PIPES

Create pipe (unidirec comm buff w/ 2 file descrips fd[0] read & fd[1] write)

```
#include <unistd.h>
int pipe(int fd[2]);
```

Data write & read FIFO base. No external/permanent name; only accessed via 2 fds.

Pipe can only be used by process that created it & its descendants.

```
close(fd) closes a file descriptor
dup(newfd) duplicates fd, dup2(newfd, oldfd) copies (more like alias)
```

Read:	Write:
Not nec atomic; may read less bytes Blocking: if no data, write fd still opens. If empty & all fd for write closes; read sees eof, returns 0	Atomic for at most PIPE_BUF bytes (512, 4k, 64k) Blocking: if buffer full & read fd open. When all fd to read closed, causes SIGPIPE sig for calling

Pros: simple, flexible, efficient comms

Cons: no way to open already existing pipe: impossible for 2 arbitrary processes to share same pipe (unless created by common ancestor)

IPC: UNIX Shared Mem

Parent & child processes run in separate addy spaces.

Shared mem segment: piece of mem that can be allocated & attached to addy space; process w/ this mem seg attached will have access to it but race conditions can occur

Procedure: find a key (unix uses key to identify shared mem segments) -> shmget() allocates a shared mem -> shmat() attach shared mem to addy space -> shmdt() detach mem from addy space -> shmctl() deallocate shared mem

Keys: global; If other processes know key, can access shared mem

Can ask sys to provide private key using IPC_PRIVATE

DIY:

```
key_t SomeKey;
SomeKey = 1234;
```

Autogenerate:

```
key_t = ftok(char *path, int ID);
```

Asking for Shared Memory:

Use shmget() to request a shared mem (returns shared mem ID):

```
shm_id = shmget(key_t key, int size, int flag)
```

Flag for our purpose either 0666 (rw) or IPC_CREAT|0666

Include following:

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

Attaching Shared Memory:

Use shmat() to attach existing shared mem to addy space (ret void pointer to mem):

```
shm_prt = shmat(int shm_id, char *ptr, int flag);
    shm_id ID from shmget(), use NULL for ptr, flag = 0
```

Detaching & Removing Shared Memory:

To detach shared memory, use shmdt(shm_ptr);
shm_ptr is pointer returned by shmat().

After shared mem *detached*, it's still there (can be reattached to use again. *Removing* a shared mem will make it stop existing)

To remove shared memory, use shmctl(shm_ID, IPC_RMID, NULL);
shm_ID is shared mem ID returned by shmget().

INTERPROCESS COMMUNICATION

Types: Message passing (Blocking/non-blocking, Datagrams, virtual circuits, streams, Remote Procedure Calls (RPC)), Shared Memory

Shared Memory: >=2 processes share part of their addy space

Adv: fast & easy to use (but concurr access to shared data can cause issues, and must sync access to shared data)

Disadv: senders & receivers must be on same machine, less secure (processes can directly access part of addy space of other processes)

Message Passing: processes want to exchange data send & receive msgs

One send and one receive:

```
send(addr, msg, length);
receive(addr, msg, length);
```

Adv: senders & receivers can be on diff machines; receiver can inspect msgs received before processing them

Disadv: hard to use (every data transfer requires send() & receive(), and receiving process must expect send())

Defining Issues of Msg Passing: Direct/Indirect communication, Blocking/Non-Blocking primitives, exception handling, quality of service

Direct Comm: send & rec calls specify processes as destination/source:

```
send(process, msg, length);
receive(process, msg, &length);
```

No intermediary b/w sender & receiver. Ex. each phone hardwired to another

Proc exe rec call must know identity of all procs likely to send msgs (bad for servers)

Indirect Comm: send & rec primitives specify intermediary as destination or source (mailbox: sys obj created by kernel @ request of user process):

```
send(mailbox, msg, size);
receive(mailbox, msg, &size);
```

Diff procs can send msgs to same mailbox. A proc can rec msgs from procs it knows nothing about, can wait for msgs from diff senders (answ 1st msg rec)

Private Mbox aka ports: proc requesting creation & children are only ones that can rec msgs through that mbox. Ceases to exist when proc requesting its creation (and all children) terminates.

Public Mboxes: owned by sys, & shared by all procs having right to rec msgs through it (works best when all procs on same machine). Survives termination of proc that requested creation. Ex. msg queues

Blocking Primitives: *blocking send* doesn't return til receiving process has received msg; no buffering needed. *Blocking receive* doesn't return til msgs have been received (std choice, but not good choice for direct comm)

Non-Blocking Primitives: *Non-blocking send* returns as soon as msg has been accepted for delivery by OS (assuming OS can store msg in a buffer, std choice). *Non-blocking receive* returns as soon as it has either retrieved msg or learned mbox is empty (retrieve() acts as receive() for receiver).

Simulating blocking receives: can sim blocking receive w/ non-blocking receiving within a loop (aka busy wait)

```
do {
   Code = receive(mbox, msg, size);
   sleep(1); // delay
} while (code == EMPTY_MBOX);
```

Simulating blocking sends: can sim blocking send w/ 2 non-blocking sends & a blocking receive

Sender sends msg & requests ACK -> sender wait for ACK from receiver using blocking receive -> receivers sends ACK

Non-blocking primitives require buffering to let OS store msgs that have been sent but not received. Buffers have a bounded capacity, but theoretically unlimited capacity.

Exception Condition Handling: must specify what to do if ½ processes dies. Esp important when the 2 procs on diff machines (must handle host failures & network partitions)

Quality of Service: when sender & receiver on diff machines, msgs can be lost, corrupted or duped, and can arrive out of sequence. Can still decide to provide reliable msg delivery

Datagrams: msgs sent individually (can be lost, duped, out of seq).

Reliable: msgs resent until ACK. Unreliable: msgs not ACK (works well when msg requests reply which acts as implicit ACK)

UDP (User Datagram Protocol) best known datagram protocol. Provides unreliable datagram services which is best for short interactions

Virtual Circuits: establishes logical connection b/w sender & receiver. Msgs guaranteed to arrive in seq w/o lost/duped msgs.

Requires virtual connection before sending any data. Best for transmitting large data amounts requiring sending several msgs (FTP, HTTP)

Streams: like virtual circuits, but does NOT preserve msg boundaries (seamless stream of bytes).

TCP (Transmission Control Protocol): best known stream protocol, providing reliable stream serv. Heavyweight (needs 3 msgs to estab virtual connection)

Remote Procedure Calls (RPC): applies to client-server model

```
send_req(args);                    rcv_reqs(&args);
                                   process(args, &results);
                                   send_reply(results);
rcv_reply(&results);
```

Adv: hides all details of msg passing, provides higher level of abstraction, extends well-known model of programming

Disadv: illusion not perfect (RPCs don't behave exactly like regular procedure calls, client & server don't share same addy space), programmer must be aware of differences

User program contains user code & calls user stub that appears to call server procedure

```
rpc(xyz, args, &results);
```

User stub procedure generated by RPC package: parcks args into request msg & performs required conversions (arg marshaling) -> sends request msg -> waits for server reply -> unpacks results & performs required conversions (arg unmarshaling)

Server stub: generic server generated by RPC pack waits for client requests, unpack request args & performs required data conversions, calls appropriate server procedure (which is written by user & does actual processing), packs results into reply msg (performs required conversions), sends reply msg

Client & server processes don't share same addy space: no global variables, can't pass by ref, can't pass dynam data structs via pointers; RPC can pass args by value & result (passes curr val to RP & copies returned val in user program).

