Test-2 Set-1

12. A. How do condition variables avoid the lost wakeup problem? Give pseudo code to producer consumer problem using condition variables with multiple producer threads and single consumer thread. ***6Marks CO4***

Answer Key-

The Producer/Consumer Problem **[Marking-2M]**

This problem is one of the small collection of standard, well-known problems in concurrent programming: a finite-size buffer and two classes of threads, producers and consumers, put items into the buffer (producers) and take items out of the buffer (consumers).

A producer must wait until the buffer has space before it can put something in, and a consumer must wait until something is in the buffer before it can take something out.

A condition variable represents a queue of threads waiting for some condition to be signalled.

Example here, has two such queues, one (*less*) for producers waiting for a slot in the buffer, and the other (*more*) for consumers waiting for a buffer slot containing information. The example also has a mutex, as the data structure describing the buffer must be accessed by only one thread at a time.

The Producer/Consumer Problem and Condition Variables [Marking-4M]

typedef struct {

```
    char buf[BSIZE];
    int occupied;
    int nextin;
    int nextout;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} buffer_t;

buffer_t buffer;
```

As shown, the producer thread acquires the mutex protecting the `buffer` data structure and then makes certain that space is available for the item being produced. If not, it calls **`pthread_cond_wait()`**, which causes it to join the queue of threads waiting for the condition *less*, representing **there is room in the buffer**, to be signaled.

At the same time, as part of the call to **`pthread_cond_wait()`**, the thread releases its lock on the mutex. The waiting producer threads depend on consumer threads to signal when the condition is true. When the condition is signaled, the first thread waiting on *less* is awakened. However, before the thread can return from **`pthread_cond_wait()`**, it must acquire the lock on the mutex again.

This ensures that it again has mutually exclusive access to the buffer data structure. The thread then must check that there really is room available in the buffer; if so, it puts its item into the next available slot.

At the same time, consumer threads might be waiting for items to appear in the buffer. These threads are waiting on the condition variable *more*. A producer thread, having just deposited something in the buffer, calls **`pthread_cond_signal ()`** to wake up the next waiting consumer. (If there are no waiting consumers, this call has no effect.)

Finally, the producer thread unlocks the mutex, allowing other threads to operate on the buffer data structure.

The Producer/Consumer Problem--the Producer

void producer(buffer_t *b, char item)

```
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock (&b->mutex);
}
```

Note the use of the **`assert ()`** statement; unless the code is compiled with `NDEBUG` defined, **`assert ()`** does nothing when its argument evaluates to true (that is, nonzero), but causes the program to abort if the argument evaluates to false (zero). Such assertions are especially useful in multithreaded programs--they immediately point out runtime problems if they fail, and they have the additional effect of being useful comments.

The comment that begins `/* now: either b->occupied ...` could better be expressed as an assertion, but it is too complicated as a Boolean-valued expression and so is given in English.

Both the assertion and the comments are examples of invariants. These are logical statements that should not be falsified by the execution of the program, except during brief moments when a thread is

modifying some of the program variables mentioned in the invariant. (An assertion, of course, should be true whenever any thread executes it.)

Using invariants is an extremely useful technique. Even if they are not stated in the program text, think in terms of invariants when you analyze a program.

The invariant in the producer code that is expressed as a comment is always true whenever a thread is in the part of the code where the comment appears. If you move this comment to just after the **`mutex_unlock()`**, this does not necessarily remain true. If you move this comment to just after the **`assert()`**, this is still true.

The point is that this invariant expresses a property that is true at all times, except when either a producer or a consumer is changing the state of the buffer. While a thread is operating on the buffer (under the protection of a mutex), it might temporarily falsify the invariant. However, once the thread is finished, the invariant should be true again.

The code for the consumer. Its flow is symmetric with that of the producer.

The Producer/Consumer Problem--the Consumer

char consumer(buffer_t *b)

```
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);

    return(item);
}
```

---

12. B. i. If Resource 1 has one slot, it is represented by R1 -> 1.Given the following scenarios, determine if there is a dead lock by drawing a resource allocation graph. R1 -> 2, R2 -> 2, P1 holds R2 requesting R1, P2 holds R1, P3 holds R1 requesting R2, P4 holds R2. ii. Considering a system with five processes P1 through P5 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken: 1. what will be the content of the need matrix? 2. Is the system in a safe state? If yes then what is the safe sequence? 3. What will happen if process p3 requests one additional instance of resource type C and two instances of resource type A? ***6.5Marks CO4***

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	2	0	2	6	5	3	1	0	1
P2	0	1	1	2	2	1			
P3	2	0	0	2	0	1			
P4	3	3	1	4	3	3			
P5	2	1	2	4	2	2			

Applying the Safety algorithm on the given system,

Step 1: Initialization Work = Available i.e. Work =3 3 2 ……P1……P2……..P3……..P4……P5….. Finish = | false | false | false | false | false |

Step 2: For i=0 Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(3 3 2) ⸝ false So P0 must wait. Step 2: For i=1 Finish [P1] = false and Need[P1]<=Work i.e. (1 2 2)<=(3 3 2) ⸝ true So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(3 3 2)+(2 0 0)=(5 3 2) …….P0……P1…….P2…….P3… ...P4…… Finish = | false | true | false | false | false |

Step 2: For i=2 Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0)<=(5 3 2) ⸝ false So P2 must wait. Step 2: For i=3 Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) ⸝ true So P3 must be kept in safe sequence. Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(1 0 1) ……P1……P2……P3……..P4…….P5….

Finish = | false | true | false | true | false |

Test-2 Set-2

12. Answer the following ***12.5Marks CO4***

12. A. What issues must a programmer be concerned with in choosing an address to attach a shared memory region to? What errors would the operating system protect against? Why does the shared memory IPC structure acquire a reference to the non_map for the segment? Illustrate shared memory data structures by giving algorithm for shmget (). ***6.5Marks CO4***

Answer Key-

[Marking Scheme – 2.5 M]
Inter Process Communication through shared memory is a concept where two or more process can access the common memory. Communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, FIFO and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

