

It is also called the Shortest Job Next (SJN) scheduling. It is both preemptive and non-preemptive. In this scheduling algorithm, the process which has the shortest burst time will be processed first by the CPU.

Shortest Remaining Time First (SRTF) scheduling algorithm

It is the preemptive mode of Shortest Job First (SJF) scheduling. In this algorithm, the process which has the short burst time is executed by the CPU. There is no need to have the same arrival time for all the processes. If another process was having the shortest burst time then the current process which is executing get stopped in between the execution, and the new arrival process will be executed first.

Priority based scheduling algorithm

This is the scheduling algorithm which is based on the priority of the processes. In priority scheduling, the scheduler itself chooses the priority of the task, and then they will be executed according to their preference which is assigned to them by the scheduler. The process which has the highest priority will be firstly processed by the CPU, and the process which has the lowest priority will be executed by the CPU when the current process terminates its execution.

Round Robin scheduling algorithm

Round robin scheduling is the preemptive scheduling algorithm. Here particular time slice is allocated to each process to which we can say as time quantum. Every process, which wants to execute itself, is present in the queue. CPU is assigned to the process for that time quantum. Now, if the process completed its execution in that quantum of time, then the process will get terminated, and if the process does not achieve its implementation, then the process will again be added to the ready queue, and the previous process will wait for its turn to complete its execution.

Highest Response Ratio Next (HRRN) scheduling algorithm

It is a non-preemptive scheduling algorithm which is done on the basis of a new parameter called Response Ratio (RR). We will calculate the response ratio of all the processes and the process which has the highest response ratio will be implemented first.
Response Ratio is given by
Response Ratio= (Waiting time+ Burst time)/ Burst time

Memory Management

Imp reads: https://tldp.org/LDP/tlk/mm/memory.html

Introduction

- The computer is able to change only that data which is present in the main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.
- All the programs are loaded in the main memory for execution

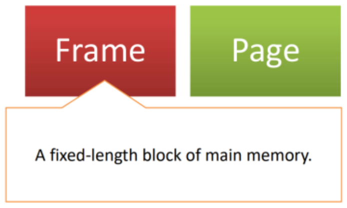
- Main memory is also known as RAM (Random Access Memory). This is volatile memory

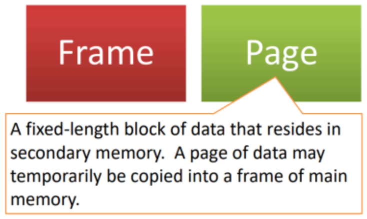
Memory Management is the process of coordinating and controlling the memory in a computer.

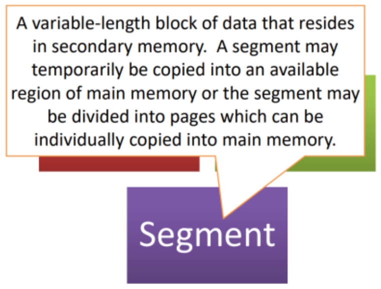
- This technique decides which process will get memory at what time.
- It also keeps the count of how much memory can be allocated to a process.
- It tracks when memory is freed or unallocated and updates the status.
- The memory management function keeps track of the status of each memory location, either allocated or free
- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time

Assignment ques - Need of memory management?

Frame, Page and Segments

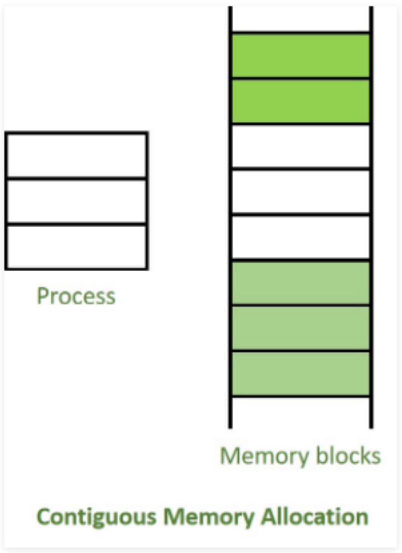



Frame
Page
A fixed-length block of data that resides in secondary memory. A page of data may temporarily be copied into a frame of main memory.


A variable-length block of data that resides in secondary memory. A segment may temporarily be copied into an available region of main memory or the segment may be divided into pages which can be individually copied into main memory.
Segment

Contiguous memory allocation

In contiguous memory allocation, when a process requests for the memory, a single contiguous section of memory blocks is assigned to the process according to its requirement.



One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process.

Hole – block of available memory; holes of various size are scattered throughout memory

When a process arrives, it is allocated memory from a hole large enough to accommodate it

Operating system maintains information about:

a) allocated partitions b) free partitions (hole)

First-fit

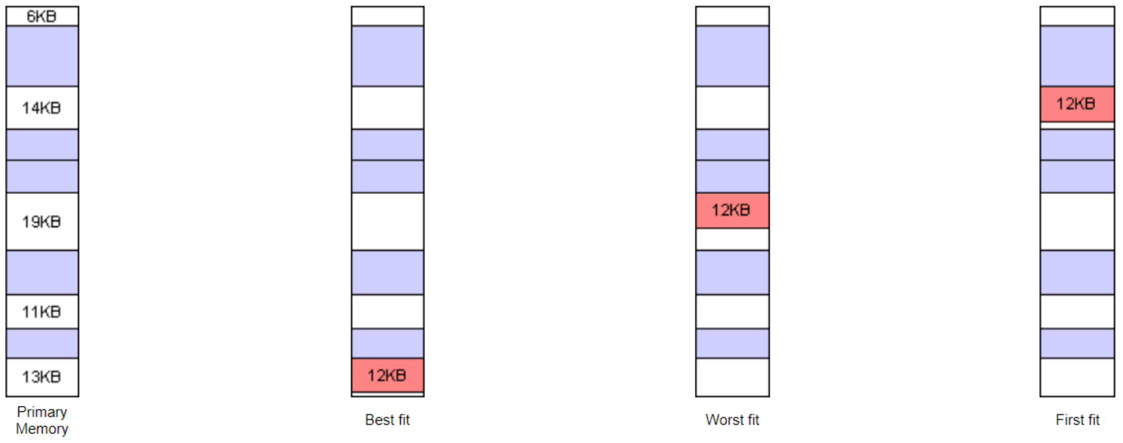
Allocate the first hole that is big enough

Best-fit

Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.

