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 to*
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) race, not rare
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.

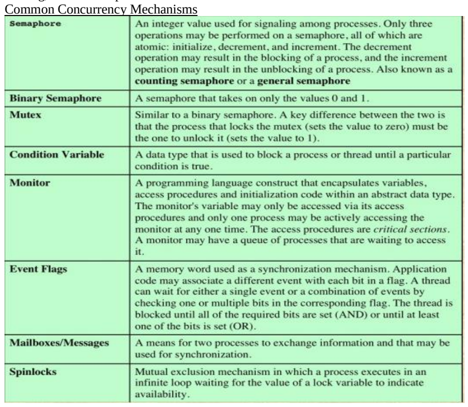
Compare & Swap Instructions:

```
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));
}
```

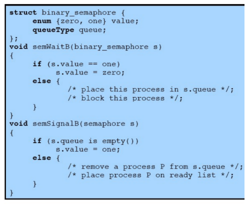
Special Machine Instruct Adv: applicable to any # of pros on single/multiple processors sharing main mem, simple & easy to verify, can be used to supp multi crit sects (each can be defined by its own variable)
Disadv: busy-wait employed, while a proc is waiting for access to ctir section it cont to consume 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 */;
   }
}
```



```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Strong semaphores: the proc that's been blocked the longest released fr queue (FIFO)
Weak semaphores: order where procs removed from queue not specified
A queue is used to hold processes waiting on the semaphore

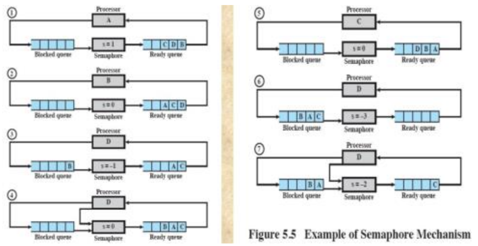


Figure 5.5 Example of Semaphore Mechanism

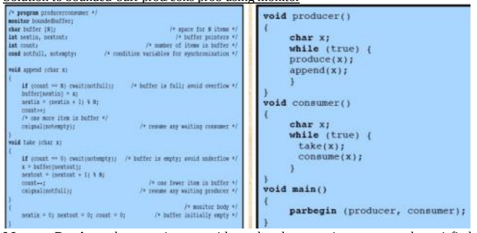
Mutual Exclusion using semaphores

```
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 */
}
```

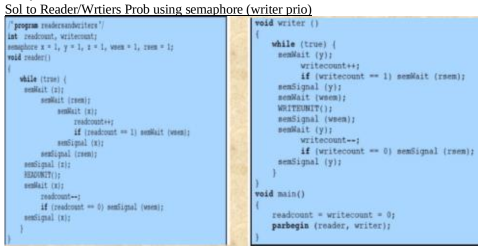
```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
      take(x);
      consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

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);
}
```

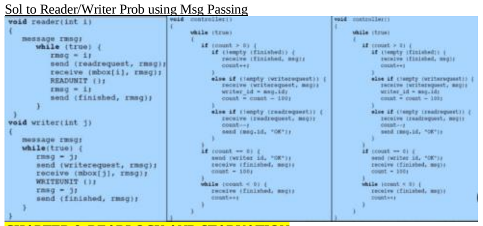
Sol to Reader/Wrtiers Prob using semaphore (writer prio)



```
/*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:

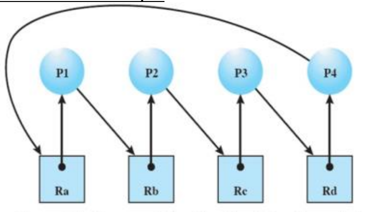
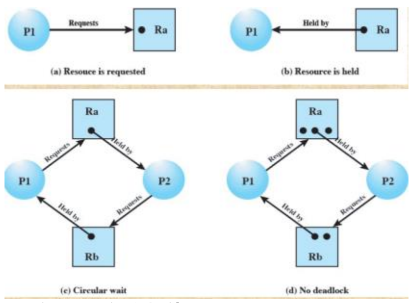


Figure 6.6 Resource Allocation Graph for Figure 6.1b



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.

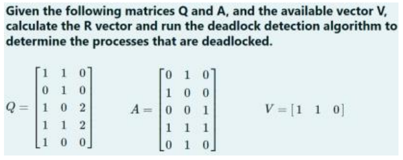
Peterson's Algorithm (Correct Solution taken from slides):

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;
}	}

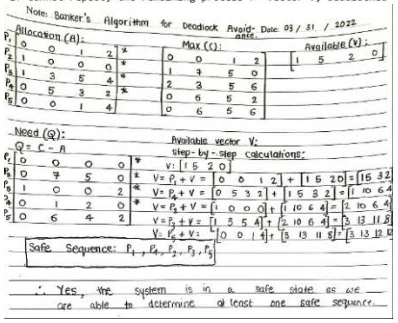
```
Bankers Algo
C = Claim matrix, A = Allo matrix,
R = Resource vector, A = available vector
    [3 2 2]       [1 0 0]         [2 2 2]
C = [6 1 3]   A = [6 1 2]   C-A = [0 0 1]   R = [9 3 6]
    [3 1 4]       [2 1 1]         [1 0 3]
    [4 2 2]       [0 0 2]         [4 2 0]