The following system calls are used:-

- **ftok()**: is use to generate a unique key.
- **shmget()**: int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.
- **shmat()**: Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg); shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.
- **shmdt()**: When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);
- **shmctl()**: when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);

shmget.c [Marking Scheme 4 M]

```
/ * shmget.c: Illustrate the shmget() function.

  * This is a simple exerciser of the shmget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

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

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

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to be passed to shmget() */
 int  shmflg;   /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);

 /* Get the shmflg value. */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

12.B. i. If Resource 1 has one slot, it is represented by R1 -> 1.Given the following scenarios, determine if there is a dead lock by drawing a resource allocation graph. R1 -> 1, R2 -> 2, R3 ->4, P1 holds R2 requesting R1, P2 holds (R2 + R1) requesting R3, P3 holds R3. ii. Considering a system with five processes P1 through P5 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken: 1.what will be the content of the need matrix? 2. Is the system in a safe state? if yes then what is the safe sequence? 3. what will happen if process p1 requests one additional instance of resource type C and two instances of resource type A? ***6Marks CO4***

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	1	4	2	6	5	3	9	0	0
P2	0	0	1	3	2	1			
P3	0	0	1	8	0	2			
P4	0	1	1	2	2	3			
P5	0	0	2	4	3	4			

• To determine whether this new system state is safe, we again execute Safety algorithm. Step 1: Initialization Here, m=3, n=5 Work = Available i.e. Work =9 0 0 …..P0……P1……P2…….P3…….P4…. Finish = | false | false | false | false | false |

Step 2: For i=0 Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(2 3 0) ⸌ false So P0 must wait. Step 2: For i=1 Finish[P1] = false and Need[P1]<=Work i.e. (0 2 0)<=(2 3 0) ⸌ true So P1 must be kept in safe sequence. Step 3: Work = Work + Allocation[P1] =(2 3 0)+(3 0 2)=(5 3 2) ……P3……P1…….P2……P5…….P4…… Finish = | false | true | false | false | false | Step 2: For i=2 Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(5 3 2) ⸌ false So P2 must wait. Step 2: For i=3 Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) ⸌ true So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3) …..P3…….P1……P2……..P3…….P4…... Finish = | false | true | false | true | false | Step 2: For i=4 Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) ⸌ true So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5) ……P2……P1……P2……P3…….P4…. Finish = | false | true | false | true | true | Step 2: For i=0 Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) ⸌ true So P0 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5) ……P1…….P2……P3……P4 ….. P5…. Finish = | true | true | false | true | true | Step 2: For i=2 Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) ⸌ true So P2 must be kept in safe sequence. Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7) …..P1…….P2……P3…….P4…….P5…. Finish= | true | true | true | true | true | Step 4:

Finish[Pi] = true for 0<=i<=4 Hence, the system is in a safe state. The safe sequence is. Conclusion: Since the system is in safe sate, the request can be granted

****

11.A. Compare the IPC functionality provided by pipes and message queues. What are the advantages and drawbacks of each? When is one more suitable than the other? Illustrate message queue data structures by giving algorithm for msgget() 6 Marks

Pipes	Message Queues are:
Are a layer over message Queues <--- Unidirectional!	UNIDIRECTIONAL
Have a maximum number of elements and each element has maximum size	Fixed number of entries, Each entry has a maximum size
is NOT A STREAMING INTERFACE. Datagram semantics, just list message Queues	All the queue memory (# entries * entry size) allocated at creation
On read, WILL PEND until there is data to read	Datagram-like behavior: reading an entry removes it from the queue. If you don't read the entire data, the rest is lost. For example: send a 20 byte message, but the receiver reads 10 bytes. The remaining 10 bytes are lost.
On write, WILL PEND until there is space in the underlying message queue	Task can only pend on a single queue using msqQReceive (there are ways to change that with alternative API)
Can use select facility to wait on multiple pipes	When sending, you will pend if the queue is full (and you don't do NO_WAIT) When receiving, you will pend if the queue is empty (and you don't do NO_WAIT) Timeouts are supported on receive and send

The biggest practical difference is that a pipe doesn't have the notion of "messages", it's just a pipe to write() bytes to and read() bytes from. The receiving end must have a way to know what piece of data constitute a "message" in your program, and you must implement that yourself. Furthermore the order of bytes is defined: bytes will come out in the order you put them in. And, generally speaking, it has one input and one output.

A message queue is used to transfer "messages", which have a type and size. So, the receiving end can just wait for one "message" with a certain type, and you don't have to worry if this is complete or not. Several processes may send to and receive from the same queue.

There are four system calls for messages: msgget returns (and possibly creates) a message descriptor that designates a message queue for use in other system calls, msgctl has options to set and return parameters associated with a message descriptor and an option to remove descriptors, msgsnd sends a message, and msgrcv receives a message.

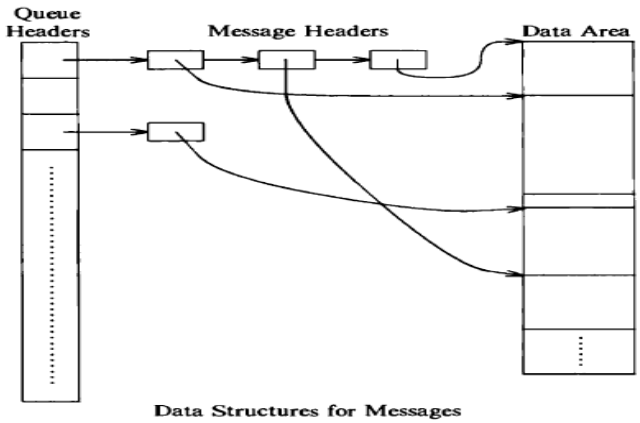
The syntax of the msgget system call is

msgqid — msgget(key, flag);

where msgqid is the descriptor returned by the call, and key and flag have the semantics described above for the general "get" calls. The kernel stores messages on a linked list (queue) per descriptor, and it uses msgqid as an index into an array of message queue headers. In addition to the general IPC permissions field mentioned above, the queue structure contains the following fields:

- Pointers to the first and last messages on a linked list.
- The number of messages and the total number of data bytes on the linked list.

- The maximum number of bytes of data that can be on the linked list.
- The process IDs of the last processes to send and receive messages.
- Time stamps of the last msgsnd, msgrcv, and msgctl operations.



Data Structures for Messages

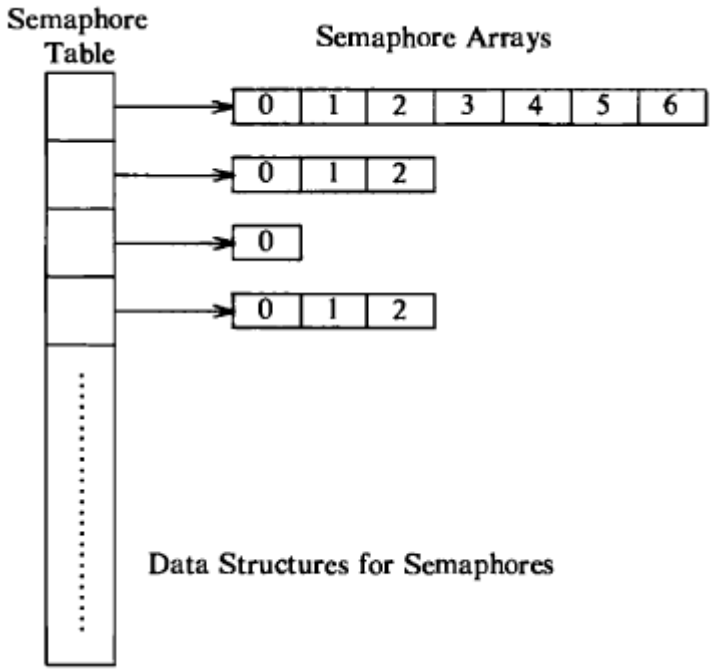
11.B. Write a program to demonstrate deadlock using semaphores. How can the IPC_NOWAIT flag be used to prevent deadlocks when using semaphores? Illustrate system V semaphores data structures by giving algorithm for semget() 6.5 Marks

implement the Producer & consumer Problem (Semaphore)

