CSE 4351/5351 Notes 4: Synchronization: Shared Memory, Message Passing, and Termination Detection

SYNCHRONIZATION IN SHARED MEMORY

Barrier (Wait for N processes to reach):

Version 1: Single-use barrier

Initialize variable x with 0

Check-in:

Fetch-and-add(x,1)
Wait for x=N

Version 2: Wrong multi-use barrier

Check-in:

if Fetch-and-add(x,1) = N-1
    x := 0
else
    Wait for x = 0

x	P1	P2	P3
?	.	.	.
0	Initialize		
1	.	f&a(x,1)=0 ⇒ wait	.
2	f&a(x,1)=1 ⇒ wait	.	.
0	.	.	f&a(x,1)=2 ⇒ GO
0	read x=0 ⇒ GO	.	.
1	.	.	f&a(x,1)=0 ⇒ wait
1	.	read x=1 ⇒ LATE READ	.
2	f&a(x,1)=1 ⇒ wait		

ALL PROCESSES KEEP READING 2

pthreads code - hangs after a few check-ins when ran with three threads on dual-processor, longer for two threads.

```
$ cat version2PT.c
/* pthreads example of incorrect reuse barrier */

#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int x=0;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

void checkIn()
{
int beforeValue;

pthread_mutex_lock(&mutex);
beforeValue=x++;
pthread_mutex_unlock(&mutex);
if (beforeValue==numThreads-1)
  x=0;
else
  while (x!=0)
```

```
    ;
}

void *loop(void *arg)
{
int i,j;
thread_data_t *threadWork=(thread_data_t*) arg;

for (i=1;i<10000000;i++)
{
  if (threadWork->threadId==0)
    printf("%d check-ins\n",i);
  checkIn();
  for (j=0;j<10000000;j++)
    ;
}
}

main()
{
thread_data_t *threadWork;
int i,status;
pthread_t thread;

printf("enter number of threads:");
scanf("%d",&numThreads);

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
              &thread, NULL, loop, threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
loop((void*) threadWork);
}
```

Version 3: Correct Reuse barrier

```
procedure barrierSync;
boolean wasless;  # private variable
{
        wasless := (x<N)
        if Fetch-and-add(x,1) = 2N-1
                x := 0
        Wait for x<N ≠ wasless
}
```

x	P1	P2	P3
?	.	.	.
0	Initialize	.	.
1	.	wasless=YES f&a(x,1)=0 ⇒ WAIT	.
2	wasless=YES f&a(x,1)=1 ⇒ WAIT	.	.
3	.	.	wasless=YES f&a(x,1)=2 ⇒ WAIT
3	.	.	x<3=NO≠YES ⇒ GO
3	x<3=NO≠YES ⇒ GO	.	.
4	.	.	wasless=NO

			f&a(x,1)=3 ⇒ WAIT
4	.	x<3=NO≠YES ⇒ GO	.
5	wasless=NO f&a(x,1)=4⇒ WAIT	.	.
	<waiting for 0>		<waiting for 0>
0	.	wasless=NO f&a(x,1)=5 ⇒ GO	.

pthreads code - runs successfully with arbitrary number of processes.

```
$ cat version3PT.c
/* pthreads example of correct reuse barrier */

#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int x=0;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

void checkIn()
{
int beforeValue,wasless;

wasless=x<numThreads;
pthread_mutex_lock(&mutex);
beforeValue=x++;
pthread_mutex_unlock(&mutex);
if (beforeValue==2*numThreads-1)
  x=0;
else
  while ((x<numThreads)==wasless)
    ;
}

void *loop(void *arg)
{
int i,j;
thread_data_t *threadWork=(thread_data_t*) arg;

for (i=1;i<10000000;i++)
{
  if (i%10==0 && threadWork->threadId==0)
    printf("%d check-ins\n",i);
  checkIn();
  for (j=0;j<100000;j++)
    ;
}
}

main()
{
thread_data_t *threadWork;
int i,status;
pthread_t thread;

printf("enter number of threads:");
scanf("%d",&numThreads);

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
```

```
                &thread, NULL, loop, threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
loop((void*) threadWork);
}
```

Very Fast Two-Thread Barrier

```
int barrier[2]={0,0};  // These act as semaphores

void barrier_wait(int thread)
{
if (thread)
{
  while (!barrier[0])
    ;
  barrier[0]=0;
  barrier[1]=1;
}
else
{
  barrier[0]=1;
  while (!barrier[1])
    ;
  barrier[1]=0;
}
}
```

Parallel Run Queue (highly concurrent)

1. Shared circular array Q[0..size-1]
2. I = Insert Pointer
3. D = Delete Pointer
4. Initially I = D = 0
5. #Qu = number of occupied (or soon-to-be) queue slots
6. #Ql = number of queue slots that have data and are not yet claimed