Worst-fit

Allocate the largest hole; must also search entire list. Produces the largest leftover hole



Note: First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Notice in the diagram above that the Best fit and First fit strategies both leave a tiny segment of memory unallocated just beyond the new process. Since the amount of memory is small, it is not likely that any new processes can be loaded here. This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as fragmentation. The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes

Assignment: Given free memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?

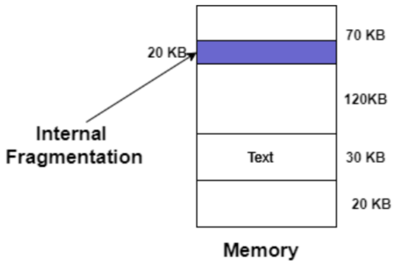
Fragmentation

A Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process.

It is of two types

- Internal fragmentation
- External fragmentation

Internal fragmentation



70 KB partition is used to load a process of 50 KB so the remaining 20 KB got wasted.

Suppose the size of the process is lesser than the size of the partition in that case some size of the partition gets wasted and remains unused. **This wastage inside the memory is generally termed as Internal fragmentation.**

External fragmentation

In external fragmentation, we have a free memory block, but we can not assign it to process because blocks are not contiguous.
Example:
Suppose there is a fixed partitioning is used for memory allocation and the different size of block 3MB, 6MB, and 7MB space in memory
three process p1, p2, p3 comes with size 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating process p1 process and p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we cannot assign it because free memory space is not contiguous. **This is called external fragmentation**

Compaction

Reduce external fragmentation by compaction
Shuffle memory contents to place all free memory together in one large block
Compaction is possible only if relocation is dynamic, and is done at execution time

Paging

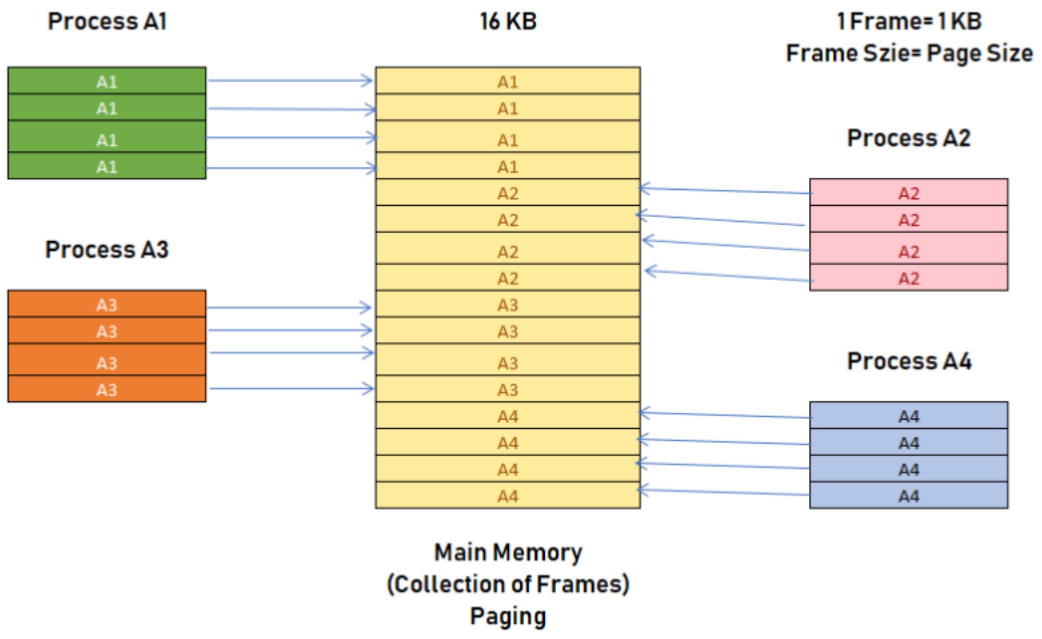
Paging is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages. In the Paging method, the main memory is divided into small fixed-size blocks of physical memory, which is called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation.

- Paging permits the **physical address space** of a process to be **non-contiguous.**
- Paging solves the problem of fitting memory chunks of varying sizes onto the backing store and this problem is suffered by many memory management schemes.
- Paging helps to **avoid external fragmentation** and the **need for compaction**.
- The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as the paging technique
- The Physical Address Space is conceptually divided into several fixed-size blocks, called frames.
- The Logical Address Space is also split into fixed-size blocks, called pages.
- Page Size = Frame Size

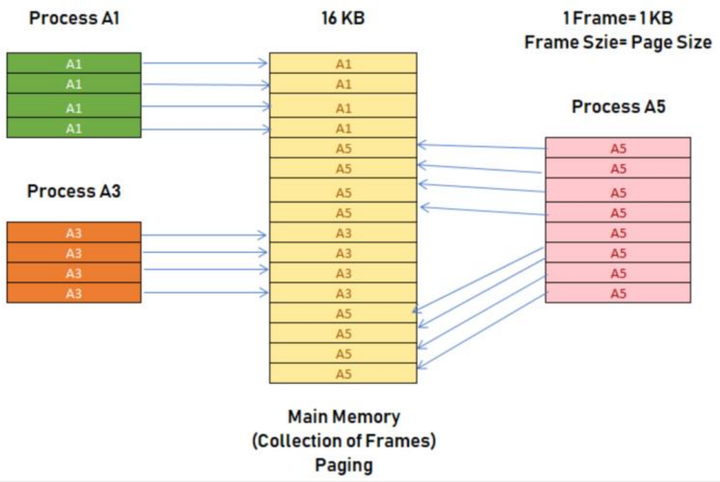
Example

For example, if the main memory size is 16 KB and Frame size is 1 KB. Here, the main memory will be divided into the collection of 16 frames of 1 KB each.