```
#define BUFFERSIZE 10
int mutex,n,empty,full=0,item,item1;
int buffer[20];
int in=0,out=0,mutex=1;
void wait(int s)
{
while(s<0)
{
printf("\nCannot add an item\n");
exit(0);
}
s--;
}
void signal(int s)
{
s++;
}
void producer()
{
do
{
wait (empty);
wait(mutex);
printf("\nEnter an item:");
scanf("%d",&item);
buffer[in]=item;
in=in+1;
signal(mutex);
signal(full);
}
while(in<n);
}
void consumer()
{
do
{
wait(full);
wait(mutex);
item1=buffer[out];
printf("\nConsumed item =%d",item1);
out=out+1;
signal(mutex);
signal(empty);
}
while(out<n);
}
void main()
{
printf("Enter the value of n:");
scanf("%d ",&n);
empty=n;
while(in<n)
producer();
while(in!=out)
consumer();
}
```

IPC_NOWAIT, do not suspend but return an error if the Zero test fails or the P operation cannot be done. The IPC_NOWAIT and SEM_UNDO flags are important when claiming multiple resources at once.

Specify SEM_UNDO on all operations; and specify IPC_NOWAIT on all but the first one. If the second or later resource is unavailable, semop() restores all preceding claims and returns an error code. As long as all processes or threads operate on semaphores in the same order, this logic prevents deadlocks, and it avoids long, fruitless suspensions.



The semaphore system calls are sem get to create and gain access to a set of semaphores, semai to do various control operations on the set, and semop to manipulate the values of semaphores.

The sem get system call creates an array of semaphores:

id = semget(key, count, flag);

where key, flag and id are similar to those parameters for messages and shared memory. The kernel allocates an entry that points to an array of semaphore structures with count elements (Figure 11.13). The entry also specifies the number of semaphores in the array, the time of the last semop call, and the time of the last semctl call. For example, the semget system call in Figure 11.14 creates a semaphore with two elements.

Processes manipulate semaphores with the semop system call:

oldval = semop(id, oplist, count);

Id is the descriptor returned by semget, oplist is a pointer to an array of semaphore operations, and count is the size of the array. The return value, oldval, is the value of the last semaphore operated on in the set before the operation was done. The format of each element of oplist is the semaphore number identifying the semaphore array entry being operated on, the operation, flags.

The kernel reads the array of semaphore operations, oplist , from the user address space and verifies that the semaphore numbers are legal and that the process has the necessary permissions to read or change

the semaphores. If permission is not allowed, the system call fails. If the kernel must sleep as it does the list of operations, it restores the semaphores it has already operated on to their values at the start of the system call; it sleeps until the event for which

Algorithm for Semaphore Operation:

```
algorithm semop              /* semaphore operations */
inputs:  (1) semaphore descriptor
         (2) array of semaphore operations
         (3) number of elements in array
output: start value of last semaphore operated on
{
      check legality of semaphore descriptor;
start: read array of semaphore operations from user to kernel space;
      check permissions for all semaphore operations;

      for (each semaphore operation in array)
      {
            if (semaphore operation is positive)
            {
                  add "operation" to semaphore value;
                  if (UNDO flag set on semaphore operation)
                        update process undo structure;
                  wakeup all processes sleeping (event semaphore value increases);
            }
            else if (semaphore operation is negative )
            {
                  if ("operation" + semaphore value >= 0)
                  {
                        add "operation" to semaphore value;
                        if (UNDO flag set)
                              update process undo structure;
                        if (semaphore value 0)
                              /* continued next page */
```

```
                              wakeup all processes sleeping (event
                                    semaphore value becomes 0);
                              continue;
                        }
                        reverse all semaphore operations already done
                              this system call (previous iterations);
                        if (flags specify not to sleep)
                              return with error;
                        sleep (event semaphore value increases);
                        goto start;       /* start loop from beginning */
                  }
                  else       /* semaphore operation is zero */
                  {
                        if (semaphore value non 0)
                        {
                              reverse all semaphore operations done
                                    this system call;
                              if (flags specify not to sleep)
                                    return with error;
                              sleep (event semaphore value == 0);
                              goto start;       /* restart loop */
                        }
                  }
            } /* for loop ends here */
            /* semaphore operations all succeeded */
            update time stamps, process ID's;
            return value of last semaphore operated on before call succeeded;
}
```

10) Many systems classify library functions as thread-safe or thread-unsafe. What causes a function to be unsafe for use by a multithreaded application? Illustrate the implementation of concurrent linked list that only allows one thread to access any given node at any instant.

Ans) Thread safety is the avoidance of data races--situations in which data are set to either correct or incorrect values, depending upon the order in which multiple threads access and modify the data. When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization to make certain that the program behaves in a deterministic manner. A procedure is thread safe when it is logically correct when executed simultaneously by several threads. At a practical level, it is convenient to recognize three levels of safety.

- Unsafe
- Thread safe - Serializable
- Thread safe - MT-Safe

Unsafe:
An unsafe procedure can be made thread safe and serializable by surrounding it with statements to lock and unlock a mutex. The below example shows three simplified implementations of `fputs()`, initially thread unsafe.

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}
```

Thread safe – Serializable:
Next is a serializable version of this routine with a single mutex protecting the procedure from concurrent execution problems. Actually, this is stronger synchronization than is usually necessary. When two threads are sending output to different files using `fputs()`, one need not wait for the other--the threads need synchronization only when they are sharing an output file.

```
/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);
```

```
    mutex_unlock(&m);
}
```

Thread safe - MT-Safe

The last version is MT-safe. It uses one lock for each file, allowing two threads to print to different files at the same time. So, a routine is MT-safe when it is thread safe and its execution does not negatively affect performance.

```
/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]0;
}
```