INSERT	DELETE
if #Qu ≥ size	if #Ql ≤ 0
<Full Queue>	<Empty>
if f&a(#Qu,1) ≥ size	if f&a(#Ql,-1) ≤ 0
f&a(#Qu,-1)	f&a(#Ql,1)
<Full Queue>	<Empty>
MyI := f&a(I,1) mod size	MyD := f&a(D,1) mod size
P(InsertSem[MyI])	P(DeleteSem[MyD])
Q[MyI] := Data	Ans := Q[MyD]
V(DeleteSem[MyI])	V(InsertSem[MyD])
f&a(#Ql,1)	f&a(#Qu,-1)

InsertSem's are initialized to 1, DeleteSem's are initialized to 0

CONCURRENT DATA STRUCTURES:

Issues:

1. Semantics of operations on structure: When are keys to be visible to operations, e.g. under deletion and insertion.
2. Types of locks, lock compatibility.
3. Race conditions during restructuring operations.

4. Lock overhead: Space of locks in structure (low-level implementation to reduce this), lock duration. THIS DECIDES WHETHER CONCURRENT APPROACH IS BETTER THAN SERIALIZING ACCESSES USING A SINGLE LOCK.

5. Proportion of execution-time spent in data structure.

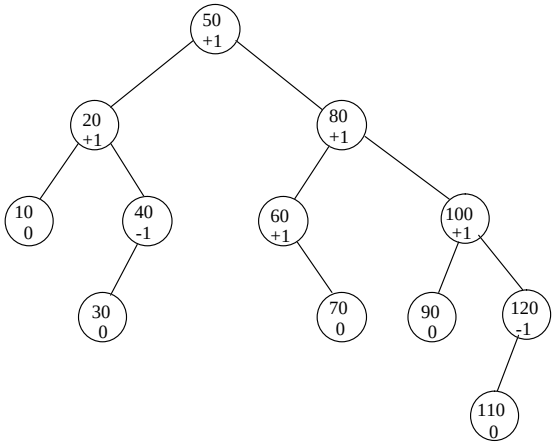
6. Deadlocks.

IMPORTANT CASE: Binary Search Trees

AVL Trees:

1. Each node stores the difference of the heights (known as the balance factor) of the right and left subtrees rooted by the children:

$$height_{right} - height_{left}$$



2. A balance factor must be +1, 0, -1 (leans right, "balanced", leans left).

3. An insert is implemented by:

   a. Attaching a leaf

   b. Rippling changes to balance factor:

      1. Right child ripple

         Parent.Bal = 0 $\Longrightarrow$ +1 and ripple to parent

         Parent.Bal = -1 $\Longrightarrow$ 0 to complete insertion

         Parent.Bal = +1 $\Longrightarrow$ +2 and ROTATION to complete insertion

      2. Left child ripple

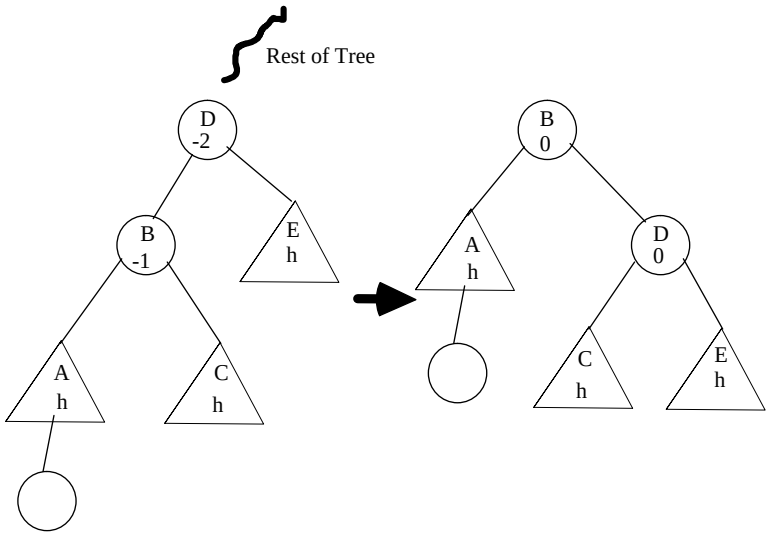
         Parent.Bal = 0 $\Longrightarrow$ -1 and ripple to parent