There are 4 separate processes in the system that is A1, A2, A3, and A4 of 4 KB each. Here, all the processes are divided into pages of 1 KB each so that operating system can store one page in one frame.



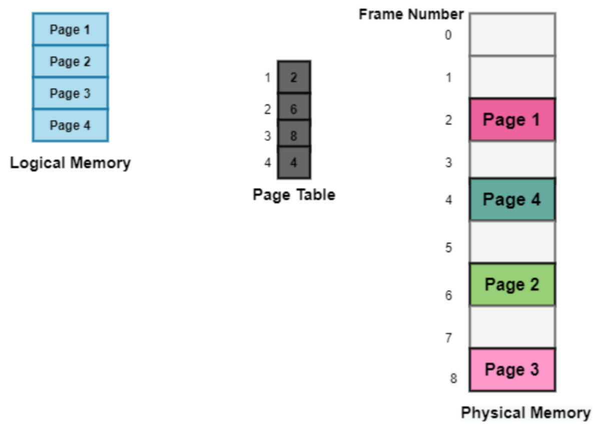
Let's say A2 and A4 are moved to the waiting state after some time and The process A5 of size 8 pages (8 KB) are waiting in the ready queue.



Here, eight non-contiguous frames which is available in the memory, and paging offers the flexibility of storing the process at the different places

Page table

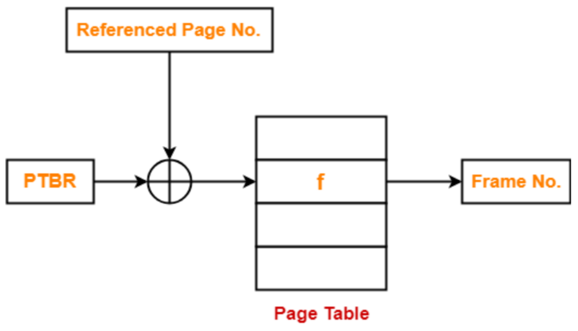
Page table mainly provides the corresponding frame number (base address of the frame) where that page is stored in the main memory i.e It maps the page number referenced by the CPU to the frame number where that page is stored.



Some of the characteristics of the Page Table are as follows:

- It is stored in the main memory.
- Generally, the Number of entries in the page table = the Number of Pages in which the process is divided.
- Page Table Base Register (PTBR) provides the base address of the page table
- Each process has its own independent page table.

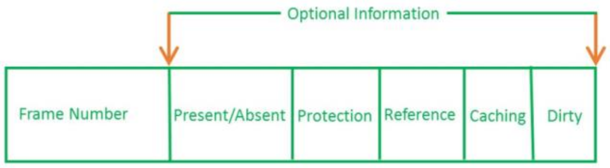
Working



Obtaining Frame Number Using Page Table

- Page Table Base Register (PTBR) provides the base address of the page table.
- The base address of the page table is added with the page number referenced by the CPU.
- It gives the entry of the page table containing the frame number where the referenced page is stored

This is how a PTE looks like:



PAGE TABLE ENTRY

This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit etc)

Reasons for not having faster access incase of larger memory size:

1. To find the frame number

2. To go to the address specified by frame number

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB)

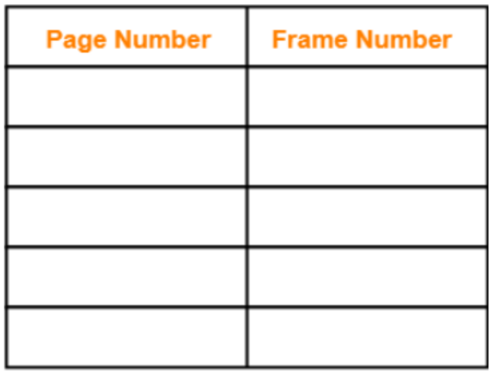
Translation Lookaside Buffer

Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used

Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss)

Translation Lookaside Buffer (TLB) consists of two columns-

1. Page Number
2. Frame Number

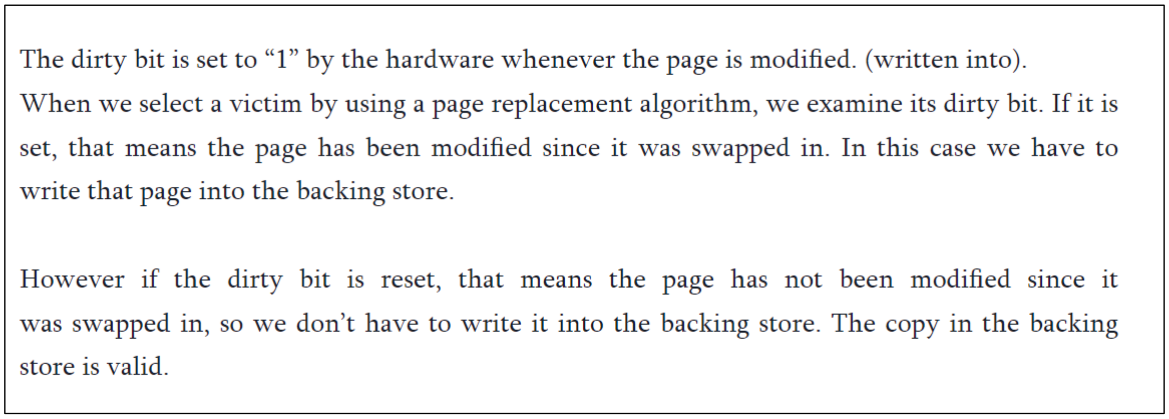


Page Number	Frame Number

Translation Lookaside Buffer

Dirty Bit

A dirty bit is a bit in memory switched on when an update is made to a page by computer hardware. When the dirty bit is switched on, the page is modified and can be replaced in memory. If it is off, no replacement is necessary since no updates have been made.