Implementation of concurrent linked list:

```
/* Concurrent Linked Lists */
#include <stdio.h>
#include <stdlib.h>
typedef struct __node_t {
 int key;
 struct __node_t *next;
} node_t;
// basic list structure (one used per list)
typedef struct __list_t {
node_t *head;
pthread_mutex_t lock;
} list_t;
void List_Init(list_t *L) {
 L->head = NULL;
 pthread_mutex_init(&L->lock, NULL);
}
int List_Insert(list_t *L, int key) {
 pthread_mutex_lock(&L->lock);
node_t *new = malloc(sizeof(node_t));
```

```
if (new == NULL) {
perror("malloc");
pthread_mutex_unlock(&L->lock);
return -1; // fail
}
new->key = key;
new->next = L->head;
L->head = new;
pthread_mutex_unlock(&L->lock);
return 0; // success
}
int List_Lookup(list_t *L, int key) {
pthread_mutex_lock(&L->lock);
node_t *curr = L->head;
while (curr) {
if (curr->key == key) {
pthread_mutex_unlock(&L->lock);
return 0; // success
}
curr = curr->next;
}
pthread_mutex_unlock(&L->lock);
return -1; // failure
}
void List_Print(list_t *L) {
 node_t *tmp = L->head;
 while (tmp) {
printf("%d ", tmp->key);
tmp = tmp->next;
 }
 printf("\n");
}
int main(int argc, char *argv[])
{
 list_t mylist;
```

```
 List_Init(&mylist);
 List_Insert(&mylist, 10);
 List_Insert(&mylist, 30);
 List_Insert(&mylist, 5);
 List_Print(&mylist);
 printf("In List: 10? %d 20? %d\n",
 List_Lookup(&mylist, 10), List_Lookup(&mylist, 20));
 return 0;
}
```

8)

Given that you can create multiple threads to perform different tasks within a program, explain why you might still need to use fork. List 5 pthread functions.

Fork is universally accepted than thread because of the following reasons:

Development is much easier on fork based implementations.

Fork based code a more maintainable.

Forking is much safer and more secure because each forked process runs in its own virtual address space. If one process crashes or has a buffer overrun, it does not affect any other process at all.

Threads code is much harder to debug than fork.

Fork are more portable than threads.

Forking is faster than threading on single cpu as there are no locking over-heads or context switching.

5 pthread functions:

1) pthread_create: used to create a new thread. ...

2) pthread_exit: used to terminate a thread. ...

3) pthread_join: used to wait for the termination of a thread. ...

4) pthread_self: used to get the thread id of the current thread.

5) pthread_equal: compares whether two threads are the same or not

7. These two rules says when locks are necessary but say nothing about when locks are unnecessary. It is important for efficiency not to lock too much, because locks reduce parallelism. If parallelism isn't important, then one could arrange to have only a single thread and not worry about locks. A simple kernel can do this on a multiprocessor by having a single lock that must be acquired on entering the kernel and released on exiting the kernel

Locks in xv6:

- xv6 has two types of locks: spin-locks and sleep-locks.
- xv6 represents a spin-lock as a struct spinlock.

Spin locks:

- The important field in the structure is locked
- a word that is zero when the lock is available and non-zero when it is held.
- Logically, xv6 should acquire a lock by executing code like
- Locks (i.e., spinlocks) in xv6 are implemented using the xchg atomic instruction
- xv6 uses locks in many places to avoid race conditions
- A hard part about using locks is deciding how many locks to use and which data and invariants each lock protects.
- There are a few basic principles.
- **First**, any time a variable can be written by one CPU at the same time that another CPU can read or write it, a lock should be introduced to keep the two operations from overlapping.
- **Second**, remember that locks protect invariants: if an invariant involves multiple memory locations, typically all of them need to be protected by a single lock to ensure the invariant is maintained.

Sleep locks:

- Sometimes xv6 code needs to hold a lock for a long time.
- For example, the file system keeps a file locked while reading and writing its content on the disk, and these disk operations can take tens of milliseconds.
- Efficiency demands that the processor be yielded while waiting so that other threads can make progress, and this in turn means that xv6 needs locks that work well when held across context switches.
- xv6 provides such locks in the form of sleep-locks.
- Xv6 sleep-locks support yielding the processor during their critical sections.
- Because sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.
- Because acquiresleep may yield the processor, sleep-locks cannot be used inside spin-lock critical sections (though spin-locks can be used inside sleep-lock critical sections).
- xv6 uses spin-locks in most situations, since they have low overhead.
- It uses sleep-locks only in the file system, where it is convenient to be able to hold locks across lengthy disk operations

6.A Consider a three-level page table organization as shown in the figure below. If a program is 4 Giga bytes, what is the total space needed for its page table (that is, the total space needed by directories and partial page tables)?

Answer:

The last 12 bits of the virtual address are the offset in a page, varying from 0 to 4095. So the page size is 4096, that is, 4K.

6.B. a)If you have to design a virtual memory management with demand paging, please design an algorithm to manipulate the hash frame table for the used frames.

Ans:

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

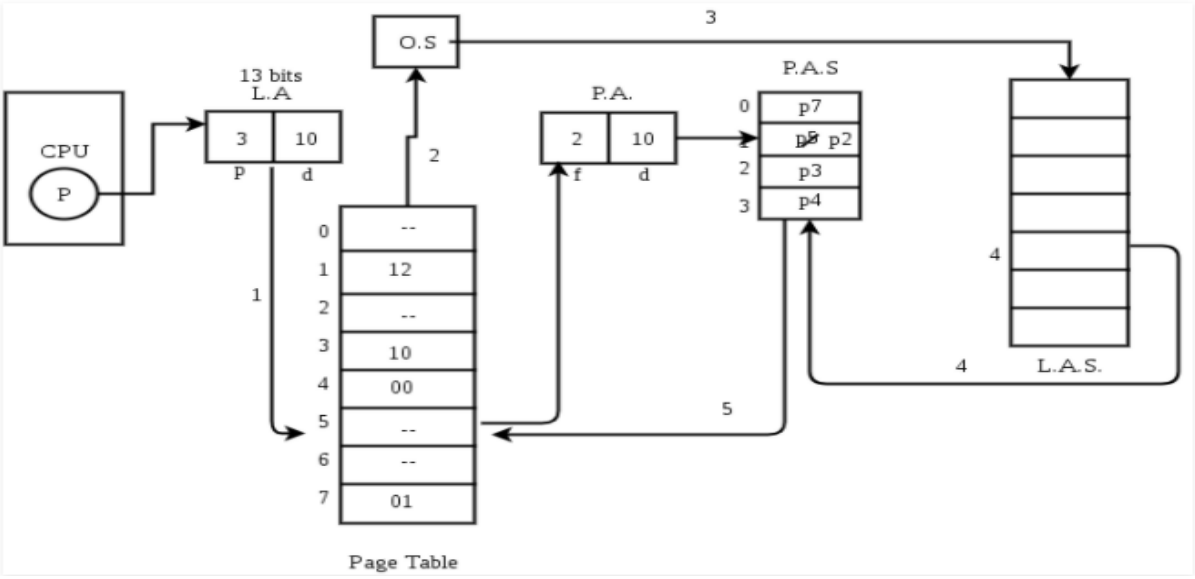
1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.
2. A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging :

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

The process includes the following steps :



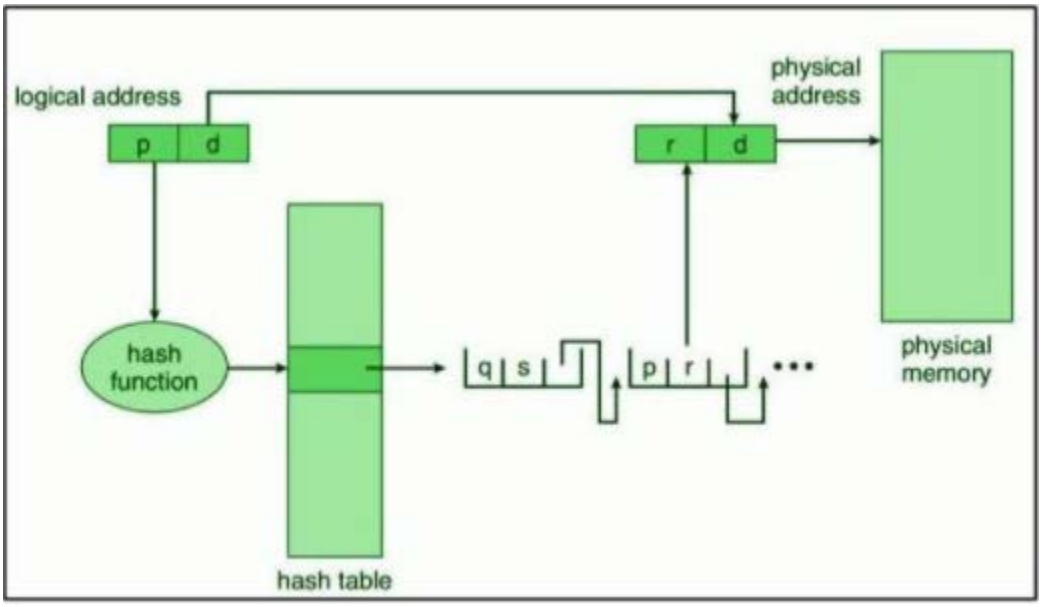
1. If CPU try to refer a page that is currently not available in the main memory, it generates an interrupt indicating memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision making of replacing the page in physical address space.
5. The page table will updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.

Hashed Page Tables :

In hashed page tables, the virtual page number in the virtual address is hashed into the hash table. They are used to handle address spaces higher than 32 bits. Each entry in the hash table has a linked list of elements hashed to the same location (to avoid collisions – as we can get the same value of a hash function for different page numbers). The hash value is the virtual page number. The Virtual Page Number is all the bits that are not a part of the page offset.

For each element in the hash table, there are three fields –

1. Virtual Page Number (which is the hash value).
2. Value of the mapped page frame.
3. A pointer to the next element in the linked list.


logical address
p
d
physical address
r
d
hash function
hash table
q
s
p
r
physical memory

B.Tech - Odd Sem : Semester in Exam-II

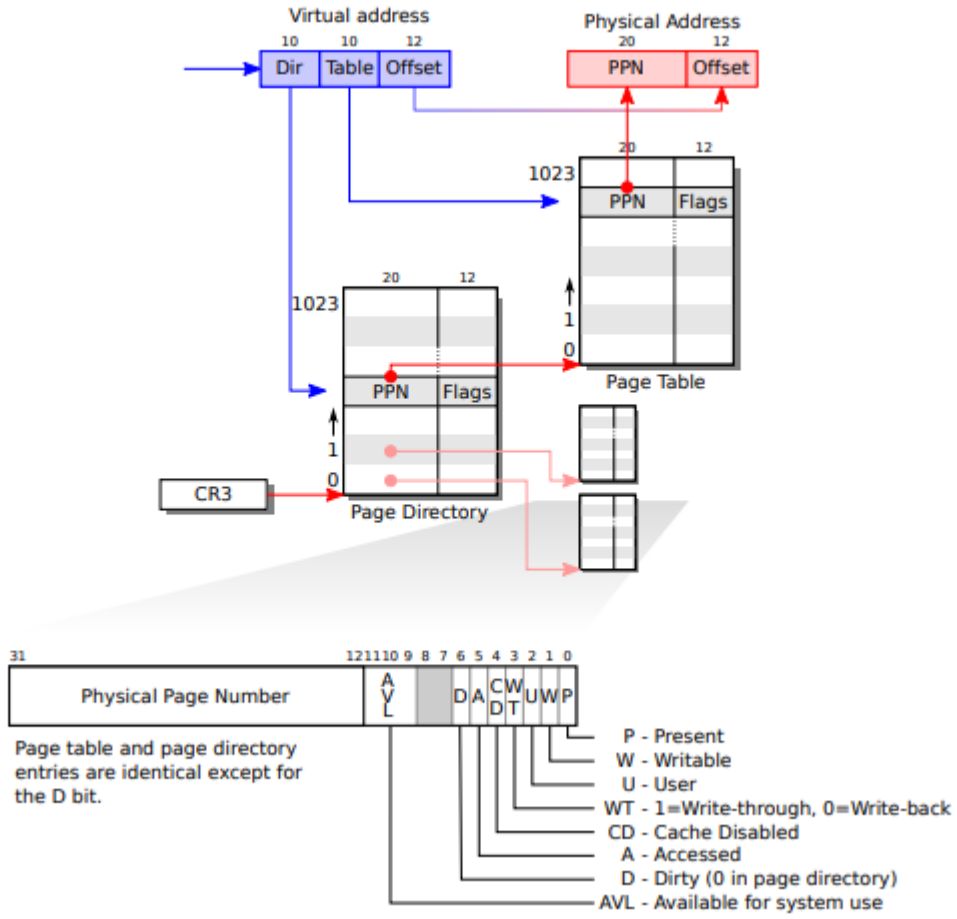
Academic Year:2020-2021 19CS2106S - OPERATING SYSTEMS DESIGN - S