CHAPTER 5: CONCURRENCY

Multi App: invented to allow processing time to be shared among active apps

Structures Appts: extension of modular design & struct programming

OS Structure: OS themselves implemented as set of processes/threads

Key Terms:

Atomic Operation: function/action implemented as sequence of >=1 instructions appearing indivisible. Sequence is guaranteed ot exe as a group or not at all

Critical Section: section within proc that requires access to shared resources & mut not be exe whle another process is in a corresponding section of code

Deadlock: situation where >= 2 processes unable to proceed bc each is waiting for one of the others to do something

Livelock: >=2 procs continuously change states in response to changes in other proc(s) w/o doing useful work

Mutual Exclusion: requirement when one process is in crit sect that accesses shared resources, no other proc may be in crit sect that accesses any of those shared resources

Principles: Interleaving & overlapping (ex of concur processing), Uniprocessor – relative speed of exe of procs can't be predicted; dependent on activities of other procs, OS interrupt handling, OS scheduling policies

Difficulties: sharing of global resources, hard for OS to optimize resource allocation management, hard to locate programming errors

Rare condition: occurs when multi procs/threads read & write data items. Final result depends on order of exe ("loser" updates last, determines final val)

OS Concerns (Design & mgmt. issues): OS keeping track of various procs, allocation & deallocate resources for each active proc, protect data & phys resources of each proc against interference, ensure procs & outputs are independent of processing speed

Interaction Process

Degree of Awareness	Relationship	Influence of 1 Process on the other	Potential Ctl Probs
Processes unaware of each other	Competition	- Results of 1 process independent of others - Timing may be affected	- Mut Exclusion - Deadlock (renewable) - Starvation
Processes indirectly aware of each other	Coop by sharing	- Results of 1 proc may depend on info fr others - Timing may be affected	- Mut Exclusion - Deadlock (renewable) - Starvation - Data coherence
Processes directly aware of each other (have comm prim available to them)	Coop by comm	- Results of 1 proc may depend on info from others - Timing may be affected	- Deadlock (consumable) - starvation

Resource Competition: concurr procs competing for use of same resource (I/O devices, mem, proc time, clock). Control Probs: need mutual exclusion, deadlock, starvation

Mutual Exclusion Requirements: Must be enforeced, proc that halts has to w/o interfering w/ other procs, no deadlock/starvation, proc must not be denied access to crit sect when no other proc using it, no assump made about relative proc speeds/# of procs, proc remains inside crit sect for finite time only

Mutual Exclusion hardware supp: Special machine instructions – compare & swap (compare & exchange instruction). Compare made b/w mem val & test val. If same, swap.

Compare & Swap Instructions:

```
const int n = /* # processes */;
int bolt;
void P(int i) {
   while (true) {
      while (compare_and_swap(bolt, 0, 1)  == 1)
         /* do nothing*/;
      /* crit section */;
      bolt = 0;
      /* remainder */;
   }
}
void main() {
   bolt = 0;
   parbegin (P(1), P(2), … , P(n));
```

Exchange Instructions:

```
const int n = /* # processes */;
int bolt;
void P(int i) {
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* crit section */;
      bolt = 0;
      /* remainder */;
   }
}
void main() {
   bolt = 0; bolt = 0;
   parbegin (P(1), P(2), … , P(n));
}
```

Special Machine Instruct Adv: applicable to any # of pros on single/multiple processors sharing main mem, simple & easy to verify, can be used to supp multi crit sects (each can be defined by its own variable)

Disadv: busy-wait employed, while a proc is waiting for access to ctir section it cont to consume processor time, starvation is possible when proc leaves crit sect & >= 1 proc is waiting, deadlock is possible

Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphore: A variable w/ int val where only 3 ops are defined -> no way to inspect/manip semaphores other than those ops: 1) may be init to nonneg int val 2) semWaita op decrements val 3) semSignal op increments val

Cons: no way to know before proc decrements semaphore if block, no way to know which proc will cont immediately on uniproc sys when 2 running concurr, don't know if another proc is waiting so # of unblocked proc is 0 or 1

Semaphore Primitives

```
struct semaphore {
   int count;
   queueType queue;
};
void semWait(semaphore s) {
   s.count--;
   if (s.count < 0) {
      /* place process in s.queue */;
      /* block this process */;
   }
void semSignal(semaphore s) {
   s.count++;
   if (s.count <= 0) {
      /* remove process P from s.queue */;
      /* place process P on ready list */;
   }
}
```

Strong semaphores: the proc that's been blocked the longest released fr queue (FIFO)

Weak semaphores: order where procs removed from queue not specified

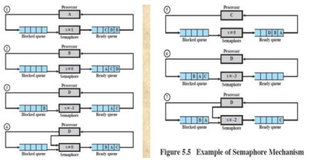


Figure 5.5 Example of Semaphore Mechanism

Mutual Exclusion using semaphores

```
const int n = n /* # processes */
semaphore s = 1;
void P(int i) {
   while (true) {
      semWait(s);
      /* crit section */;
      semSignal(s);
      /* remainder */;
   }
}
void main() {
   parbegin (P(1), P(2), … , P(n));
```

Producer/Consumer Problem: Ensure that produced can't add data into full buff & consumer can't remove data from empty buff

General situation: >= 1 producers gen data & placing in buff where a consumer taking items out individually, but only 1 producer/consumer may access buff at any one time.

Solution using semaphores:

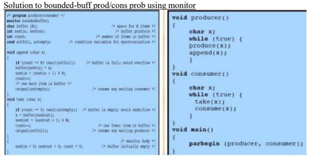
```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer() {
   while (true) {
      produce();
      semWait(s);
      append();
      semSignal(s);
      semSignal(n);
   }
}
void consumer() {
   while (true) {
      semWait(n);
      semWait(s);
      take();
     semSignal(s);
     consume();
   }
}
void main() {
   parbegin (producer, consumer);
}
```

Implementation of semaphore: imperative semWait & semSignal ops implemented as atomic primitives in hardware/firmware, Dekker's/peterson's algorithm can be used. Use one of hardware-supported schemes for mutual exclusion

Monitors: construct that provides equiv functionality to semaphores that is easier to control that is implemented in a number of langs, & as program lib. Software module consisting of >= 1 procedures, an init sequence, & local data

Characteristics: local data variables accessible only by mon's procedures (no ext procedure), process enters mon by invoking one of its procedures, only 1 proc exe in mon at a time. Synchronization: achieved by condition variables contained within & only accessible within mon. cwait(c) suspends exe of calling proc on condition c, csignal(c) resumes exe of some proc blocked after cwait on same condition

Solution to bounded-buff prod/cons prob using monitor



```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                              /* space for N items */
int nextin, nextout;                          /* buffer pointers */
int count;                                    /* number of items in buffer */
cond notfull, notempty;        /* condition variables for synchronization */
void append (char x)
{
    if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                        /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0) cwait(notempty);  /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                  /* one fewer item in buffer */
    csignal(notfull);                         /* resume any waiting producer */
}
{                                             /* monitor body */
    nextin = 0; nextout = 0; count = 0;       /* buffer initially empty */
}
```

```
void producer()
{
    char x;
    while (true) {
    produce(x);
    append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
      take(x);
      consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Message Passing: when proc interact with another these requirements must be satisfied and msg: synchronization (ensure mut exclu) and comm (to exchange info)

Blocking send, Blocking rec: both sender & rec blocked til msg delivered, allowing for tight synchronization b/w procs

Nonblocking send, blocking rec: sender continues but rec blocked til requested msg arrives. Most useful combo. Sends >= 1 msg to to variety destinations asap

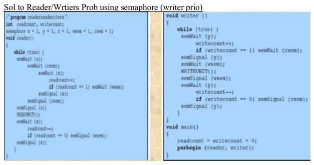
Nonblocking send, nonblocking rec: neither party required to wait

Readers/Writers Prob: data area shared among many procs, following conditions must be satisfied: any # of readers may read file simultaneously, only 1 writer may write at a time, if writer is writing, no reader may read it

Sol to Reader/Wrtiers Prob using semaphore (reader prio)

```
/* program readersandwriters */
int readcout;
semaphore x = 1, wsem = 1;
void reader() {
   while (true) {
      semWait(x);
      readcout++;
      if (readcount == 1) semWait (wsem);
      semSignal(x);
      READUNIT();
      semWait(x);
      readcount--;
      if (readcount == 0) semSignal(wsem);
      semSignal(x);
   }
}
void writer() {
   while (true) {
      semWait(wsem);
      WRITEUNIT();
      semSignal(wsem);
   }
}
void main() {
   readcount = 0;
   parbegin (reader, writer);
}
```

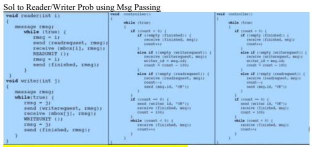
Sol to Reader/Wrtiers Prob using semaphore (writer prio)



```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
   while (true) {
     semWait (z);
        semWait (rsem);
           semWait (x);
              readcount++;
              if (readcount == 1) semWait (wsem);
           semSignal (x);
        semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
     semSignal (x);
   }
}
```

```
void writer ()
{
   while (true) {
     semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
     semSignal (y);
   }
}
void main()
{
   readcount = writecount = 0;
   parbegin (reader, writer);
}
```

Sol to Reader/Writer Prob using Msg Passing