The dirty bit is set to "1" by the hardware whenever the page is modified. (written into).
When we select a victim by using a page replacement algorithm, we examine its dirty bit. If it is set, that means the page has been modified since it was swapped in. In this case we have to write that page into the backing store.

However if the dirty bit is reset, that means the page has not been modified since it was swapped in, so we don't have to write it into the backing store. The copy in the backing store is valid.

Ques of CCEE

Dirty bit is used to indicate which of the following?

A. A page fault has occurred
B. A page has corrupted data
C. A page has been modified after being loaded into cache
D. An illegal access of page

CPU throttling

Adjusting the clock speed of the CPU. Also called "dynamic frequency scaling," CPU throttling is commonly used to automatically slow down the computer when possible to use less energy and conserve battery, especially in laptops. CPU throttling can also be adjusted manually to make the system quieter, because the fan can then run slower.

Processor throttling is also known as "automatic underclocking". Automatic overclocking (boosting) is also technically a form of dynamic frequency scaling, but it's relatively new and usually not discussed with throttling.

Dynamic frequency scaling reduces the number of instructions a processor can issue in a given amount of time, thus reducing performance. Hence, it is generally used when the workload is not CPU-bound.

Reentrant Function

A computer program or routine is described as reentrant if it can be safely called again before its previous invocation has been completed (i.e it can be safely executed concurrently).

Conditions for a reentrant function or code

- It may not use global and static data
- It should not call another non-reentrant function but it can call any reentrant function.
- It should not modify it’s own code

// A non-reentrant example

```
// [The function depends on global variable i]

int i;

// Both fun1() and fun2() are not reentrant

// fun1() is NOT reentrant because it uses global variable i

int fun1()

{

        return i * 5;

}

// fun2() is NOT reentrant because it calls a non-reentrant

// function

int fun2()

{

return fun1() * 5;

}
```

// Both fun1() and fun2() are reentrant

```
int fun1(int i)

{

        return i * 5;

}

int fun2(int i)

{

return fun1(i) * 5;

}
```

Inter Process Communication

Most modern computer systems use the notion of a process that lets us execute multiple tasks at any time. And, as multiple processes execute at the same time, often they need to communicate with each other for various reasons.

Processes executing concurrently in a computer system are of two types –

- independent processes
- cooperating processes

A process is independent if it cannot affect or be affected by any other process executing in the computer system (does not share data with any other process)

A process is cooperating if it can affect or be affected by other processes executing in the computer system (shares data with other processes is a cooperating process)

The cooperating processes need to communicate with each other to exchange data and information. **Inter-process communication is the mechanism of communicating between processes**

Modes of IPC

There are two modes through which processes can communicate with each other –

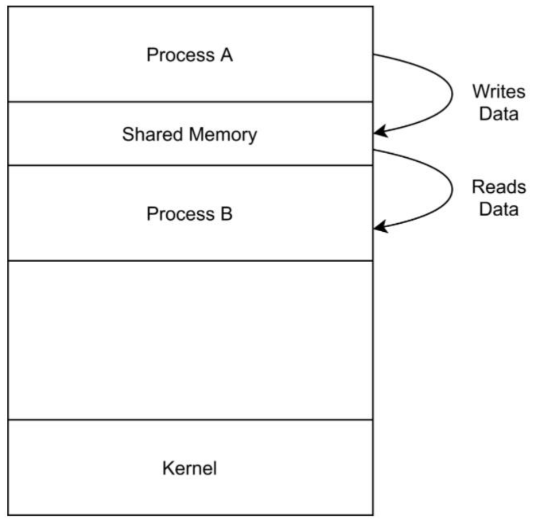
- shared memory
- message passing

As the name suggests, the shared memory region shares a shared memory between the processes.

On the other hand, the message passing lets processes exchange information through messages.

Shared Memory

Interprocess communication through the shared memory model requires communicating processes to establish a shared memory region. In general, the process that wants to communicate creates the shared memory region in its own address space. Other processes that wish to communicate to this process need to attach their address space to this shared memory segment:



What did we learn?

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
- Fast
- Limitation: Error prone. Needs synchronization between processes

Shared memory functions used for IPC

int shmget (key, size, flags)

- Create a shared memory segment;
- Returns ID of segment : shmid
- key : unique identifier of the shared memory segment
- size : size of the shared memory (rounded up to the PAGE_SIZE)

int shmat (shmid, addr, flags)

- Attach shmid shared memory to address space of the calling process
- addr : pointer to the shared memory address space

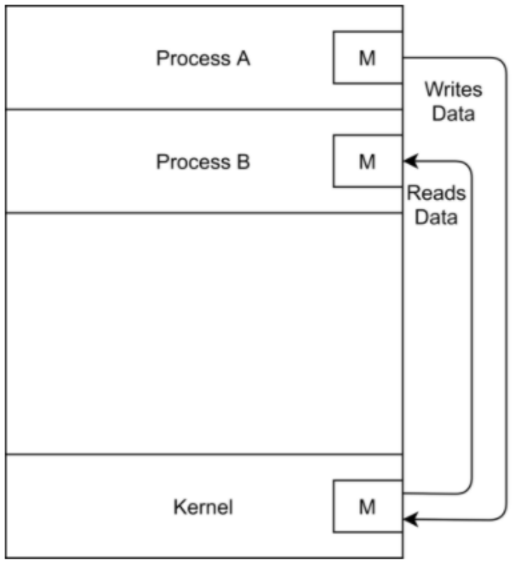
int shmdt (shmid)

- Detach shared memory

Message Passing

In this mode, processes interact with each other through messages with assistance from the underlying operating system i.e via kernel.

Here two processes A, and B are communicating with each other through message passing. Process A sends a message M to the operating system (kernel). This message is then read by process B.



Other modes of IPC

- signals
- FIFOS (named pipes)
- Pipes

Signals

Signals allow for asynchronous communication between processes, but of a very limited nature. A signal communicates that an event has occurred, but nothing else. There's no opportunity for sending additional data concerning the event

Pipes

Pipes – direct the outstream of one process to feed the input of another process.