Set No: 1

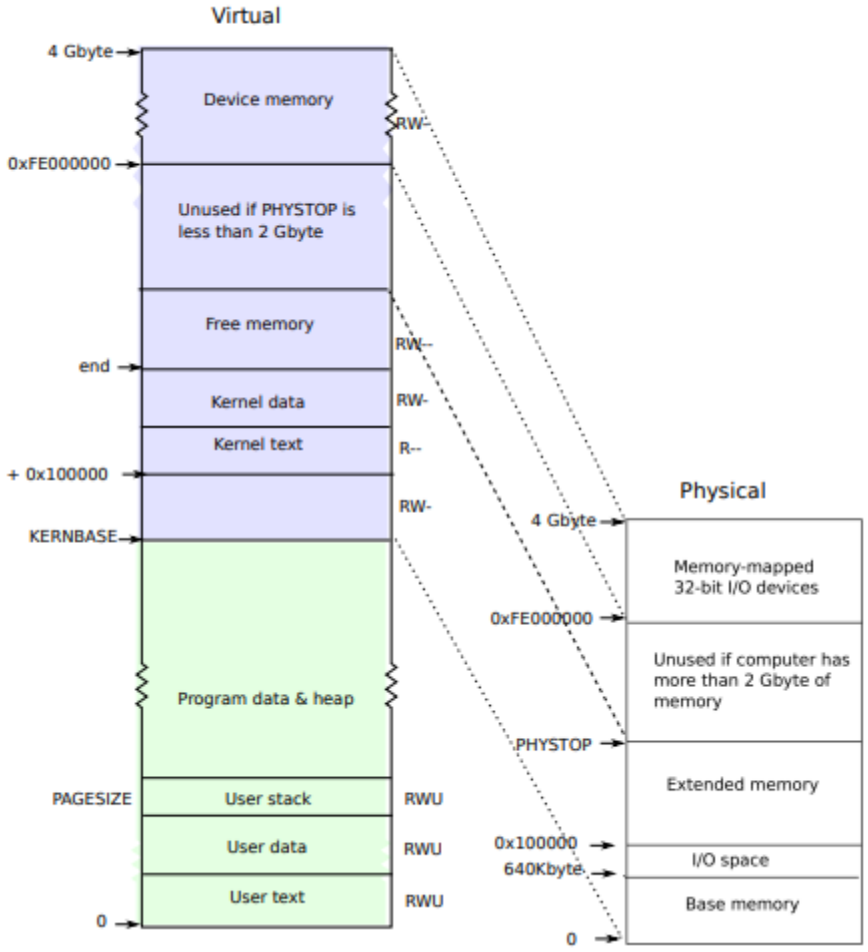
5A. In xv6, explain first address space using Paging. Illustrate creating and running first process. (explanation 3M, figure 3M)

main calls kvmalloc (1840) to create and switch to a page table with the mappings above KERNBASE required for the kernel to run. Most of the work happens in setupkvm (1818). It first allocates a page of memory to hold the page directory. Then it calls mappages to install the translations that the kernel needs, which are described in the kmap (1809) array. The translations include the kernel's instructions and data, physical memory up to PHYSTOP, and memory ranges which are actually I/O devices. setupkvm does not install any mappings for the user memory; this will happen later.

mappages (1760) installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid (1772). walkpgdir (1735) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address. walkpgdir uses the upper 10 bits of the virtual address to find the page directory entry (1740). If the page directory entry isn't present, then the required page table page hasn't yet been allocated; if the alloc argument is set, walkpgdir allocates it and puts its physical address in the page directory. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (1753)



x86 page table hardware.



Layout of the virtual address space of a process and the layout of the physical address space. Note that if a machine has more than 2 Gbyte of physical memory, xv6 can use only the memory that fits between KERNBASE and 0xFE00000.

5 B. For a given virtual address in a binary number, write and explain OS Handled translation lookaside buffer control flow algorithm (Algorithm 4M , Explanation 2.5M)

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True) // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset = VirtualAddress & OFFSET_MASK
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7            Register = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
10   else // TLB Miss
```

```
11    PTEAddr = PTBR + (VPN * sizeof(PTE))
12    PTE = AccessMemory(PTEAddr)
13    if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15    else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17    else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

TLB Control Flow Algorithm

The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a TLB hit, which means the TLB holds the translation.

Success! We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4). If the CPU does not find the translation in the TLB (a TLB miss), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These set of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache (i.e., are hits). If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to

most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.

Q4)

a) KERNBASE limits the amount of memory a single process can use, which might be irritating on a machine with a full 4 GB of RAM. Would raising KERNBASE allow a process to use more memory?

Answer: The answer to this question is no, since the whole mechanism around xv6 is designed to work with KERNBASE on a specific address space.
xv6 can work with both higher and lower KERNBASE values. We can test this by changing KERNBASE to, say, 0x90000000 and then changing the relevant value in kernel.ld (the linker script which puts things in expected addresses).
The real issue here, is that xv6 doesn't do any paging to disk. Now, in xv6 addresses 0x80000000 (KERNBASE) and up map linearly to 0x00000000..0xffffffff. This means that any byte of memory you allocate in the whole system maps to 2 different physical addresses in 32-bit space. Since xv6 does no paging to disk, this means that if it allocates memory for the user process (using the sbrk() system call, used by malloc() in userspace), then it keeps it around in memory the whole time. So again, since we have 2 "copies", or more precisely 2 mappings to the same address, we can't ever actually use more than half the memory available in 32-bit address space.
Now, recall that KERNBASE is defined as 0x80000000, which is exactly that: half of the available memory. So no, raising KERNBASE under these conditions can't give us more userspace memory.

b) How does the kernel ensure consistency of the TLB and the virtual address cache during an exec system call? Since the UNIX kernel is nonpaged, what could lead to a change in a TLB entry for a kernel page?

Answer:

As a general rule, any process switch implies changing the set of active Page Tables. Local TLB entries relative to the old Page Tables must be flushed; this is done automatically when the kernel writes the address of the new Page Global Directory into the cr3 control register. In some cases, however, the kernel succeeds in avoiding TLB flushes, which are listed here:

• When performing a process switch between two regular processes that use the same set of Page Tables

• When performing a process switch between a regular process and a kernel thread. Kernel threads do not have their own set of Page Tables; rather, they use the set of Page Tables owned by the regular process that was scheduled last for execution on the CPU.