         Parent.Bal = +1 $\Longrightarrow$ 0 to complete insertion

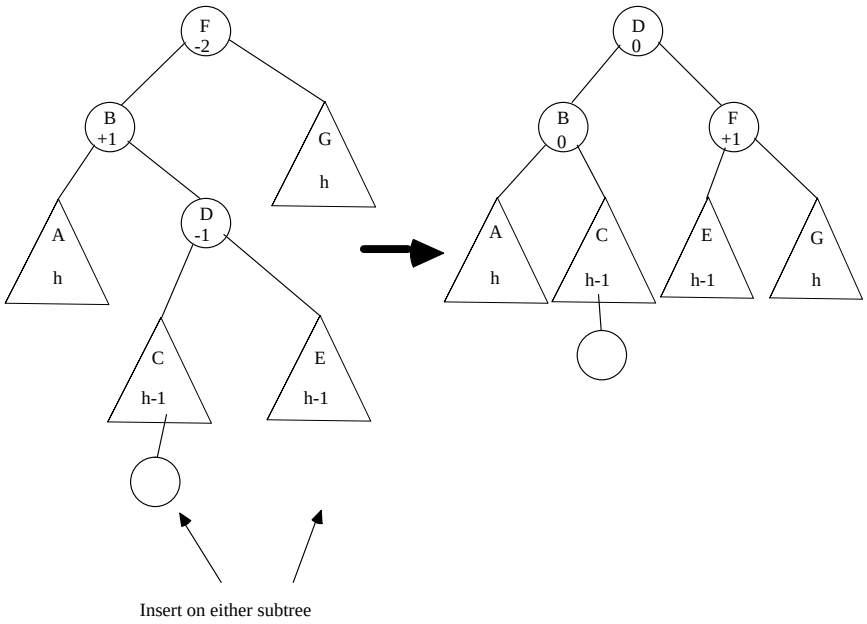
         Parent.Bal = -1 $\Longrightarrow$ -2 and ROTATION to complete insertion

4. Rotations

   a. Single (LL)



b. Double (LR)



c. RR - symmetric to LL

d. RL - symmetric to LR

CONCURRENT VERSION OF AVL TREES

Difficulty - blocking simultaneous pointer changes

Add 3 lock fields

ρ = Required before examining node contents

α = Keeps other insertion processes away from path by including rotation point through insertion point

ξ = Keeps search and insertion processes away from rotation nodes

Lock compatibility matrix:

Already Held \ Another Process Requests	ρ	α	ξ
ρ	√	√	×
α	√	×	×
ξ	×	×	×

Operations:

Search

1. Based on usual search of binary search tree.

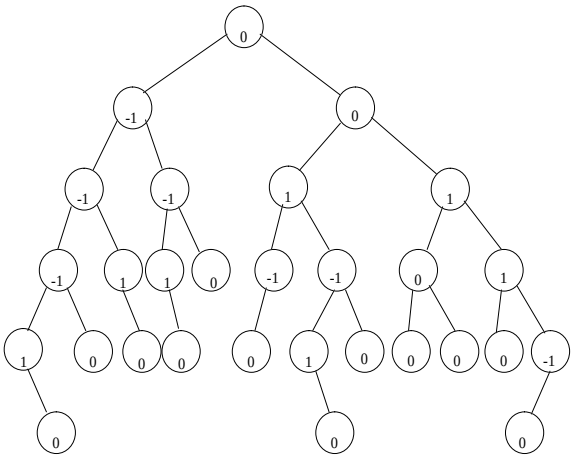
2. Before examining node contents, lock ρ.

3. After examining node contents, unlock ρ.

Insert

1. Based on usual insert into binary search tree.

2. Set α locks going down tree on every node with a balance factor of 0. Release ancestor locks (except parent) when a node that will terminate balance factor change propagation is encountered. Parent α lock is held because a rotation (and parent pointer change) may occur in the child.

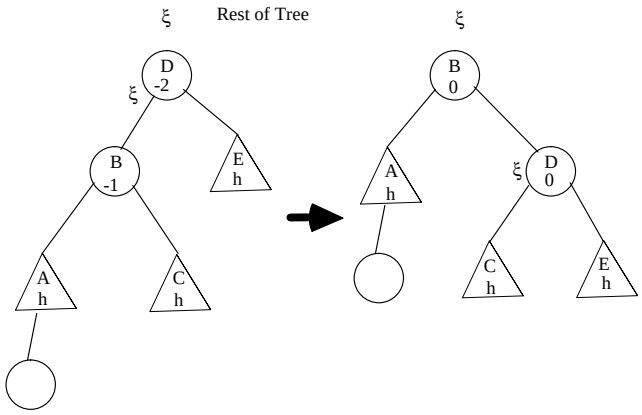


3. Attach node for new key.

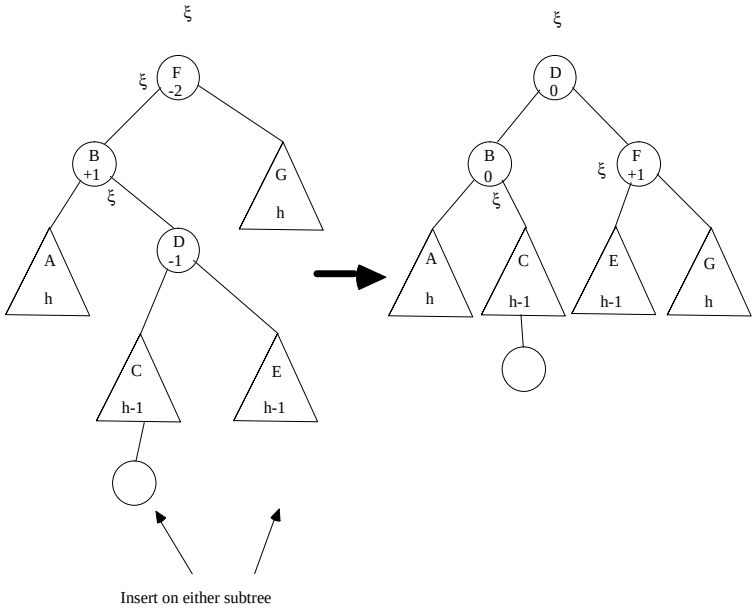
4. Ripple balance factor changes.