```
void reader(int i)
{
   message rmsg;
   while (true) {
      rmsg = i;
      send (readrequest, rmsg);
      receive (mbox[i], rmsg);
      READUNIT ();
      rmsg = i;
      send (finished, rmsg);
   }
}
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void controller()
{
   while (true)
   {
      if (count > 0) {
         if (!empty (finished)) {
            receive (finished, msg);
            count++;
         }
         else if (!empty (writerequest)) {
            receive (writerequest, msg);
            writer_id = msg.id;
            count = count - 100;
         }
         else if (!empty (readrequest)) {
            receive (readrequest, msg);
            count--;
            send (msg.id, "OK");
         }
      }
      if (count == 0) {
         send (writer_id, "OK");
         receive (finished, msg);
         count = 100;
      }
      while (count < 0) {
         receive (finished, msg);
         count++;
      }
   }
}
```

```
void controller()
{
   while (true)
   {
      if (count > 0) {
         if (!empty (finished)) {
            receive (finished, msg);
            count++;
         }
         else if (!empty (writerequest)) {
            receive (writerequest, msg);
            writer_id = msg.id;
            count = count - 100;
         }
         else if (!empty (readrequest)) {
            receive (readrequest, msg);
            count--;
            send (msg.id, "OK");
         }
      }
      if (count == 0) {
         send (writer_id, "OK");
         receive (finished, msg);
         count = 100;
      }
      while (count < 0) {
         receive (finished, msg);
         count++;
      }
   }
}
```

CHAPTER 6: DEADLOCK AND STARVATION

Deadlock: permanent blocking of set of procs that compete for sys resources or comm w/ each other. Set is deadlocked when each proc in the set is blocked waiting for event that can only be triggered by another blocked process in the set

Resource categories:

Reusable: can be safefly used by only proc at a time & isn't depleted after use (processors, I/O channels, main & secondary mem, devices & data structs)

Consumable: created/produced & destroyed/consumed. Interrs, sigs, msgs, & info in I/O buff

Deadlock Conditions:

- Mutual Exclusion: one lone process may use resource at a time. If access to resource required it, then it must be supported by OS
- Hold-&-Wait: proc may hold allocated resources while awaiting assnt of others. Requires proc to request all required resources at one tie & blocking til all requests can be granted simultaneously
- No pre-emption: no resource can be forcibly removed from proc holding it; if proc holding certain resources is denied further request, it must first release original resources & request them again
- Circular wait: closed chain of procs exists, s.t. each proc hold >=1 resource needed by next proc in chain. Define linear ordering of resource types.

Resource Allocation Graphs:

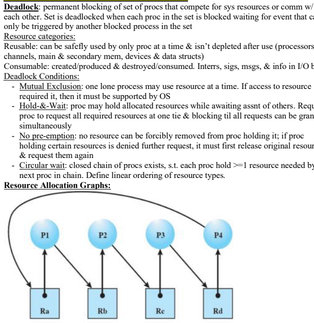
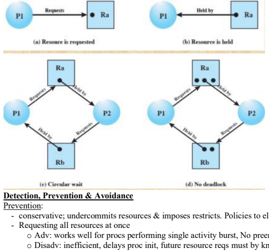


Figure 6.6 Resource Allocation Graph for Figure 6.1b



(a) Resouce is requested

(b) Resource is held

(c) Circular wait

(d) No deadlock

Detection, Prevention & Avoidance

Prevention:

- conservative; undercommits resources & imposes restricts. Policies to elim condition
- Requesting all resources at once
  - Adv: works well for procs performing single activity burst, No preempt needed
  - Disadv: inefficient, delays proc init, future resource reqs must by known
- Prevention Strats:
  - Indirect: prevents occurrence of 1 of 3 necessary conditions
  - Direct: prevent occurrence of circular wait
- Pre-emption
  - Adv: convenient when applied to resources whose state saved & stored easily
  - Disadv: preempts more often than necessary
- Resource ordering
  - Adv: feasible to enfore via compile-time checks, needs no run-time comp since prog solved in sys design
  - Disadv:disallows incremental resource requests

Avoidance: Midway b/w detection & prevention. Make dynamic choices based on state of resource allocation

- Manip to find at least 1 safe path
- Adv: no pre-emp and rollback processes needed.
- Disadv: future resource reqs must be known by OS, procs can be blocked for long periods, proc under consideration must be independent & no sync reqs, fixed # of resources to allocate, no proc may exit while holding resources
- Resource Allocation Denial (Bankers Algo): doesn't grant an incremental resource request to proc if might lead to deadlock
- Process Init Denial: doesn't start proc if its demands might lead to deadlock

Detection: very liberal; requested resources are granted where possible. Detect & take action to recover

- Invokes periodically testing for deadlock
- Adv: never delays proc init, facilitates online handling. Disadv: preempt loss & can consume considerable processor time

Deadlock Detection Algorithm:

1. Mark each process that has a row in the allocation matrix of all zeros.
2. Initialize a temporary vector W equal to the available vector.
3. Find index *i* s.t *i* is unmarked & *i*th row of Q <= W. If not found, terminate
4. If such a row is found, mark process *i* & add the corresponding row of the allocation matrix to W.

Peterson's Algorithm (Correct Solution taken from slides):

Need to observe state of both processes, which has right to insist on entering into CS

```
                        boolean flag[2];
                        int turn;
Process 0                              Process 1
{                                      {
   flag[0] = TRUE;                        flag[1]=TRUE;
   turn = 1;                              turn = 0;
   while (flag[1] && turn == 1)           while (flag[0] && turn == 0)
      /* do nothing */;                      /* do nothing */;
   /*CS*/                                 /*CS*/
   flag[0]=FALSE;                         flag[1]=FALSE;
}                                      }
```

```
Bankers Algo
C = Claim matrix, A = Allo matrix,
R = Resource vector, A = available vector
    [3 2 2]         [1 0 0]           [2 2 2]
C = [6 1 3]     A = [6 1 2]    C-A = [0 0 1]    R = [9 3 6]
    [3 1 4]         [2 1 1]           [1 0 3]
    [4 2 2]         [0 0 2]           [4 2 0]

V = R - A = [9 3 6] - [9 2 5] = [0 1 1]

C-A_row <= V, True, then V = V + A_row
Safe State -> V = R

C-A_1: [2 2 2] > [0 1 1] so fail
C-A_2: [0 0 1] < [0 1 1] so, new V = [0 1 1] + [6 1 2] = [6 2 3]
C-A_3: [1 0 3] < [6 2 3] so, new V = [6 2 3] + [2 1 1] = [8 3 4]
C-A_4: [4 2 0] < [8 3 4] so, new V = [8 3 4] + [0 0 2] = [8 3 6]

Since not all process were finished, go back to failed P1:
C-A_1: [2 2 2] < [8 3 6], so new V = [8 3 6] + [1 0 0]| = [9 3 6]

R = V ? TRUE !
```

Given the following matrices Q and A, and the available vector V, calculate the R vector and run the deadlock detection algorithm to determine the processes that are deadlocked.

$$Q = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad V = [1 \;\; 1 \;\; 0]$$

```
R vector = resource vector. Obtain by adding vectors A+V
R = [2 3 2] + [1 1 0] = [3 4 2]

Check A to find row of all 0's & mark. If no, continue.
Find a row in Q <= allocation vector V.

Applicable to P1:
V = [1 1 0] + [0 1 0] = [1 2 0]
[1 1 0] is original vector V, [1 2 0] is a row from A.