Beside process switches, there are other cases in which the kernel needs to flush some entries in a TLB. For instance, when the kernel assigns a page frame to a User Mode process and stores its physical address into a Page Table entry, it must flush any local TLB entry that refers to the corresponding linear address. On multiprocessor systems, the kernel must also flush the same TLB entry on the CPUs that are using the same set of Page Tables, if any.

To invalidate TLB entries, the kernel uses the following functions and macros:

flush tlb one

Invalidates the local TLB entry of the page, including the specified address.

flush tlb page

To avoid useless TLB flushing in multiprocessor systems, the kernel uses a technique called lazy TLB mode. The basic idea is if several CPUs are using the same Page Tables and a TLB entry must be flushed on all of them, then TLB flushing may, in some cases, be delayed on CPUs running kernel threads.

3.a) Some UNIX systems have a vfork system call that creates a new process but requires that the child does not return from the function that called vfork and that it invokes only the exit or execve system calls. Why is this curious system call a useful addition? How and why do modern CPUs reduce the need for vfork?

Ans:The restrictions imposed by the vfork system call mean that the child process can initially share its address space with its parent, rather than requiring that the OS makes a copy .the time spent doing that copy would be wasted if the child performs an exit or execve system call soon after. Vfork is less important in modern systems that support copy on write.

The fork system call may initially appear more complex to use because it makes loading and executing a program in a new process a two-stage operation first of all forking a new process and then using execve to load and execute the specified program. The benefit of this is that fork can be used to create hierarchies of co-operating processes from the same executable image (i.e. when fork is *not* followed by execve) whereas a simple create process operation could only load and execute files that can be named.:----4 Marks

3b)Give a solution to deallocate the swap space on the disk in detail.

b)**Swap-SpaceManagement :**

Swap-Swap management is another low-level task pf the operating system. Disk space is used as an extension of main memory by the virtual memory. As we know the fact that disk access is much slower than memory access, In the swap-space management we are using disk space, so it will significantly decreases system performance. Basically, in all our systems we require the best throughput, so the goal of this swap-space implementation is to provide the virtual memory the best throughput. In these article, we are going to discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

Swap-SpaceUse :

Swap-space is used by the different operating-system in various ways. The systems which are implementing swapping may use swap space to hold the entire process which may include image, code and data segments. Paging systems may simply store pages that have been pushed out of the main memory. The need of swap space on a system can vary from a megabytes to gigabytes but it also depends on the amount of physical memory, the virtual memory it is backing and the way in which it is using the virtual memory.

Theory explanation 2marks
Solution 2 marks .

Q2 Please give your version of the realloc() call using system calls to allocate the swap space for the process to be swapped out You can descript it with a C-like algorithm

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

When swapper wakes up to swap processes in, it selects the processes which has spent the longest time in the state "ready to run but swapped out". The algorithm *swapper* is given below:

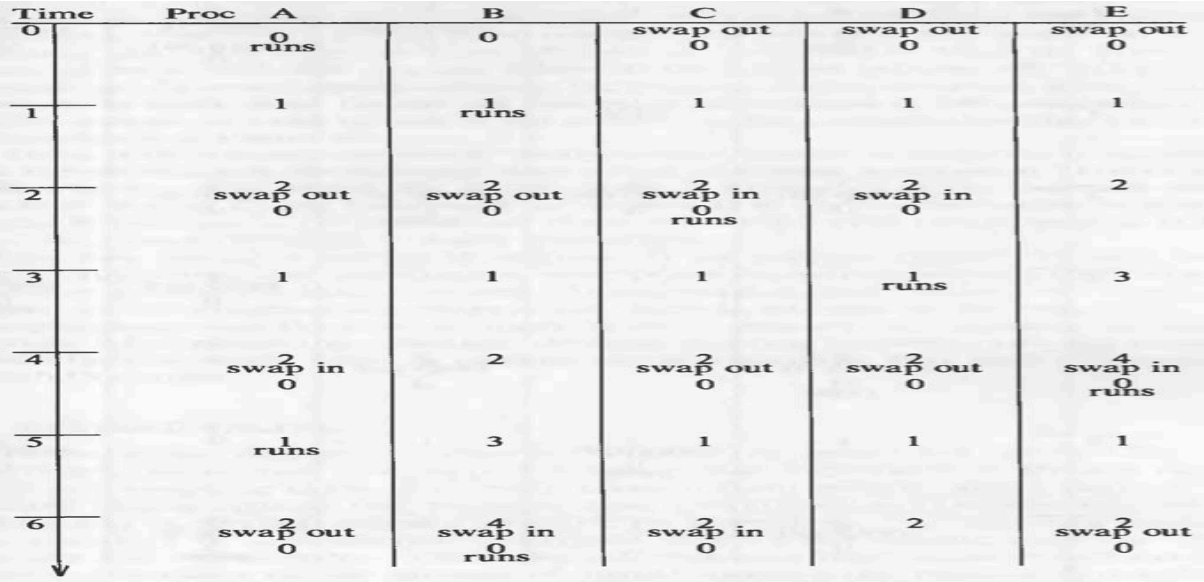
```
/*  Algorithm: swapper
 *  Input: none
 *  Output: none
 */

{
        loop:
                for (all swapped out processes that are ready to run)
                        pick process swapped out longest;
                if (no such process)
                {
                        sleep (event: must swap in);
                        goto loop;
                }
                if (enough room in main memory for process)  →2.5M
                {
                        swap process in;
                        goto loop;
                }
        // loop2: here in revised algorithm given later
                for (all processes loaded in main memory, not zombie and not locked
in memory)
                {
                        if (there is a sleeping process)
                                choose process such that priority + residence time
is numerically highest;
                        else            // no sleeping process
                                choose process such that residence time + nice is
numerically highest;
                if (chosen process not sleeping or residency requirements not
satisfied)
                        sleep (event must swap process in);   --→2M
                else
                        swap out process;
                goto loop;      // goto loop2 in revised algorithm given later
}
```

The swapper tries to swap out sleeping processes rather than ready-to-run processes, because ready-to-run processes might get scheduled earlier than sleeping processes. A ready-to-run process must be core resident for at least 2 seconds before being swapped out, and a process to be swapped in must have been swapped out for at least 2 seconds.

If the swapper cannot find a processes to swap out or if neither the process to be swapped in nor the process to be swapped out have accumulated more than 2 seconds residence time in their environment, then swapper sleeps on the event that it wants to swap a process into memory but cannot find room for it. The clock will awaken the swapper once a second in that state. The kernel also awakens swapper if another process goes to sleep, since it may be more eligible for swapping out than the processes previously considered by the swapper. In any case, the swapper wakes up and begins execution from the beginning, attempting to swap in eligible processes.