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.

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

```
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
```



Note: Banker's Algorithm for Deadlock Avoid-ance. Date: 03 / 31 / 2022

Allocation (A):

P1	0	0	1	2	*
P2	1	0	0	0	*
P3	1	3	5	4	*
P4	0	5	3	2	*
P5	0	0	1	4	

Max (C):

0	0	1	2
1	7	5	0
2	3	5	6
0	6	5	2
0	6	5	6

Available (V): [1 5 2 0]

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: P_1, P_4, P_2, P_3, P_5

∴ 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)

----

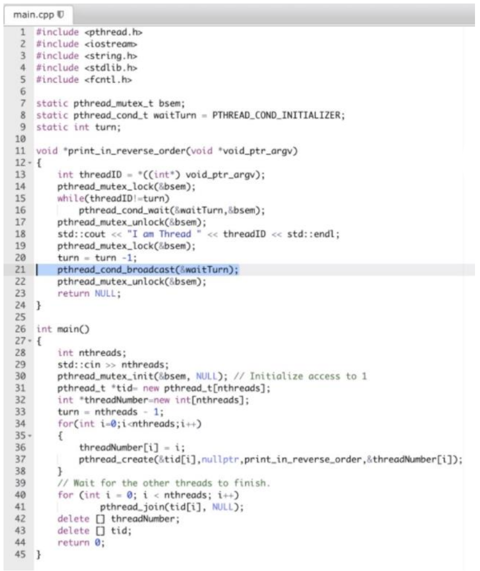
3. A restaurant has a single employee taking orders and has three seats for its customers. The employee can only serve one customer at a time and each seat can only accommodate one customer at a time. Complete the following function template in a way that guarantees that customers will never have to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



main.cpp

```
#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;
}
```

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; // global variable that's gonna be avaliable for both processes
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER; // initializing the pthread conditions for rincon and castro
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;
static char turn[] = "RINCON"; // Initial family to enter
static bool busy = false;

void *access_one_at_a_time(void *family_void_ptr)
{

   pthread_mutex_lock(&bsem); // lock the global variable
   char fam[20];
   strcpy(fam,(char *) family_void_ptr);
   while (busy == true || strcmp(fam,turn)!=0) // while the family is not the same as the current family's turn OR when one family is busy
   {
      if(strcmp(fam,"RINCON")==0) // If the current family is rincon, then wait wait for signal on rincon, same as unlocking bsem, waiting for rincon signal
         pthread_cond_wait(&rincon, &bsem); // then locking bsem again
      else
         pthread_cond_wait(&castro, &bsem); // wait for signal on castro
   }
   busy = true;
   std::cout << fam << " member inside the house\n";
   pthread_mutex_unlock(&bsem);

   usleep(100);

   pthread_mutex_lock(&bsem); // lock the global variable
   std::cout << fam << " member leaving the house\n"; // Family is leaving the house, so they are no longer busy
   busy = false;
   if (strcmp(turn,"RINCON") == 0) // If it was currently Rincon's turn, signal to castro to have their turn and complete their process. Set turn to castro
   {
      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]; // 2D array
   for(int i=0;i<nmembers;i++)
      family[i]=new char[20];
   for(int i=0;i<nmembers;i++)
   {
      if(i %2 == 0) // If it's an even iteration, then it is a thread for rincon. Otherwise, it's a iteration for Castro
         strcpy(family[i],"RINCON");
      else
         strcpy(family[i],"CASTRO");
      if(pthread_create(&tid[i], NULL, access_one_at_a_time,(void *)family[i])) // thread creation
      {
         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;
}
```