    [0 0 0]       *[0 0 0]
    [0 1 0]        [1 0 0]
Q = [1 0 2]   A =  [0 0 1]|
    [1 1 2]        [1 1 1]
    [1 0 0]        [0 1 0]
Repeat.
Applicable to P2:

    [0 0 0]       *[0 0 0]
    [0 0 0]       *[0 0 0]
Q = [1 0 2]   A =  [0 0 1]
    [1 1 2]        [1 1 1]
    [1 0 0]        [0 1 0]

Repeat.
Applicable to P5:

    [0 0 0]       *[0 0 0]
    [0 0 0]       *[0 0 0]
Q = [1 0 2]   A =  [0 0 1]
    [1 1 2]        [1 1 1]
    [0 0 0]       *[0 1 0]

If cannot repeat, the remaining process >= vector V, deadlocked
```

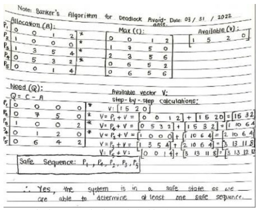
3. **A restaurant has a single employee taking orders and has three seats for its customers. The employee can only serve one customer at a time and each seat can only accommodate one customer at a time. Complete the following function template in a way that guarantees that customers will never have to wait for a seat while holding the food they have just purchased.**

```
semaphore seats = 3;
semaphore employee = 1;
void customer () {
     semWait(&seats);
     semWait(&employee);
     order_food();
     semSignal(&employee);
     eat();
     semSignal(&seats);
} // customer
```

a) Set the initial values of the semaphores **(5 points).**

b) Select the missing instructions after **order_food()** and **eat()** from the following list **(15 points)**:

1. semSignal(&seats);
2. semWait(&employee);
3. semWait(&seats);
4. semSignal(&employee);



Note: Banker's Algorithm for Deadlock Avoidance. Date: 03 / 31 / 2022

	Allocation (A):					Max (C):				Available (V):
P1	0	0	1	2	*	0	0	1	2	[1 5 2 0]
P2	1	0	0	0	*	1	7	5	0	
P3	1	3	5	4	*	2	3	5	6	
P4	0	5	3	2	*	0	6	5	2	
P5	0	0	1	4		0	6	5	6	

Need (Q):
Q = C - A

P1	0	0	0	0	*
P2	0	7	5	0	*
P3	1	0	0	2	*
P4	0	1	2	0	*
P5	0	6	4	2	

Available vector V:
step-by-step calculations:
V: [1 5 2 0]
$V = P_1 + V = [0\ 0\ 1\ 2] + [1\ 5\ 2\ 0] = [1\ 5\ 3\ 2]$
$V = P_4 + V = [0\ 5\ 3\ 2] + [1\ 5\ 3\ 2] = [1\ 10\ 6\ 4]$
$V = P_2 + V = [1\ 0\ 0\ 0] + [1\ 10\ 6\ 4] = [2\ 10\ 6\ 4]$
$V = P_3 + V = [1\ 3\ 5\ 4] + [2\ 10\ 6\ 4] = [3\ 13\ 11\ 8]$
$V = P_5 + V: [0\ 0\ 1\ 4] + [3\ 13\ 11\ 8] = [3\ 13\ 12\ 12]$

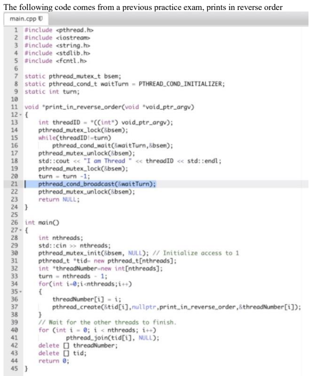
Safe Sequence: P1, P4, P2, P3, P5

∴ Yes, the system is in a safe state as we are able to determine at least one safe sequence.

Note: P5 is also supposed to be marked with a *

This is still bankers, just a different method (don't ask me I didn't do this one)

The following code comes from a previous practice exam, prints in reverse order



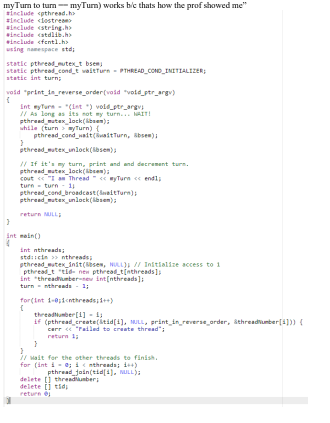
```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t waitTurn = PTHREAD_COND_INITIALIZER;
static int turn;

void *print_in_reverse_order(void *void_ptr_argv)
{
    int threadID = *((int*) void_ptr_argv);
    pthread_mutex_lock(&bsem);
    while(threadID!=turn)
        pthread_cond_wait(&waitTurn,&bsem);
    pthread_mutex_unlock(&bsem);
    std::cout << "I am Thread " << threadID << std::endl;
    pthread_mutex_lock(&bsem);
    turn = turn -1;
    pthread_cond_broadcast(&waitTurn);
    pthread_mutex_unlock(&bsem);
    return NULL;
}

int main()
{
    int nthreads;
    std::cin >> nthreads;
    pthread_mutex_init(&bsem, NULL); // Initialize access to 1
    pthread_t *tid= new pthread_t[nthreads];
    int *threadNumber=new int[nthreads];
    turn = nthreads - 1;
    for(int i=0;i<nthreads;i++)
    {
        threadNumber[i] = i;
        pthread_create(&tid[i],nullptr,print_in_reverse_order,&threadNumber[i]);
    }
    // Wait for the other threads to finish.
    for (int i = 0; i < nthreads; i++)
            pthread_join(tid[i], NULL);
    delete [] threadNumber;
    delete [] tid;
    return 0;
}
```

The following code comes from a prev exam. I was told "test to see if this switch from (turn > myTurn to turn == myTurn) works b/c thats how the prof showed me"



```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
using namespace std;

static pthread_mutex_t bsem;
static pthread_cond_t waitTurn = PTHREAD_COND_INITIALIZER;
static int turn;

void *print_in_reverse_order(void *void_ptr_argv)
{
    int myTurn = *(int *) void_ptr_argv;
    // As long as its not my turn... WAIT!
    pthread_mutex_lock(&bsem);
    while (turn > myTurn) {
        pthread_cond_wait(&waitTurn, &bsem);
    }
    pthread_mutex_unlock(&bsem);

    // If it's my turn, print and and decrement turn.
    pthread_mutex_lock(&bsem);
    cout << "I am Thread " << myTurn << endl;
    turn = turn - 1;
    pthread_cond_broadcast(&waitTurn);
    pthread_mutex_unlock(&bsem);

    return NULL;
}

int main()
{
    int nthreads;
    std::cin >> nthreads;
    pthread_mutex_init(&bsem, NULL); // Initialize access to 1
     pthread_t *tid= new pthread_t[nthreads];
    int *threadNumber=new int[nthreads];
    turn = nthreads - 1;

    for(int i=0;i<nthreads;i++)
    {
        threadNumber[i] = i;
        if (pthread_create(&tid[i], NULL, print_in_reverse_order, &threadNumber[i])) {
            cerr << "Failed to create thread";
            return 1;
        }
    }
    // Wait for the other threads to finish.
    for (int i = 0; i < nthreads; i++)
            pthread_join(tid[i], NULL);
    delete [] threadNumber;
    delete [] tid;
    return 0;
}
```

PRACTICE EXAM PQ: Complete the following C++ program to guarantee that only one person at a time will be in the house, alternating between a Rincon fam mem & a Castro family mem (starting w/ Rincon fam member). Your program will receive from STDIN the # of people (npeople). The # of Rincon fam mems is ceil(npeople / 2) & the # of Castro fam mems is npeople - the # of Rincon fam mems. For npeople = 5, the # of Rincon fam mems is 3 & the # of Castro family mems is 2.

```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER;
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;
static char turn[] = "RINCON";
static bool busy = false;

void *access_one_at_a_time(void *family_void_ptr) {
  pthread_mutex_lock(&bsem);
  char fam[20];
  strcpy(fam,(char *) family_void_ptr);
  while (busy == true || strcmp(fam,turn)!=0) {
    if(strcmp(fam,"RINCON")==0)
         pthread_cond_wait(&rincon, &bsem);
    else
         pthread_cond_wait(&castro, &bsem);
  }
  busy = true;
    std::cout << fam << " member inside the house\n";
    pthread_mutex_unlock(&bsem);
  usleep(100);
    pthread_mutex_lock(&bsem);
    std::cout << fam << " member leaving the house\n";
  busy = false;
    if (strcmp(turn,"RINCON") == 0) {
         strcpy(turn,"CASTRO");
         pthread_cond_signal(&castro);
    }
    else {
         strcpy(turn,"RINCON");
         pthread_cond_signal(&rincon);
    }
         pthread_mutex_unlock(&bsem);
         return NULL;
}

int main() {
  int nmembers;
  std::cin >> nmembers;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
    pthread_t *tid= new pthread_t[nmembers];
    char **family=new char*[nmembers];
  for(int i=0;i<nmembers;i++)
    family[i]=new char[20];
    for(int i=0;i<nmembers;i++) {
         if(i%2 == 0)
              strcpy(family[i],"RINCON");
         else
              strcpy(family[i],"CASTRO");
         if(pthread_create(&tid[i], NULL, access_one_at_a_time,(void *)family[i])) {
              fprintf(stderr, "Error creating thread\n");
              return 1;
         }
    }
    // Wait for the other threads to finish.
    for (int i = 0; i < nmembers; i++)
       pthread_join(tid[i], NULL);
  for(int i=0;i<nmembers;i++)
    delete [] family[i];

  delete [] family;
  delete [] tid;
  return 0;
}
```

```
Bankers Algo
C = Claim matrix, A = Allo matrix,
R = Resource vector, A = available vector
    [3 2 2]        [1 0 0]          [2 2 2]
C = [6 1 3]   A = [6 1 2]   C-A = [0 0 1]   R = [9 3 6]
    [3 1 4]        [2 1 1]          [1 0 3]
    [4 2 2]        [0 0 2]          [4 2 0]