IPC is a very common mechanism in Linux and Pipe maybe one of the most widely used IPC methods. When you type cat file | grep bar, you create a pipe to connect stdout of cat to stdin of grep

A pipe, as its name states, can be understood as a channel with two ends.

Pipe is actually implemented using a piece of kernel memory. The system call pipe always create a pipe and two associated file descriptions, fd[0] for reading from the pipe and fd[1] for writing to the pipe.

Synatx in C for pipe system call – pipe()

int pipe(int fds[2]);

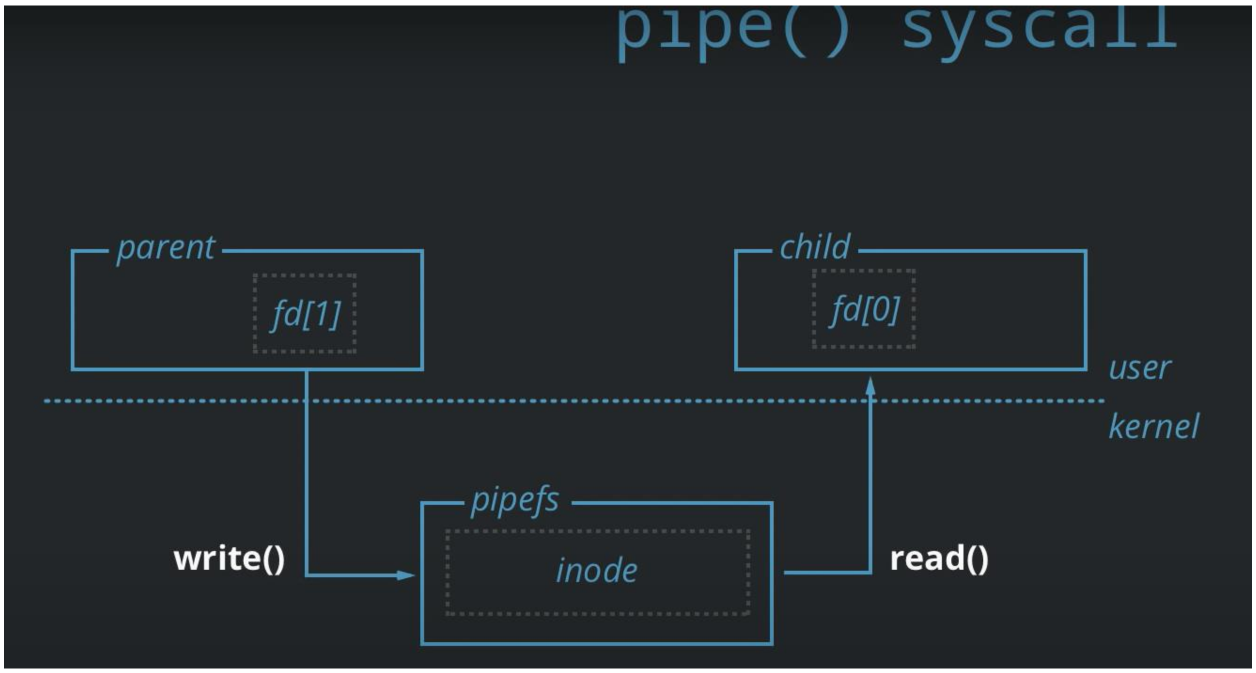
Parameters:

- fd[0] will be the fd(file descriptor) for the read end of pipe. So, fd[0] is argument of read() system call
- fd[1] will be the fd(file descriptor) for the write end of pipe. So, fd[1] is argument of write() system call
- These file descriptors are used for read and write operations
- Returns: 0 on Success
- -1 on error – to know the cause of error or failure – we can use perror() system call

After the pipe system call executes, the array pipefd [2] contains two file descriptors, pipefd [0] is for reading from the pipe and pipefd [1] is for writing to the pipe.

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

pipe() syscall to shows parent writes and child reads



FIFO Named Pipe

A *FIFO special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo.

The mkfifo function is declared in the header file sys/stat.h.

Function: int mkfifo (const char *filename, mode_t mode)

The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the file's permissions

The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned.

In addition to the usual file name errors, the following error conditions are defined for this function:

EEXIST

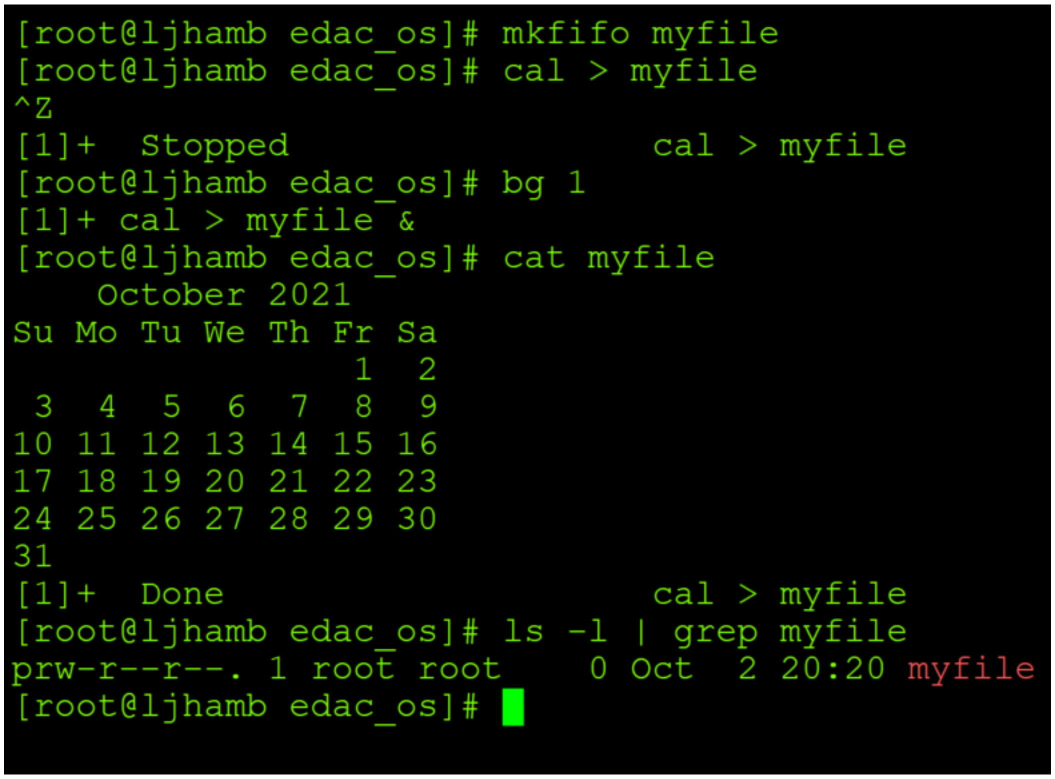
The named file already exists.