5. Locking for rotation:

Single (LL): ξ lock -2 balance factor node (D) and its parent



Double (LR): ξ lock -2 balance factor node (F), its parent and child of rotated node (B)



6. Perform rotation
7. Release locks

Why this approach works: Go down each column of compatibility matrix. Compatibilities first

1. Readers do not conflict: compat[ρ,ρ]

2. Reader looking at potential change in balance factor node is unimportant: compat[α, ρ]

3. Inserter may use a read-locked node since read does not care: compat[ρ, α]

Incompatibilities:

4. compat[ξ, ρ] - indicates unsafe for read access since pointers are changing

Single rotation:

Parent of D, pointer changes to B, so D temporarily "disappears"

D, pointer changes to C, so B temporarily "disappears"

B is safe, possible to search through D twice, but all keys can still be found

Double rotation:

Parent of F, pointer changes to D, so F temporarily "disappears"

F, pointer changes to E, so B temporarily "disappears"

B, pointer changed to C, so D temporarily "disappears"

D is safe, possible to search B or F twice

5. compat[α, α] - Avoid coordinating simultaneous balance factor ripples

6. compat[ξ, α] - Cannot change balance factor while node is moving

7. compat[ρ, ξ] - Let read complete before changing

8. compat[α, ξ] - Cannot happen

9. compat[ξ, ξ] - Cannot happen

Problems:

1. Contention at root/shallow levels.

2. No deletion

Code (avlSpinXiPT.c) - For sake of details in using locking primitives

```
/* xi lock is NOT a field in node.  Instead, the
   mutex is held during the rotation */
/*Ellis's concurrent AVL tree algorithms*/
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

int numThreads;
int numKeys;

typedef struct thread_data {
  int threadId;
} thread_data_t;

barrier_t barrier;

typedef char BOOLEAN;
#define TRUE 1
#define FALSE 0

typedef struct node *ref;
struct node
{
  int key;
  pthread_mutex_t lock;
  char rhoCount;
  BOOLEAN alphaLock;
  ref left,right;
  int bal;
};
typedef struct node treenode;

treenode root;
int key;

int lastkey;  /*for printtree*/

float elapsedCPU()
{
struct rusage rusage;
getrusage(RUSAGE_SELF,&rusage);
return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
  + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

setRho(p)
ref p;
```

```
{
while (pthread_mutex_trylock(&p->lock)==EBUSY)
  ;
p->rhoCount++;
pthread_mutex_unlock(&p->lock);
}

clearRho(p)
ref p;
{
while (pthread_mutex_trylock(&p->lock)==EBUSY)
  ;
p->rhoCount--;
pthread_mutex_unlock(&p->lock);
}

setAlpha(p)
ref p;
{
while (1)
{
  while (pthread_mutex_trylock(&p->lock)==EBUSY)
    ;
  if (!(p->alphaLock))
  {
    p->alphaLock=TRUE;
    pthread_mutex_unlock(&p->lock);
    return;
  }
  pthread_mutex_unlock(&p->lock);
}
}

clearAlpha(p)
ref p;
{
while (pthread_mutex_trylock(&p->lock)==EBUSY)
  ;
p->alphaLock=FALSE;
pthread_mutex_unlock(&p->lock);
}

setXi(p)
ref p;
{
while (1)
{
  while (pthread_mutex_trylock(&p->lock)==EBUSY)
    ;
  if (!(p->rhoCount))
  {
    return;
  }
  pthread_mutex_unlock(&p->lock);
}
}

clearXi(p)
ref p;
{
pthread_mutex_unlock(&p->lock);
}

BOOLEAN search(x)
int x;
{
ref current,son;

setRho(&root);
current=(&root);
son=root.left;
while (son && x!=son->key)
{
  setRho(son);
  clearRho(current);
  current=son;
  son=(x<current->key) ? current->left : current->right;
}
clearRho(current);
return son ? TRUE : FALSE;
}

BOOLEAN insert(x)
int x;
{
ref current,son,cn,fatherOfCn,p,p1,p2;
ref alphaLocked[200];
int alphaPt,i;

setAlpha(&root);
current=(&root);
alphaLocked[alphaPt=0]=current;
fatherOfCn=current;
```

```
son=root.left;
cn=root.left;
while (son && x!=son->key)
{
  setAlpha(son);
  alphaLocked[++alphaPt]=son;
  if (son->bal!=0)
  {
    fatherOfCn=current;
    cn=son;
    for (i=0;alphaLocked[i]!=current;i++)
      clearAlpha(alphaLocked[i]);
    alphaLocked[0]=alphaLocked[alphaPt-1];
    alphaLocked[1]=alphaLocked[alphaPt];
    alphaPt=1;
  }
  current=son;
  son=(x<current->key) ? current->left : current->right;
}
if (son)
{
  for (i=0;i<=alphaPt;i++)
    clearAlpha(alphaLocked[i]);
  return FALSE;
}
p=(ref) malloc(sizeof(treenode));
if (!p) abort();
p->key=x;
pthread_mutex_init(&p->lock,NULL);
p->rhoCount=0;
p->alphaLock=FALSE;
p->left=NULL;
p->right=NULL;
p->bal=0;
if (!cn)
{
  root.left=p;
  clearAlpha(&root);
  return TRUE;
}
if (x<current->key)
  current->left=p;
else
  current->right=p;
for (i=2;i<=alphaPt;i++)
  alphaLocked[i]->bal=(x<alphaLocked[i]->key) ? (-1) : 1;
cn->bal+=(x<cn->key) ? (-1) : 1;
if (cn->bal==(-2))
  if (cn->left->bal==(-1))
  /* LL single rotation */
  {
    setXi(fatherOfCn);
    setXi(cn);
    p1=cn->left;
    cn->left=p1->right;
    p1->right=cn;
    cn->bal=0;
    if (x<fatherOfCn->key)
      fatherOfCn->left=p1;
    else
      fatherOfCn->right=p1;
    p1->bal=0;
    clearXi(fatherOfCn);
    clearXi(cn);
  }
  else
  /* LR double rotation */
  {
    setXi(fatherOfCn);
    setXi(cn);
    setXi(cn->left);
    p1=cn->left;
    p2=p1->right;
    p1->right=p2->left;
    p2->left=p1;
    cn->left=p2->right;
    p2->right=cn;
    cn->bal=(p2->bal==(-1)) ? 1 : 0;
    p1->bal=(p2->bal==1) ? (-1) : 0;
    if (x<fatherOfCn->key)
      fatherOfCn->left=p2;
    else
      fatherOfCn->right=p2;
    p2->bal=0;
    clearXi(fatherOfCn);
    clearXi(cn);
    clearXi(p1);
  }
else if (cn->bal==2)
  if (cn->right->bal==1)
  /* RR single rotation */
  {
    setXi(fatherOfCn);
```

```
    setXi(cn);
    p1=cn->right;
    cn->right=p1->left;
    p1->left=cn;
    cn->bal=0;
    if (x<fatherOfCn->key)
      fatherOfCn->left=p1;
    else
      fatherOfCn->right=p1;
    p1->bal=0;
    clearXi(fatherOfCn);
    clearXi(cn);
  }
  else
  /* RL double rotation */
  {
    setXi(fatherOfCn);
    setXi(cn);
    setXi(cn->right);
    p1=cn->right;
    p2=p1->left;
    p1->left=p2->right;
    p2->right=p1;
    cn->right=p2->left;
    p2->left=cn;
    cn->bal=(p2->bal==1) ? (-1) : 0;
    p1->bal=(p2->bal==(-1)) ? 1 : 0;
    if (x<fatherOfCn->key)
      fatherOfCn->left=p2;
    else
      fatherOfCn->right=p2;
    p2->bal=0;
    clearXi(fatherOfCn);
    clearXi(cn);
    clearXi(p1);
  }
for (i=0;i<=alphaPt;i++)
  clearAlpha(alphaLocked[i]);
return TRUE;
}

int printtree(pt,indent)
ref pt;
int indent;
/*Do not call during parallel execution*/
{
int depthl,depthr;
int i;
if (pt)
{
  depthr=printtree(pt->right,indent+1);

/*
  for (i=0;i<indent;i++)
    printf("   ");
*/

  if (pt->key>lastkey)
    printf("ORDERING ERROR\n");
  lastkey=pt->key;

/*
  printf("%d %d\n",pt->key,pt->bal);
*/

  depthl=printtree(pt->left,indent+1);
  if (pt->bal != depthr-depthl)
    printf("BF wrong at %d\n",pt->key);
  return 1 + ((depthr>depthl) ? depthr : depthl);
}
return 0;
}

void *driver(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
float startCPU;

startCPU=elapsedCPU();
for (i=threadWork->threadId*numKeys/numThreads;
    i<(threadWork->threadId+1)*numKeys/numThreads;i++)
  if (!insert(i))
    printf("Insert failed %d\n",i);
  else if (!search(i))
    printf("Search failed %d\n",i);
barrier_wait(&barrier);
printf("thread %d  used %f\n",threadWork->threadId,
  elapsedCPU()-startCPU);
}

main()
{
```

```
int status,i;
thread_data_t *threadWork;
pthread_t thread;

printf("How many keys?\n");
scanf("%d",&numKeys);
printf("How many threads?\n");
scanf("%d",&numThreads);
fflush(stdout);
fflush(stdin);
root.key=999999999;
pthread_mutex_init(&root.lock,NULL);
root.rhoCount=0;
root.alphaLock=FALSE;
root.bal=0;
root.left=NULL;
root.right=NULL;

barrier_init (&barrier, numThreads);
for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
               &thread, NULL, driver, threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
driver((void*) threadWork);
/*
lastkey=9999999999;
printtree(root.left,0);
*/
}
```