V = R - A = [9 3 6] - [9 2 5] = [0 1 1]

C-A_row <= V, True, then V = V + A_row
Safe State -> V = R

C-A_1: [2 2 2] > [0 1 1] so fail
C-A_2: [0 0 1] < [0 1 1] so, new V = [0 1 1] + [6 1 2] = [6 2 3]
C-A_3: [1 0 3] < [6 2 3] so, new V = [6 2 3] + [2 1 1] = [8 3 4]
C-A_4: [4 2 0] < [8 3 4] so, new V = [8 3 4] + [0 0 2] = [8 3 6]

Since not all process were finished, go back to failed P1:
C-A_1: [2 2 2] < [8 3 6], so new V = [8 3 6] + [1 0 0] = [9 3 6]

R = V ? TRUE !
```

WOLFEY'S SOLUTION

```
#include <cmath>
#include <fcntl.h>
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static pthread_mutex_t bsem;
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER;
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;

bool rinconGo = true;

bool isRincon(char *member) { return strcmp(member, "RINCON") == 0; }

void outputMember(char *member) {
  std::cout << member << " member inside the house" << std::endl;
  std::cout << member << " member leaving the house" << std::endl;
}

void *access_one_at_a_time(void *family_void_ptr) {
  char *member = (char *)family_void_ptr;
  bool am_i_rincon = isRincon(member);

  pthread_mutex_lock(&bsem);
  if (rinconGo != am_i_rincon) {
    pthread_cond_wait(am_i_rincon ? &rincon : &castro, &bsem);
  }

  outputMember(member);
  rinconGo = !rinconGo;
  pthread_cond_signal(am_i_rincon ? &castro : &rincon);
  pthread_mutex_unlock(&bsem);

  return NULL;
}

int main() {
  int nmembers;
  std::cin >> nmembers;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
  pthread_t *tid = new pthread_t[nmembers];
  char **family = new char *[nmembers];
  for (int i = 0; i < nmembers; i++) {
    family[i] = new char[20];
    strcpy(family[i], (i % 2 ? "CASTRO" : "RINCON"));
  }

  for (int i = 0; i < nmembers; i++) {

    pthread_create(&tid[i], NULL, access_one_at_a_time, family[i]);
    // Write the code to call pthread_create
  }

  // Wait for the other threads to finish.
  for (int i = 0; i < nmembers; i++)
    pthread_join(tid[i], NULL);

  for (int i = 0; i < nmembers; i++)
    delete[] family[i];
  delete[] family;
  delete[] tid;
  return 0;
}
```

CHAPTER 5 MUTUAL EXCLUSION

The central themes of operating system design are all concerned with the management of processes and threads. - true
It is possible in a single-processor system to not only interleave the execution of multiple processes but also to overlap them. - false
As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes. - true
Race condition is a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work. - false
The sharing of main memory among processes is useful to permit efficient and close interaction among processes because such sharing does not lead to any problems. - false
When processes cooperate by communication, the various processes participate in a common effort that links all of the processes. - true
Atomicity guarantees isolation from concurrent processes. - true
Concurrent processes do not come into conflict with each other when they are competing for the use of the same resource. - false
Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. - true
The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes. - true
A process that is waiting for access to a critical section does not consume processor time. - false
The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. - true
It is possible for one process to lock the mutex and for another process to unlock it. - false
One of the most common problems faced in concurrent processing is the producer/consumer problem. - true
Processes need to be synchronized to enforce mutual exclusion. – true
The management of multiple processes within a uniprocessor system is __________ .

A) multiprogramming
B) structured applications
C) distributed processing
D) multiprocessing
A situation in which a runnable process is overlooked indefinitely by the scheduler, although it is able to proceed, is _________ .

A) mutual exclusion
B) deadlock
C) starvation
D) livelock -
The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources is ________ .

A) critical section
B) livelock
C) mutual exclusion
D) atomic operation
A means for two processes to exchange information is with the use of _________ .

A) spinlocks
B) event flags
C) condition variables
D) messages
A semaphore that does not specify the order in which processes are removed from the queue is a ________ semaphore.

A) weak
B) general
C) strong
D) binary
A ________ occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

A) atomic operation
B) race condition
C) livelock
D) deadlock
A _________ is an integer value used for signaling among processes.

A) semaphore
B) message
C) mutex
D) atomic operation -

________ is when the sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state.

A) Critical section
B) Mutual exclusion
C) Atomic operation
D) Starvation
________ are memory words used as a synchronization mechanism.

RINCON'S PRACTICE EXAM SOLUTION

```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER;
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;
static char turn[] = "RINCON";
static bool busy = false;

void *access_one_at_a_time(void *family_void_ptr)
{
		pthread_mutex_lock(&bsem);
		char fam[20];
		strcpy(fam,(char *) family_void_ptr);
		while (busy == true || strcmp(fam,turn)!=0)
  {
    if(strcmp(fam,"RINCON")==0)
				pthread_cond_wait(&rincon, &bsem);
    else
      pthread_cond_wait(&castro, &bsem);
  }
  busy = true;
		std::cout << fam << " member inside the house\n";
		pthread_mutex_unlock(&bsem);

  usleep(100);

		pthread_mutex_lock(&bsem);
		std::cout << fam << " member leaving the house\n";
  busy = false;
		if (strcmp(turn,"RINCON") == 0)
  {
    strcpy(turn,"CASTRO");
				pthread_cond_signal(&castro);
  }
  else
  {
    strcpy(turn,"RINCON");
    pthread_cond_signal(&rincon);
  }
		pthread_mutex_unlock(&bsem);
		return NULL;
}

int main()
{
  int nmembers;
  std::cin >> nmembers;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
		pthread_t *tid= new pthread_t[nmembers];
		char **family=new char*[nmembers];
  for(int i=0;i<nmembers;i++)
    family[i]=new char[20];
		for(int i=0;i<nmembers;i++)
		{
		  if(i%2 == 0)
						strcpy(family[i],"RINCON");
				else
						strcpy(family[i],"CASTRO");
		  if(pthread_create(&tid[i], NULL, access_one_at_a_time,(void *)family[i]))
				{
						fprintf(stderr, "Error creating thread\n");
						return 1;

				}

		}
		// Wait for the other threads to finish.
		for (int i = 0; i < nmembers; i++)
		pthread_join(tid[i], NULL);
  for(int i=0;i<nmembers;i++)
    delete [] family[i];
  delete [] family;
  delete [] tid;
		return 0;
}
```

A) Semaphores
B) Event flags
C) Counting semaphores
D) Mailboxes - B. Event flags
The term _______ refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.

A) spin waiting
B) general semaphore
C) critical resource
D) message passing
A _______ is a data type that is used to block a process or thread until a particular condition is true.

A) deadlock
B) general semaphore
C) condition variable
D) mutex -
A semaphore whose definition includes the policy that the process that has been blocked the longest is released from the queue first is called a _________ semaphore.

A) general
B) strong
C) weak
D) counting
The ________ is a programming language construct that provides equivalent functionality to that of semaphores and is easier to control.

A) atomic operation
B) coroutine
C) critical section
D) monitor
Probably the most useful combination, ________ allows a process to send one or more messages to a variety of destinations as quickly as possible.

A) blocking send, blocking receive
B) nonblocking send, blocking receive
C) nonblocking send, nonblocking receive
D) blocking send, nonblocking receive
A ________ relationship allows multiple server processes to provide concurrent service to multiple clients.

A) many-to-many
B) one-to-many
C) many-to-one
D) one-to-one –

RINCON'S SAMPLE CODE OF CONDITION VARIABLE

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define NTHREADS 5
#define FAMILYNAME "RINCON"


static pthread_mutex_t bsem;
static int members;
static pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void *access_house(void *family_void_ptr)
{
		pthread_mutex_lock(&bsem);
		char fam[20];
		strcpy(fam,(char *) family_void_ptr);
		printf("%s member arrives to the house\n", fam);
		if (strcmp(fam,FAMILYNAME)!=0)
				pthread_cond_wait(&empty, &bsem);
		members++;
		printf("%s member inside the house\n", fam);
		pthread_mutex_unlock(&bsem);

		sleep(5);

		pthread_mutex_lock(&bsem);
		printf("%s member leaving the house\n", fam);
		members--;
		if (strcmp(fam,FAMILYNAME) == 0 && members == 0)
				pthread_cond_broadcast(&empty);
		pthread_mutex_unlock(&bsem);
		return NULL;
}

int main()
{

		pthread_t tid[NTHREADS];
		static int members=0;
		char family[NTHREADS][20];
		pthread_mutex_init(&bsem, NULL); // Initialize access to 1

		for(int i=0;i<NTHREADS;i++)
		{
				if(i%2 == 0)
						strcpy((char *) &family[i],"RINCON");
				else
						strcpy((char *)&family[i],"CASTRO");
				if(pthread_create(&tid[i], NULL, access_house,(void *)&family[i]))
				{
						fprintf(stderr, "Error creating thread\n");
						return 1;

				}
				sleep(1);
		}
		// Wait for the other threads to finish.
		for (int i = 0; i < NTHREADS; i++)
		pthread_join(tid[i], NULL);
		return 0;
}
```