ENOSPC

The directory or file system cannot be extended.

EROFS

The directory that would contain the file resides on a read-only file system.



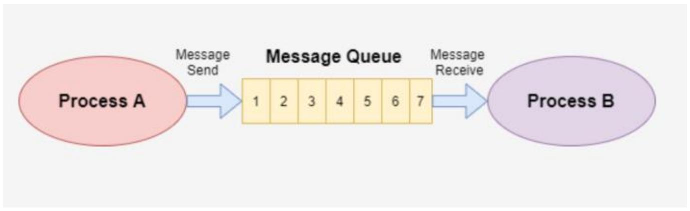
```
[root@ljhamb edac_os]# mkfifo myfile
[root@ljhamb edac_os]# cal > myfile
^Z
[1]+  Stopped                 cal > myfile
[root@ljhamb edac_os]# bg 1
[1]+ cal > myfile &
[root@ljhamb edac_os]# cat myfile
    October 2021
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
[1]+  Done                    cal > myfile
[root@ljhamb edac_os]# ls -l | grep myfile
prw-r--r--. 1 root root    0 Oct  2 20:20 myfile
[root@ljhamb edac_os]#
```

Message queues

Message queues are one of the inter-process communication mechanism which allows processes to exchange data in the form of messages between two processes. It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.

IPC using message queue involves the following steps:

- Defining a Message Structure
- Creating the Message Queue
- Sending the Message to the Process A through the Message Queue
- Receiving the Message from the Process A through the Message Queue



Each message is given an identification or "type" so that processes can select the appropriate message.

Process must share a common "key" in order to gain access to the queue in the first place.

System calls used for message queues

- ftok(): used to generate a unique key
- msgget() – creating message using a queue
- msgsnd() - sending a message to a queue
- msgrcv() – fetching a message from a queue
- msgctl() - controlling a queue

// Program for Message Queue (Writer Process) -- client.c

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

 // message queue structure

struct mesg_buffer {

   long mesg_type;

   char mesg_text[100];

} message;

 int main()

{

   key_t key;

   int msgid;

     // generate unique key

   key = ftok("somefile", 65);

     // create a message queue and return identifier

   msgid = msgget(key, 0666 | IPC_CREAT);

   message.mesg_type = 1;

     printf("Insert message  : ");

   gets(message.mesg_text);

     // send message

   msgsnd(msgid, &message, sizeof(message), 0);

     // display the message

   printf("Message sent to server : %s\n", message.mesg_text);

     return 0;

}
```

//Program for Message Queue (Reader Process) - server.c

```
#include <stdio.h>
```

```
#include <sys/ipc.h>

#include <sys/msg.h>

  // structure for message queue

struct mesg_buffer {

    long mesg_type;

    char mesg_text[100];

} message;

  int main()

{

    key_t key;

    int msgid;

      // generate unique key

    key = ftok("somefile", 65);

      // create a message queue and return identifier

    msgid = msgget(key, 0666 | IPC_CREAT);

        printf("Waiting for a message from client...\n");

    // receive message

    msgrcv(msgid, &message, sizeof(message), 1, 0);

      // display the message

    printf("Message received from client : %s\n",message.mesg_text);

      // to destroy the message queue

    msgctl(msgid, IPC_RMID, NULL);

      return 0;

}
```

Now task is to compile and run ./client and enter message to be sent to server and then run ./server and observe the output

Semaphores

To avoid a series of issues caused by multiple processes (or programs) accessing a shared resource at the same time, we need a method that can be authorized by generating and using a token, so that only one execution thread can access the critical section of the code at any given time. The critical section is a code segment where the shared variables can be accessed, and the atomic action is required in this section.

Binary Semaphore (0 or 1) aka Mutexes

A semaphore is implemented as an integer variable with atomic increment and decrement operations; so long as the value is not negative the thread will continue, it will block otherwise.

- The increment operation is called V, or signal or wake-up or up
- the decrement is called P, or wait or sleep or down

wait(): decreases the counter by one; if the counter is negative, then it puts the thread on a queue and blocks.

signal(): increments the counter; wakes up one waiting process.

The definition of wait operation is as follows:

```
wait(s)

{

   while (s==0); //no operation

   s=s-1;

}
```

The definition of signal operation is as follows:

```
signal(s)

{

s=s+1;