Monitors - Higher-level synchronization mechanism to support data abstractions

Key Ideas:

1. Only one process at a time may be executing within monitor. Other calls to monitor procedures wait in the *entry queue*.

2. A process may become blocked (*waiting*) within a monitor and thus temporarily relinquishes the monitor until a *signal* occurs. Signals are <u>not</u> saved.

3. Waits and signals are named as *condition variables* that act as fifo or priority queues (*minrank* in Andrews).

4. Signalling may be either non-preemptive (*signal-and-continue*) or preemptive (*signal-and-wait*). These are not interchangeable. (pthreads uses signal-and-continue.)

5. ‘‘Passing the condition’’ - signal is used to reflect a significant change in the state of a monitor - signals are never discarded.

Simple example - semaphore:

```
monitor Semaphore {
        int s=0;
        cond pos;

        procedure Psem() {
                while (s==0) wait(pos);
                s--;
        }

        procedure Vsem() {
                s++;
                signal(pos);
        }
}
```

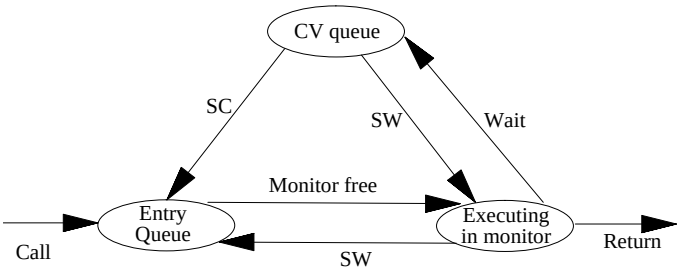
Semaphore using passing the condition:

```
monitor FIFOsemaphore {
        int s=0;
        cond pos;

        procedure Psem() {
                if (s==0)
                        wait(pos);
                else
                        s--;
        }

        procedure Vsem() {
                if (empty(pos))
                        s++;
                else
                        signal(pos);
        }
}
```

Readers/writers problem:

```
monitor RW_Controller {
        int nr=0, mw=0;
        cond oktoread;
        cond oktowrite:

        procedure request_read() {
                while (nw>0)
                        wait(oktoread);
                nr++;
        }

        procedure release_read() {
                nr--;
                if (nr==0)
                        signal(oktowrite);
        }

        procedure request_write() {
                while (nr>0 || nw>0)
                        wait(oktowrite);
                nw++;
        }

        procedure release_write() {
                nw--;
                signal(oktowrite);
                signal_all(oktoread);
        }
}
```

State diagram for monitors



Monitors in pthreads:

A monitor requires a lock along with one or more condition variables.
A thread must hold the lock before calling `pthread_cond_wait()`.
`pthread_cond_wait()` will release the lock, but the lock is restored when `pthread_cond_signal` succeeds.

```
cat bufferPT.c
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <pthread.h>

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv1=PTHREAD_COND_INITIALIZER;
pthread_cond_t cv2=PTHREAD_COND_INITIALIZER;

char x[81];

void *echo(void *arg)
{
while (1)
{
  pthread_mutex_lock(&mutex);
  pthread_cond_wait(&cv1,&mutex);
  pthread_mutex_unlock(&mutex);
  printf("echo 1: %s\n",x);
  printf("enter 1: ");
  fflush(stdout);
  scanf("%s",x);
  pthread_cond_signal(&cv2);
}
}

main()
{
pthread_t thread;
int status;

status=pthread_create(&thread,NULL,echo,NULL);
if (status != 0)
{
  printf("failed to create thread\n");
  exit(0);
}

while (1)
{
  printf("enter 0: ");
  fflush(stdout);
  scanf("%s",x);
  pthread_cond_signal(&cv1);
  pthread_mutex_lock(&mutex);
  pthread_cond_wait(&cv2,&mutex);
  pthread_mutex_unlock(&mutex);
  printf("echo 0: %s\n",x);
}
}
```

SYNCHRONIZATION BY MESSAGE-PASSING

There are four "general" forms of message passing between two processes ("sender" and "receiver") as exemplified in the SR (Synchronizing Resources) programming language:

1. (Remote) procedure call - sender issues call, receiver declares queue as proc

    a. sender sends argument list to a queue for the procedure and waits ("blocks")
    b. receiver with procedure's implementation receives from sender, processes request and sends return value(s)
    c. sender receives return values and continues

2. Dynamic process creation - sender issues send, receiver declares queue as proc

    a. sender sends argument list to a queue for the procedure without blocking
    b. receiver with procedure's implementation receives from sender and creates a process (actually a thread)
    c. thread is deleted upon proc exit

3. Asynchronous message messing - sender issues send, receiver removes message from queue using receive or in

    a. sender sends argument list to a queue without blocking
    b. receiving process uses receive or in to remove message from queue

4. Rendezvous (generalized synchronous message passing) - sender issues call, receiver removes message from queue using receive or in

    a. sender process is executing
    b. receiver process is executing
    c1. sender issues call with argument list and blocks waiting for a receiver
    c2. receiving process uses receive or in to remove message from queue, waits if nothing there
    d. both processes continue after receive or in is completely processed

Examples:

Dynamic process creation:

```
resource quick()
  op sort(var a[1:*]: int)
  var n: int
  # read in n
  getarg(1,n)
  var a[1:n]: int
  # read in data to be sorted
  fa i := 1 to n -> read(a[i]) af
  write("input:"); fa i := 1 to n -> write(a[i]) af
  sort(a)
  write("sorted:"); fa i := 1 to n -> write(a[i]) af

  proc sort(a)
    if ub(a) <= 1 -> write("small array", ub(a)); return fi
    fa i := 1 to ub(a) -> writes(a[i], " ") af; write()
    var pivot := a[1]
    var lx := 2, rx := ub(a)
    do lx <= rx ->
        if a[lx] <= pivot -> lx++
        [] a[lx] >  pivot -> a[lx] :=: a[rx]; rx--
        fi
    od
    a[rx] :=: a[1]
    co sort(a[1:rx-1]) // sort(a[lx:ub(a)]) oc
  end
end quick
```

Asynchronous Message Passing

Simple semaphore:

```
resource CS()
  const N := 20      # number of processes
  op mutex() {send}  # mutual exclusion for x
  send mutex()       # initialize mutex
  var x := 0         # shared variable
  process p(i := 1 to N)
    # non-critical section
    # critical section
    receive mutex()  # enter critical section
    x := x+1;
    write(x," by process ",i);
    send mutex()     # exit critical section
    # non-critical section
  end
end
```

Buffer pool:

```
resource main()
  op pool(index: int)
  const B := 20       # number of buffers
  const N := 10       # number of processes
  var buffer[1:B]: T  # T is the buffer type
  fa i := 1 to B -> send pool(i) af
  process p(i := 1 to N)
     ...
    receive pool(x)   # request a buffer
    # use buffer[x]
    send pool(x)      # release the buffer
     ...
  end
end
```

Barrier using SR's built-in semaphores

```
resource barrier()
  const N := 20  # number of processes
  sem done := 0, continue[N] := ([N] 0)
  # declaration of variables shared by workers
  process worker(i := 1 to N)
    do true ->
      # code to implement one iteration of task i
      V(done)
      P(continue[i])
    od
  end

process coordinator
    do true ->
      fa w := 1 to N -> P(done) af
      fa w := 1 to N -> V(continue[w]) af
    od
  end
end
```

Rendezvous:

```
resource main()
  op swap(var x: int)
  process p1
    ...
    call swap(y)
    ...
  end
  process p2
    ...
    call swap(z)
    ...
  end
  process q
    ...
    in swap(x1) -> in swap(x2) -> x1 :=: x2 ni ni
    ...
  end
end
```

Termination Detection

In a distributed message-passing situation, it is useful to determine when a computation has reached a *quiescent state*. In particular, it must be determined that 1) all processes are idle and 2) there are no messages in transit or residing in any queue. Numerous solutions have been proposed (see the paper by Matocha and Camp on the course web page) using a variety of notions to address various needs. To simplify the presentation, we will assume that each process has a single thread of control (like MPI) that regularly blocks to wait for a message. Two solutions will be examined: Mattern's credit-recovery technique and the often-referenced Dijkstra-Scholten tree technique.

Mattern's Credit-recovery Technique

Each process in this technique maintains a *credit value*. Initially, one process (called the environment) has a credit value of 1, while all other processes have a credit value of 0. Whenever a process sends a message to another process, a fraction of the credit value is then sent to the receiving process to add to its credit value. Just before a process becomes idle it will send its entire credit value back to the environment. Of course, dealing with arbitrary fractions can be difficult and leads to variations in the implementation.