CHAPTER 6:DEADLOCK AND STARVATION

Both process interleaving and process overlapping are examples of concurrent processes and both present the same basic problems. - True.
T / F - Concurrency issues are a concern on multiprocessor systems, but do not impact uniprocessor systems. - False, Impact both types of systems.
The following requirement(s) must be met by any facility or capability that is to provide support for mutual exclusion: - 1. Only one process at a time can be allowed into a critical code section
2. A process remains in its critical code section for a finite time only
3. No assumptions can be made about relative process speeds
In a uniprocessor system, mutual exclusion can be guaranteed by: - Disabling Interrupts
The situation where Process 1 (P1) holds Resource 1 (R1), while P2 holds R2, and P1 needs R2 to complete and P2 needs R1 to complete is referred to as ___________. - deadlock.
When only one process is allowed in its critical code section at a time, then __________ is enforced. - mutual exclusion
In multiprocessor configurations, special machine instructions that carry out two actions in a single instruction cycle are said to do so ________. - atomically.
The technique in which a process can do nothing until it gets permission to enter its critical section but continues to test the appropriate varaible to gain entrance is called __________. - busy waiting.
A semaphore that does not specify the order in which processes are removed from the queue is called a: - Weak semaphore
Weak semaphores guarantee freedom from starvation, but strong semaphores do not. - False, opposite is true.
a finite circular buffer and an infinite buffer are two ways to implement a data storage area for the classic Producer / Consumer Problem. - False, only finite buffer.
The major difficulty with semaphores is that wait and signal operations may be scattered throughout a program and it is difficult to see the overall effect of these operations on the semaphores they affect. - True
The classic concurrency problem that involves multiple readers that can read from shared data area when no single writer is exclusively writing to it is the __________ problem. - Readers / Writers
All deadlocks involve conflicting needs for resources by: - Two or more processes
An example of a consumable resource is the following: - Messages
T / F - Although deadlock can potentially exist without it, the condition known as Circular Wait is required for deadlock to actually take place. - I dont have this sheet, but the possibility of deadlock exists without circular wait, but the existence of deadlock means there is a circular wait, so I guess its true?
T / F - The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is minimized. - False
T / F - The Deadlock Avoidance approach to solving the deadlock problem allows the three necessary conditions for deadlock to exist. - True
T / F - In the Resource Allocation Denial approach to Deadlock Avoidance, a safe state is defined as one in which all potential process sequences do not result in a deadlock. - true
T / F - Although each strategy that deals with deadlocks has its advantages and disadvantages, the best solution to the problem is to choose one and stick with it. - False, a combination is best
The Dining Philosopher's Problem is a standard test case for evaluating approaches to implementing: - Synchronization
A programming-language construct that provides equivalent functionality to that of a semaphore and that is easer to control - Monitor
Two fundamental requirements that must be satisfied for process interaction: - -Synchronization and Communication
The problem where a data area is shared among many processes, some processes only read the data area, some only write to the area - Readers/Writers Problem
3 Conditions to satisfy the Readers/Writers Problem - 1. Multiple readers may read the file at once. 2. Only one writer at a time may write. 3. if a writer is writing to the file, no reader may read it.
The condition where only one process may use a resource at a time. - Mutual Exclusion
The condition where a process may hold allocated resources while awaiting assignment of other resources - Hold-and-wait
The condition where no resource can be forcibly removed from a process that is holding it. - No Pre-emption
3 conditions for possible deadlock: - Mutual Exclusion, Hold-and-wait, No Pre-emption
The condition in which a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain. - Circular Wait.
Three approaches for dealing with deadlock: - Prevention, Avoidance, and Detection
This approach to dealing with deadlock eliminates one of more of the conditions required for deadlock to be possible: - Deadlock Prevention
Method of deadlock prevention in which all three necessary conditions are prevented from occuring at once: - Indirect Deadlock Prevention, example: Hole-and-wait - require a process request all of its required resources at once time.
Method of deadlock prevention in which circular waits are prevented - Direct Deadlock Prevention, example: Define linear order of resource types Given {R1, R2} and P(use R1), Q(use R2) Q requests R1 is not possible
Problems with deadlock avoidance: - 1. Inefficient use of resources. 2. Inefficient execuation of processes.
This approach to dealing with deadlock makes dynamic decisions whether the current resource allocation request will, if granted, potentially lead to deadlock - Deadlock Avoidance
This method of deadlock avoidance does not start a process if its demands might lead to deadlock - Process Initiation Denial - A process is only started if the maximum claim of all current processes plus those of the new process can be met. • Not optimal!!!
This method of deadlock avoidance does not grant an incremental resource request to a process if this allocation might lead to deadlock - Resource Allocation Denial - banker's algorithm
Deadlock Avoidance Advantages: - • It is not necessary to preempt and rollback processes, as in deadlock detection
• It is less restrictive than deadlock prevention
Deadlock Avoidance Restrictions: - • Maximum resource requirement must be stated in advance• Processes under consideration must be independent and with no synchronization requirements
• There must be a fixed number of resources to allocate
• No process may exit while holding Resources

USING CONDITION TO PRINT IN CERTAIN ORDER

```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t waitTurn = PTHREAD_COND_INITIALIZER;
static int order;

void *print_in_reverse_order(void *void_ptr_argv)
{

		pthread_mutex_lock(&bsem);
		int *index = (int *) void_ptr_argv;
		while((*index) != order)
		  pthread_cond_wait(&waitTurn, &bsem);

		  std::cout<< "I am Thread " << (*index) << std::endl;
		  order--;
		  pthread_cond_broadcast(&waitTurn);
		  pthread_mutex_unlock(&bsem);
		return NULL;
}

int main()
{
  int nthreads;
  std::cin >> nthreads;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
  order = nthreads - 1;
		pthread_t *tid= new pthread_t[nthreads];
		int number[nthreads];
		int *threadNumber=new int[nthreads];
  for(int i=0;i<nthreads;i++){
    number[i]= i;
    if(pthread_create(&tid[i], NULL, print_in_reverse_order,&number[i]))
				{
						fprintf(stderr, "Error creating thread\n");
						return 1;

				}

  }
		// Wait for the other threads to finish.
		for (int i = 0; i < nthreads; i++)
		pthread_join(tid[i], NULL);

  delete [] threadNumber;
  delete [] tid;
		return 0;
}
```