}
```

Note: Initial value of semaphore variable is 1

Example: The critical section of Process P is in between P and V operation.

Process

Code //Initial code of process

P(s); //here if value of s=0, then code will get stuck, else if it is 1(initial value), then it will be decremented to 0, and process will enter the critical section of code

CS; //critical section of the code

V(s); //after execution of critical section, the value 0 will again be incremented and process exits the critical section of the code and moves to execute the remaining section of code

Remaining Section of the code //noncritical section of code

P1	S=1	Executing noncritical section of the code
P2		Executing noncritical section of the code
P1	S=0	Enters the critical section
P2		Executing noncritical section of the code
P1	S=0	Executing the critical section of the code
P2		Wants to enter critical section but can't because s=0
P1	S=1	Exits the critical section and increments s=1
P2		Enters the critical section and updates the s=0

It shows mutual exclusion. Here we have two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.

What did we learn?

Mutually exclusive because at one time only one process can execute the critical section of code

Counting Semaphore

A binary semaphore is restricted to values of zero or one, while a counting semaphore can assume any nonnegative integer value.

A counting semaphore has two components-

- An integer value
- An associated waiting list (usually a queue)

The value of counting semaphore may be positive or negative.

- Positive value indicates the number of processes that can be present in the critical section at the same time.
- Negative value indicates the number of processes that are blocked in the waiting list.

- The waiting list of counting semaphore contains the processes that got blocked when trying to enter the critical section.
- In waiting list, the blocked processes are put to sleep.
- The waiting list is usually implemented using a queue data structure.
- Using a queue as waiting list ensures bounded waiting.
- This is because the process which arrives first in the waiting queue gets the chance to enter the critical section first.

- The wait operation is executed when a process tries to enter the critical section.
- Wait operation decrements the value of counting semaphore by 1.
- Then, following two cases are possible-

<u>Counting Semaphore Value >= 0</u>

- If the resulting value of counting semaphore is greater than or equal to 0, process is allowed to enter the critical section.

<u>Counting Semaphore Value < 0</u>

- If the resulting value of counting semaphore is less than 0, process is not allowed to enter the critical section.
- In this case, process is put to sleep in the waiting list.

- The signal operation is executed when a process takes exit from the critical section.
- Signal operation increments the value of counting semaphore by 1.
- Then, following two cases are possible-

<u>Counting Semaphore <= 0</u>

- If the resulting value of counting semaphore is less than or equal to 0, it picks the process from blocked state and puts it in ready queue or we can say that - a process is chosen from the waiting list and wake up to execute.

<u>Counting Semaphore > 0</u>

- If the resulting value of counting semaphore is greater than 0, no action is taken.

- By adjusting the value of counting semaphore, the number of processes that can enter the critical section can be adjusted.
- If the value of counting semaphore is initialized with N, then maximum N processes can be present in the critical section at any given time.

What did we learn?

- Consider n units of a particular non-shareable resource are available.
- Then, n processes can use these n units at the same time.
- So, the access to these units is kept in the critical section.
- The value of counting semaphore is initialized with ‘n’.
- When a process enters the critical section, the value of counting semaphore decrements by 1.
- When a process exits the critical section, the value of counting semaphore increments by 1.

Virtual Memory

Virtual memory is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of data from random access memory to disk storage.

This means that when RAM runs low, virtual memory can move data from it to a space called a paging file. This process allows for RAM to be freed up so that a computer can complete the task.

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.

Example

Let's assume that an OS requires 300 MB of memory to store all the running programs. However, there's currently only 50 MB of available physical memory stored on the RAM.

The OS will then set up 250 MB of virtual memory and use a program called the Virtual Memory Manager(VMM) to manage that 250 MB.

So, in this case, the VMM will create a file on the hard disk that is 250 MB in size to store extra memory that is required.

The OS will now proceed to address memory as it considers 300 MB of real memory stored in the RAM, even if only 50 MB space is available.

It is the job of the VMM to manage 300 MB memory even if just 50 MB of real memory space is available

Swapping is the process the OS uses to move data between RAM and virtual memory

Demand Paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.

A page is copied to the main memory when its demand is made or page fault occurs.

Hence, A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.

What is a Page Fault?

Page fault is more like an error. It mainly occurs when any program tries to access the data or the code that is in the address space of the program, but that data is not currently located in the RAM of the system.

So basically when the page referenced by the CPU is not found in the main memory then the situation is termed as Page Fault.

Whenever any page fault occurs, then the required page has to be fetched from the secondary memory into the main memory.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated

First In First Out (FIFO)

In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example: Page Reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find number of page faults.

Answer 6

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Bélády's anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

Try it out:

Try to find the number of page faults when:

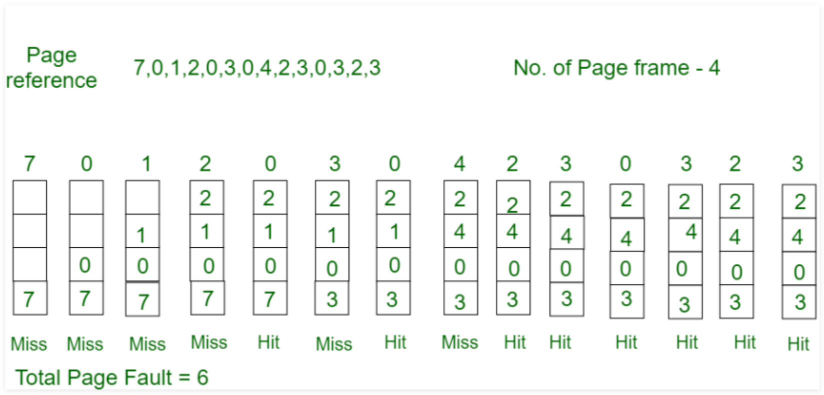
Reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 has 3 slots

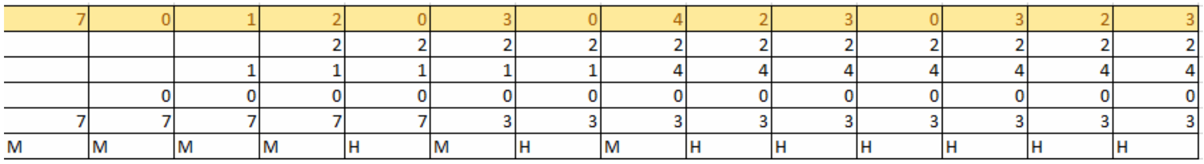
Same reference string with 4 slots

Optimal Page replacement

Here pages are replaced which would not be used for the longest duration of time in the future.

Try page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,3 with 4 page frame



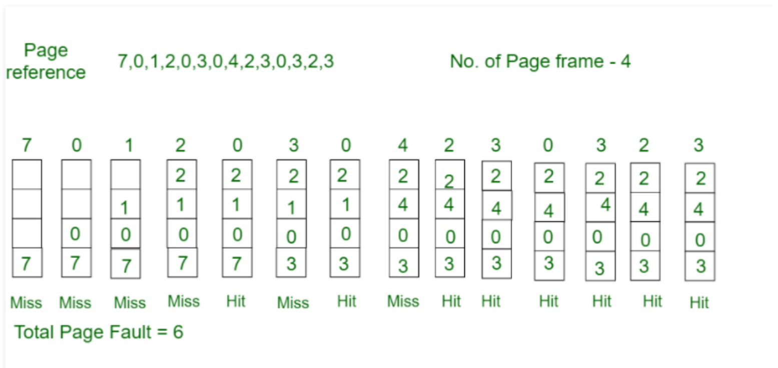


7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
M	M	M	M	H	M	H	M	H	H	H	H	H	H

Least Recently Used

In this algorithm page will be replaced which is least recently used.

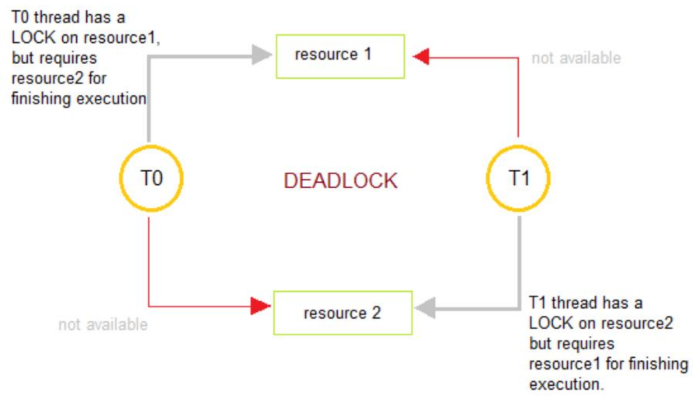
Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.



Here LRU has same number of page fault as optimal but it may differ according to question.

Deadlocks

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

T0 and T1 are in a deadlock because each of them needs the resource of others to complete their execution but neither of them is willing to give up their resources

Normal mode of Operation utilization of resources by a process is in the following sequence

- Request: Firstly, the process requests the resource.
- Use: The Process can operate on the resource
- Release: The Process releases the resource.

Necessary Conditions that must hold for a deadlock

Mutual Exclusion

According to this condition, atleast one resource should be non-shareable (non-shareable resources are those that can be used by one process at a time.)

Hold and wait

According to this condition, A process is holding atleast one resource and is waiting for additional resources.

No preemption

Resources cannot be taken from the process because resources can be released only voluntarily by the process holding them.

Circular wait

In this condition, the set of processes are waiting for each other in the circular form.

Handling Deadlocks

Ignore

According to this method, it is assumed that deadlock would never occur.

Avoid

This method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state. This method checks every step performed by the operating system. Any process continues its execution until the system is in a safe state. Once the system enters into an unsafe state, the operating system has to take a step back.

This concept is more comfortable for single user system because they use their system for simply browsing as well as other simple activities.

Prevent

The main aim of the deadlock prevention method is to violate any one condition among the four; because if any of one condition is violated then the problem of deadlock will never occur.

Starvation vs Deadlock

Starvation	**Deadlock**
When all the low priority processes got blocked, while the high priority processes execute then this situation is termed as Starvation.	Deadlock is a situation that occurs when one of the processes got blocked.
Starvation is a long waiting but it is not an infinite process.	Deadlock is an infinite process.
It is not necessary that every starvation is a deadlock.	There is starvation in every deadlock.
Starvation is due to uncontrolled priority and resource management.	During deadlock, preemption and circular wait does not occur simultaneously.

Deadlock Simulator Program