Here is an example of process scheduling in a system where A, B, C, D are same sized processes and only 2 processes can stay in main memory. This is the scheduling timeline: ----------→2.5M



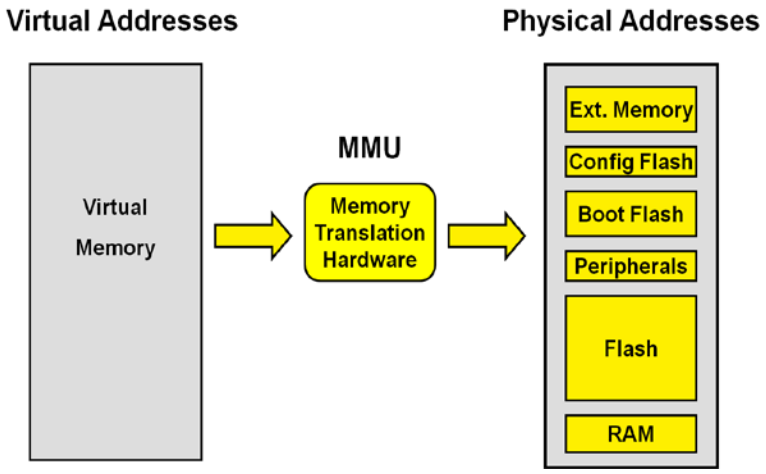
The swapper chooses processes to swap in based on the amount of time the processes had been swapped out. Another criterion could have been to swap in the highest-priority process that is ready to run, since such processes deserve a better chance to execute. It has been demonstrated that such a policy results in "slightly" better throughput under heavy system load.

1. **Illustrate hardware-based dynamic relocation, Hardware Requirements and Operating System Responsibilities**

Transforming a **virtual address into a physical address** is referred as **address translation**; that is, the hardware takes a virtual address and transforms it into a physical address which is where the data actually resides. Because this relocation of the address **happens at runtime,** and because we can move address spaces even after the process has started running, the technique is often referred to as **dynamic relocation.**

To implement Dynamic (Hardware-based) Relocation, we'll need two hardware registers within each CPU: one is called the base register, and the other the bounds (sometimes called a limit register).The base and bounds registers are kept on the chip (one pair per CPU) which is called as **memory management unit (MMU).**

physical address = virtual address + base



Each memory reference generated by the process is a virtual address; the hardware in turn adds the contents of the base register to this address and the result is a physical address that can be issued to the memory system

Dynamic Relocation: Hardware Requirements

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Dynamic Relocation: Operating System Responsibilities

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise; likely action is to terminate offending process*

What three segments are usually found in the memory allocated to a process?

Text Segment:

The Text segment (a.k.a the Instruction segment) contains the executable program code and constant data. The text segment is marked by the operating system as read-only and can not be modified by the process. Multiple processes can share the same text segment. Processes share the text segment if a second copy of the program is to be executed concurrently. In this setting, the system references the previously loaded text segment with the pointer rather than reloading a duplicated. If needed, shared text, which is the default when using the C/C++ compiler, can be turned off by using the -N option on the compile time.

Data Segment:

The data segment, which is contiguous (in a virtual sense) with the text segment, can be subdivided into initialized data (e.g. in C/C++, variables that are declared as static or are static by virtual of their placement) and uninitialized (or 0-initizliazed) data. The uninitialized data area is also called BSS (Block Started By Symbol). For example, Initialized Data section is for initialized global variables or static variables, and BSS is for uninitialized. During its execution lifetime, a process may request additional data segment space. Library memory allocation routines (e.g., new, malloc, calloc, etc.) in turn make use of the system calls brk and sbrk to extend the size of the data segment. The newly allocated space is added to the end of the current uninitialized data area. This area of available memory is also called "heap". Generally speaking, you can call the whole data area as heap, but restrictly, people only refers the umapped area in the fig.

Stack Segment:

The stack segment is used by the process for the storage of automatic identifier, register variables, and function call information. In the above figure, the stack grows towards the uninitialized data segment.

Distinguish between an activation record and a stack frame?

stack frame:

A stack frame is a memory management technique used in some programming languages for generating and eliminating temporary variables. In other words, it can be considered the collection of all information on the stack pertaining to a subprogram call. Stack frames are only existent during the runtime process. Stack frames help programming languages in supporting recursive functionality for subroutines.

A stack frame also known as an activation frame or activation record.

An **Activation Record** is a data structure that holds all the information needed to support one call of a function. It contains all the local variables of that function, and a reference (or pointer) to another

activation record; that pointer is known as the **Dynamic Link**. Stack Frames are an implementation of Activation Records. The dynamic link corresponds to the "saved FP" entry; it tells you which activation record to return to when the current function is finished. The frame pointer itself is simply a way of indicating which activation record is currently in use. The dynamic links tie all the activation records for a program together in one long linked list, showing the order they would appear in a stack.

How long does a stack frame last?

Local variables come and go at the whim of a running program. If a function is called resursively, many different versions of its local variables will exist at the same time, each having their own private allocation of memory. Local variables are kept on a **Stack**. Before a program starts, the **Stack Pointer**, a hardware CPU register that is reserved for pointing to the end of the stack, is set to point to the highest addresses available.

Every time a function is called, space is made on the stack for all of its local variables (plus a little extra) simply by subtracting from SP the number of bytes required. Usually at this point the new value of the stack pointer is copied into another reserved CPU register called the **Frame Pointer**. This allows the stack to still be used for saving temporary results, without losing track of our position.

What is contained in a stack frame?

The stack frame, also known as activation record is the collection of all data on the stack associated with one subprogram call.

The stack frame generally includes the following components:

The return address

Argument variables passed on the stack

Local variables (in HLLs)

Saved copies of any registers modified by the subprogram that need to be restored (e.g. $s0 - $s8 in MAL).

```
#include <stdio.h>

void f(void)
{       int x,y;
        static int w;
        printf("statics in f:   &w=%10u\n", &w);
        printf("locals in f:    &x=%10u\n", &x);
        printf("                &y=%10u\n", &y);
}

int a,b;

void main(void)
{
```

```
        int m,n;
        static o;
        printf("functions:   &main=%10u\n", &main);
        printf("                &f=%10u\n", &f);
        printf("globals:        &a=%10u\n", &a);
        printf("                &b=%10u\n", &b);
        printf("dynamic:  malloc()=%10u\n", malloc(100));
        printf("          malloc()=%10u\n", malloc(100));
        printf("static in main: &o=%10u\n", &o);
        f();
        printf("locals in main: &m=%10u\n", &m);
        printf("                &n=%10u\n", &n);
}
```