Pipes a pipe is a technique for passing information from one program process to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only.
pros and cons of pipes Pros:
-Simple
-Flexible
-Efficient Communication
Cons:
-No way to open an already existing pipe. This makes it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process
procedure for using shared memory - find a key
-use shmget() to allocate a shared memory
-use shmat() to attach a shared memory to an address space
-use shmdt() to detach a shared memory from an address space
-use shmctl() to deallocate a shared memory
Message passing Communicating between multiple processors by explicitly sending and receiving information.
requires:
-One send
send(addr, msg.length);
-One receive(addr,msg,length);
Advantages of message passing -Very general
- senders and receivers can be on different machines
-relatively secure
-receiver can inspect the messages it has received before processing them
Disadvantages of message passing -Hard to use
-every data transfer REQUIRES send() and receive()
-Receiving process MUST expect the send()
- might require forking a special thread
shared memory two or more processes share a part of their address space
Advantages of shared memory -Fast and easy to use
BUT
-concurrent accesses to the shared data can result into small disasters
-MUST synchronize access to shared data
Disadvantages of shared memory Not a general solution
-sender and receiver MUST be on same machine
Less secure
-Processes can directly access a part of the address space of other processes
Defining issues of message passing-Direct/indirect communication
-Blocking/non blocking primitives
-exception handling
-quality of service
direct communication -send and receive system calls ALWAYS specify the process as destination or source
-no intermediary between sender and receiver
what must a process know when executing the receive call of direct communication? process must know the identity of all processes that are likely to send a message
-this is a very bad solution for servers as servers have to answer request from arbitrary processes
indirect communication send and receive must specify an intermediary entity as destination or source
Mailbox created by the kernel at request of user process as an intermediary entity for indirect communication
mailbox calls send(mailbox, msg, size);
receive(mailbox,msg,size);
sending and receiving of mailbox-different processes can send messages to same mailbox
-a process can receive message from another process without having to know it
- a process can wait for messages coming from different senders and will answer the first message received
mailboxes can be PRIVATE
-attached to a specific process (Ex. cellphone)
PUBLIC
-attached to system objects (ex. homephone)
Private mailbox -Process that requested its creation and its children are the only process that can receive messages through the mailbox are that process and its children
-Cease to exist when the process that requested the creation terminates
-also called PORTS
Public mailbox -shared by all the processes having the right to receive messages through it
-works best when all processes are on same machine
-survive termination of the process that requested the creation
Blocking send the sender is blocked until the message is received
does not return until the receiving process has received the message
-no buffer needed
ex. when you call someone that doesnt have a voicemail set up
blocking receive-the receiver is blocked until a message is available
-does not return until a message has been received
ex. waiting by phone for an important messages or standing by mailbox all day waiting for mailman
Non-blocking send the sending process sends the message and resumes operation
-returns as soon as message has been accepted by delivery for OS
Non-blocking receive Receiver returns after retrieving either a valid message or null
ex. checking whether your mail has arrived or not
busy wait simulates a block receive with a non blocking receive within a loop
simulating blocking sends with two non-blocking sends -sender sends message and requests an acknowledgement(ACK)
-sender waits for ACK from receiver using a blocking receive
-receiver sends ACK
when it comes to blocking/non blocking, we prefer -Indirect naming
-Non blocking sends
sender does not care what happens after message gets sent
-Blocking receives
receiver needs data to continue
Buffering -required by non blocking primitives to let the OS store messages that have been sent but not yet received somewhere
Bounded capacity when the buffer can take only a certain amount of messages
unlimited capacity when the buffer can recieve unlimited amount of capacity
-buffers are theoretically unlimited capacity
Blocking and receive does not go well with direct communication Processes cannot wait for messages from several sources without using special parallel programming constructs:
Dijkstra's alternative command
using blocking receives with direct naming a bad combination
does not allow the receiving process to recieve any messages from any process except for the one that's specified
exception condition handling- Must specify what to done if one of the two processes dies
-important whenever two processes are on two different machines
must handle:
-host failures
-network partitions
Quality of service when sender and receiver are on different machines -can be lost, corrupted or duplicated
-arrive out of sequence
BUT can still decide to provide reliable message delivery
positive acknowledgement (ACK) -Best technique for providing reliable delivery of messages
-Destination sends an acknowledgement message for every message correctly delivered
-damaged messages are ignored
- sender resends any messages that has not been acknowledged within a fixed time frame
rule of acknowledgement Receiver must acknowledge the second time a message has been sent otherwise it would be resent ONE MORE TIME.
acknowledge any message that does not need to be resent
datagrams messages are sent one at a time
-some messages can be lost, others duplicated or arrive out of sequence
Virtual Circuits -ordered sequence of messages
-connection oriented service
-Require setting up a virtrual connection BEFORE sending data
-costlier than datagrams
-best for transmitting large amounts of data that requires sending several messages
ex. FTP, HTTP
streams -ordered sequence of bytes
-message boundaries are ignored
-similar to virtrual circuits
-do not preserve message boundaries
-receiver sees a seamless stream of bytes
-record boundaries dont count
-messages boundaries should not count
reliable datagrams unsent until they acknowledged
unreliable datagrams -messages are not acknowledged
-works well when message request a reply
-reply is implicit acknowledgment of a message
reason to request is when it might take a long time to reply to it
User Datagram Protocol (UDP) -Best known datagram protocol
-Provides an unreliable datagram service
-best for short interactions (one request and one reply)
Transmission Control Protocol (TCP) -best known stream protocol
-provides a reliable stream service
-heavy weight
requires three messages (packets) to establish a virtual connection
Datagrams vs Streams
Datagrams:
-unreliable
-not ordered
-lightweight
-deliver messages
ex. UDP
Streams:
-reliable
-ordered
-heavyweight
-stream oriented
ex. TCP
busy wait -a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.
Remote Procedure Call (RPC) - -A protocol that enables a process on one computer to call a process on another computer.
-Apply to client-server mode of computation
advantages of RPC - -Hides all details of message passing
Programmer can focus on the logic of her application
-Provides a higher level of abstraction
-Extends a well known programming model
-Anybody that can use procedures and function can quickly learn to use remote procedure calls
Disadvantages of RPC - -The illusion is not perfect
-RPCs do not always behave exactly like regular procedure calls
-Client server do not share the same address space
-Programmer must remain aware of these subtle and not so subtle differences
user program - contains the user code
calls the user stub
rpc(xyz,args, &result);
Appears to call the server procedure
user stub - Procedure generated by RPC package:
-Packs arguments into request message and performs required data conversions (argument marshaling)
-sends request message
-waits for server's reply message
-unpacks the results and performs required data conversions (argument unmarshaling)
argument unmarshaling - Unpacks results and performs required data conversions
the server stub - Generic server generated by RPC package:
-waits for client requests
-unpacks requests argument and performs required data conversions
-calls appropriate server procedure
-packs the results into reply message and performs required data conversions
-send reply message
argument marshaling - Packs arguments into request message and performs required data conversions
server procedure - -Procedure called by the server stub
-written by the user
-Does the actual processing of user requests
Differences with regular PC - -Client and server processes do NOT share the same address space
-Client and server can be on different machines
-Must handle partial failures
No shared address space - means:
No global variables
-Cannot pass addresses
-Cannot pass arguments by reference
-Cannot pass dynamic data structures through pointers
The solution - RPC can pass arguments by value and result
-Pass the current value of the argument to the remote procedure
-Copy the returned value in the user program
-Not the same as passing arguments by reference
Passing dynamic types - -you can NOT pass dynamic data structures through pointers
-must send a copy of the data structure
-For a linked in list:
send array with elements of linked list plus unpacking instructions
little endians - start at the LEAST significant byte
ex- Intel's 80x86 including pentium
Big endians - start at the MOST significant byte
ex. Sun's SPARC and most RISC processors
Solution to architecture considerations - -Define a network order and convert all numerical variables to that order
-use hton family of functions
-Same as requiring all air traffic control communications to be in english
Detecting partial failures - -The client must detect server failures
-Can send "Are you alive?" messages to the server at fixed time intervals
Handling partial executions - -Client must deal with possibility that the server could have crashed after having partially executed the rquest
ex. ATM machine calling the bank computer
was the account debited or not?
first solution to handling partial executions - -Ignore the problem and always resubmit requests that have not been answered
Some requests may be executed more than once
-Will work if requests are idempotent
Executing them several times has the same effect as executing them exactly once
Technique is used by all RPCs in the Sun Microsystems' Network File System (NFS)
first solution to handling partial executions - -Attach a serial number to each request
-Server can detect replays of requests it has previously received and refuse to execute them
-At most once semantics
cheap but not perfect
some requests could end up being partially executed
third solution of handling partial execution - Use a transaction mechanism
-Guarantees that each request will either be fully executed or have no effect
All or nothing semantics
-Best and costliest solution
-use it in all financial transactions
SUN RPC - -Developed by sun microsystems
-used to implement their network file system
MSRPC (Microsoft RPC) - -Proprietary version of the DCE/RPC protocol
-Was used in the Distributed Component Object Model(DCOM)
SOAP - -Exchanges XML- based messages
-Runs on the top of HTTP:
-very portable
-very verbose

JSON-RPC - uses Javascript object notation (JSON)
multiple processes - Operating system design is concerned with the management of processes and threads:
-multiprogramming
-multiprocessing
-Distributed processing
multiple applications - invented to allow processing time to be shared among active applications
Structured applications - an extension of the principles of modular design and structured programming,
some applications can be effectively programmed as a set of concurrent processes.
operating system structure - OS themselves implemented as a set of processes or threads
atomic operation - A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
Critical Section - A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock - a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something
livelock - A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
Mutual Exclusion - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition - -Occurs when multiple processes or threads read and write data items
-The final result depends on the order of execution
The "loser" of the race is the process that updates last and will determine the final value of the variable
starvation - A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
Principles of concurrency - Interleaving and overlapping
- can be viewed as examples of concurrent processing
- both present the same problems