```
#include <iostream>

using namespace std;

//Description: This program simulates the execution of 3 processes that will forever be stuck in a deadlock. Each process is looking for a particular person, and each process also can release a person another process is looking for.

//This program is set up in such a way in which none of the processes will be able to ever find the person that another process could release.

int user1 = 0; //Represents user1

int user2 = 0; //Represents user2

int user3 = 0; //Represents user3

int found_someone = 0; //Used to determine if any of the processes have found the user or not

//Release user1, allowing a process to find him

void release_user1()

{

        user1 = 1; //Set user1 to found

}

//Look for user1

void find_user1()

{

        //If user1 is found: Print that he was caught

        //Else: Ask for user1
```

```
	if(user1)
	{
		cout<<"Caught user1"<<endl;
	}
	else
	{
		cout<<"Where is user1"<<endl;
	}
}
//Release user2, allowing a process to find him
void release_user2()
{
	user2 = 1; //Set user2 to found
}
//Look for user2
void find_user2()
{
	//If user2 is found: Print that he was found
	//Else: Ask where user2 is
	if(user2)
	{
		cout<<"Caught user2"<<endl;
	}
	else
	{
		cout<<"Where is user2?"<<endl;
	}
}
//Release user3, allowing a process to find her
void release_user3()
{
	user3 = 1;
}

//Look for user3
```

```
void find_user3()
{
	//If user3 is found: Print that she was found
	//Else: Ask where in the world she is
	if(user3)
	{
		cout<<"caught user3"<<endl;
	}
	else
	{
		cout<<"Where is user3?"<<endl;
	}
}

//Process looking for user1
void process1()
{
	find_user1();
	
	//If user1 is found: Release user2, allowing another process to find him
	if(user1)
	{
		release_user2();
		found_someone = 1;
	}
}
//Process looking for user2
void process2()
{
	find_user2();
	
	//If user2 is found: Release user3, allowing another process to find her
	if(user2)
	{
		release_user3();
```

```
                found_someone = 1;
        }
}
//Process looking for user3
void process3()
{
        find_user3();
        
        //If user3 is found: Release user1, allowing another process to find him
        if(user3)
        {
                release_user1();
                found_someone = 1;
        }
}
int main()
{
        while(!found_someone)
        {
                process1();
                process2();
                process3();
        }
}
```

Producer Consumer Problem

The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Prepared By:

Lavish Jhamb

Solutions Architect, Compliance Solutions

Qualys, Inc.