Uniprocessor: the relative speed of execution of processes cannot be predicted
- depends on the activities of other processes
- the way the OS handles interrupts
- scheduling policies of the OS
Difficulties of Concurrency - -Sharing of global resources
-Difficult for the OS to manage the allocation of resources optimally
-Difficult to locate programming errors as results are not deterministic and reproducible
Operating system concerns - -Design and management issues raised by the existence of concurrency:
-The OS must:
be able to keep track of various processes
-allocate and deallocate resources for each active process
-protect the data and physical resources of each process against interference by other processes
-ensure that processes and outputs are independent of the processing speed

resource competition - Concurrent processes come into conflict when they are competing for use of the same resource

In case of the competing processes three control problems must be faced:
-the need for mutual exclusion
-deadlock
-starvation
requirements for mutual exclusion - -Must be enforced
-A process that halts must do so without interfering with other processes
-No deadlock or starvation
-A process must not be denied access to a critical section when there is no other process using it
-No assumptions are made about relative process speeds or number of processes
-A process remains inside its critical section for a finite time only
Interrupt Disabling - -uniprocessor system
-disabling interrupts guarantees mutual exclusion
Disadvantages of Interrupt Disabling - -the efficiency of execution could be noticeably degraded
-this approach will not work in a multiprocessor architecture
Advantages of special machine instruction
-Applicable to any number of processes on either a single processor or multiple processors sharing main memory
-Simple and easy to verify
-Can be used to support multiple critical sections; each critical sections can be defined by its own variable
Disadvantages of special machine instruction –
-Busy waiting is employed, thus while a process is waiting for access to a critical section to continues to consume processor time
-starvation is possible when a process leaves a critical section and more than one process is waiting
-deadlock is possible
semaphore - a variable that has an integer value upon which only three operations are defined -> there is no way to inspect or manipulate sempahores other than these three operations
may be initialized to a nonnegative integer value
semWait operation - decrements the value
semSignal operation - increments the value
consequences of semaphore- 1. There is no way to know before a process decrements a semaphore whether it will block or not
2. There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently
3. You dont know whether another process is waiting so then umber of unblocked processes may be zero or one
strong semaphores - the process that has been blocked the longest is released from the queue first
weak semaphores - the order in which processes are removed from the queue is not specified
queue - used to hold processes waiting on the semaphore
Producer-Consumer Problem - General situation
-one or more producers are generating data and placing these in a buffer
-a single consumer is taking items out of the buffer one at a time
-only one producer or consumer may access the buffer at any one time
The problem
-ensure that the producer cant add data into full buffer and consumer cant remove data from an empty buffer
Implementation of Semaphores - -Imperative that the semWait and semSignal operations be implemented as atomic primitives
-can be implemented in hardware or firmware
-Software schemes such as Dekker's or Peterson's algorithms can be used
-Use one of the hardware-supported schemes for mutual exclusion
Monitors - - programming language construct that provides equivalent functionality to that of sempahores and is easier to control
-Implemented in Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3 and Java
-Has also been implemented as a program library
-Software module been consisting of one or more procedures, an initialization sequence and local data
Monitor Characteristics - -Local data variables are accessible only by the monitor's procedures and not by any external procedure
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time
synchronization - -Achieved by the use of condition variables that are contained within the monitor and accessible within the monitor
Condition variable functions –
cwait
csignal
cwait(c) - suspend execution of calling process on condition c
csignal(c) - resume execution of some process blocked after a cwait on the same condition
Fundamental requirements of message passing - synchronization and communication
synchronization of message passing - to enforce mutual exclusion
communication of message passing - to exchange information
Blocking send, Blocking receive - -Both the sender and receiver are blocked until the message is delivered;
-this is sometimes referred to as a rendezvous .
- This combination allows for tight synchronization between processes.
Nonblocking send, blocking receive - Sender continues on but receiver is blocked until the requested message arrives,
-most useful combination,
sends one or more messages to a variety of destinations as quickly as possible,
example -- a service process that exists to provide a service or resource to other processes
Nonblocking send, nonblocking receive - neither party is required to wait
Direct Addressing - The send primitive includes a specific identifier of the destination process
The receive primitive can be handled in one of two ways:
-require that the process explicitly designate a sending process
effective for cooperating concurrent processes
-implicit addressing
source parameter of the receive primitive possesses a value returned when the receive operation has been performed
Indirect Addressing - 1.Messages are sent to a shared data structure consisting of queues that can temporarily hold messages
2.Queues are referred to as mailboxes,
3. One Process sends a message to the mailbox and the other process picks up the message from the mailbox, 4.Allows for greater flexibility in the use of messages.
Reader/Writer Problem - -A data area is shared among many processes
-some processes only read the data area (readers)
and some only write to the data area (writers)
Conditions that must be satisfied in Readers/Writers Problem –
1. any number of readers may simultaneously read the file
2. only one writer at a time may write to the file
3. if a writer is writing to the file, no reader may read it

DEADLOCK AND STARVATION
Deadlock -The permanent blocking of a set of processes that either communicate for system resources or communicate with each other
-A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
-no efficient solution
-Permanent
Resource Categories reusable and consumable
Reusable Resource Can be safely used by only one process and is not depleted by that use
ex. processors, I/O channels. main and secondary memory, devices and data structures such as files, databases and semaphores
Consumable resource One that can be created (produced) and destroyed (consumed)
-ex Interrupts, signals, messages, and information. inside I/O buffers
Deadlock Detection Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

Conditions for deadlock
Mutual exclusion- only one process may use a resource at a time
Hold and wait- a process may hold allocated resources while awaiting assignment of others
No pre-emption- no resource can be forcibly removed from a process holding it
Circular wait- a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
Deadlock Prevention adopt a policy that eliminates one of the conditions
Deadlock Avoidance make the appropriate dynamic choices based on the current state of resource allocation
requires knowledge of future process requests
Deadlock Detection attempt to detect the presence of deadlock and take action to recover
Deadlock prevention strategy Design a system in such a way that the possibility of deadlock is excluded
two methods: indirect and direct
Direct Deadlock Prevention prevent the occurrence of a circular wait
Indirect Deadlock Prevention prevent the occurrence of one of the three necessary conditions
Conditions of Deadlock prevention
Mutual Exclusion - if access to a resource requires mutual exclusion then it must be supported by the OS
Hold and wait require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
No preemption- if a process holding certain resources is denied a further request, then process must release its original resources and request them again.
OS may preempt the second process and require it to release its resources
Circular Wait define a linear ordering of resource types
Approaches to Deadlock Avoidance Resource Allocation Denial - do not grant an incremental resource request to a process if this allocation might lead to deadlock
Process Initiation Denial - do not start a process if its demands might lead to deadlock
Resource Allocation denial
AKA banker's algorithm
-State of the system reflects the current allocation of resources to processes
Safe state is one in which there is at least one sequence of resources allocations to processes that does not result in a deadlock
unsafe state is a state that is not safe
Deadlock avoidance formula
if (C-A) <= available vector -> allocation + available,

resource vector must equal available vector at end if safe state
Deadlock Detection formula
set work to equal available vector
if Q <= Work -> allocation + work
mark each process after finishing it
Advantages of deadlock avoidance
-It is not necessary to preempt and rollback processes, as in deadlock detection
-It is less restrictive than deadlock prevention
Restrictions of Deadlock Avoidance
-Maximum resource requirement for each process must be stated in advance
-Processes under consideration must be independent and with no synchronization
-There must be a fixed number of resources to allocate
-No process may exit while holding resources
Differences between Deadlock detection and prevention strategies
Deadlock prevention strategies are very conservative, they limit access to resources by imposing restrictions on processes
Deadlock detection strategies do the opposite, resources are granted whenever possible
Advantages of Deadlock detection
-it leads to early detection
-the algorithm is relatively simple
Disadvantages of Deadlock prevention
frequent checks consume considerable processor time
Recovery strategies
- abort all deadlocked processes
- back up each deadlocked process to some previously defined checkpoint and restart all processes
- successively abort deadlocked processes until deadlock no longer exists
- successively preempt resources until deadlock no longer exists

Dining Philosophers Problem The idea that resources must be shared among several processes in a deadlock and starvation free manner
(uses mutual exclusion)