The provided MPI code represents the fraction $1/2^k$ by simply saving k. All processes, except the environment, only deal with credit values that are negative powers of 2. The environment, however, must deal with an arbitrary binary fraction. When the computation is initiated at the root by sending p messages, MPI processes 1 through p receive the credit values 2^{-1}, 2^{-2}, . . ., 2^{p-1}, 2^{p-1} which add up to 1. Each time a process (besides the environment) issues a message, half the credit value is sent to the destination while half is kept for itself. Again, when a process becomes idle its entire credit will be sent to the environment where the credit is subtracted from a ‘‘debt’’ maintained in the environment. When the environment's debt becomes 0, the distributed computation has terminated.

```
// MPI for Mattern's termination detection using credit shares.
// See Information Processing Letters, 30 (4), Feb 27, 1989, 195-200,
// especially section 3. A realization of the principle
// BPW, 5/2001

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

MPI_Status status;
int rank,numProcesses;

#define ACTIVATION 1
#define GONEPASSIVE 2
#define GLOBALTERMINATION 3

#define MAXDEBTS 20000

int generateRandom(minRange,maxRange)
int minRange,maxRange;
{
/* returns integer in range minRange <= x <= maxRange*/
return minRange+abs(random()) % (maxRange-minRange+1);
}
```

```
void rootProcess(int iterNumber,int descendants)
{
int i,j,k,cr;
char debts[MAXDEBTS+1];
int debtsCount=1;

printf("0: round %d started\n",iterNumber);
for (i=1;i<descendants;i++)
  MPI_Send(&i,1,MPI_INT,i,ACTIVATION,MPI_COMM_WORLD);
k=descendants-1;
MPI_Send(&k,1,MPI_INT,descendants,ACTIVATION,MPI_COMM_WORLD);
debts[0]=1;
for (i=1;i<=MAXDEBTS;i++)
  debts[i]=0;

while (debtsCount!=0)
{
  MPI_Recv(&cr,1,MPI_INT,MPI_ANY_SOURCE,GONEPASSIVE,MPI_COMM_WORLD,&status);
  k=cr;
  if (k>MAXDEBTS)
  {
    printf("0(%d): MAXDEPTS too small!\n",__LINE__);
    MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
  while (debts[k]==0)
  {
    debts[k]=1;
    k--;
    debtsCount++;
  }
  debts[k]=0;
  debtsCount--;
}
for (i=1;i<numProcesses;i++)
  MPI_Send(&iterNumber,1,MPI_INT,i,GLOBALTERMINATION,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
printf("0: round %d completed\n",iterNumber);
}

void slaveProcess(int iterNumber,float prob)
{
int i,k,credit=0,cr;

//printf("%d: round %d started\n",rank,iterNumber);
do
{
  MPI_Recv(&cr,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  switch (status.MPI_TAG)
  {
    case (ACTIVATION):
      credit=cr;
      // Randomly diffuse (or replace with real work that may diffuse)
      while (1)
      {
        k=generateRandom(0,100000);
        if (k>prob*100000)
          break;
        k=generateRandom(1,numProcesses-1);
        credit++;
        MPI_Send(&credit,1,MPI_INT,k,ACTIVATION,MPI_COMM_WORLD);
      }
      MPI_Send(&credit,1,MPI_INT,0,GONEPASSIVE,MPI_COMM_WORLD);
      credit=0;
      break;
    case (GLOBALTERMINATION):
      break;
    default:
      printf("%d(%d): bad slaveProcess request\n",rank,__LINE__);
      MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
} while (status.MPI_TAG!=GLOBALTERMINATION);
if (credit>0)
{
  printf("%d(%d): termination with outstanding credits!\n",rank,__LINE__);
```

```
  MPI_Abort(MPI_COMM_WORLD,__LINE__);
}
if (cr!=iterNumber)
{
  printf("%d(%d): rounds out of synch\n",rank,__LINE__);
  MPI_Abort(MPI_COMM_WORLD,__LINE__);
}
MPI_Barrier(MPI_COMM_WORLD);
//printf("%d: round %d completed\n",rank,iterNumber);
}

main(int argc, char** argv)
{
int seed,iterations,descendants,i;
float prob;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0)
{
  printf("Enter seed: ");
  fflush(stdout);
  scanf("%d",&seed);
  printf("Enter number of iterations: ");
  fflush(stdout);
  scanf("%d",&iterations);
  printf("Enter probability: ");
  fflush(stdout);
  scanf("%f",&prob);
  printf("Enter # of root descendants: ");
  fflush(stdout);
  scanf("%d",&descendants);
}
MPI_Bcast(&seed,1,MPI_INT,0,MPI_COMM_WORLD);
srandom(seed+rank);
MPI_Bcast(&iterations,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&prob,1,MPI_FLOAT,0,MPI_COMM_WORLD);
for (i=0;i<iterations;i++)
  if (rank==0)
    rootProcess(i,descendants);
  else
    slaveProcess(i,prob);
MPI_Finalize();
}
```

Dijkstra-Scholten Tree Technique

Like Mattern's technique, the Dijkstra-Scholten technique is based on a computation ''diffusing'' from the environment (process 0 for this program). When a process is awoken, it will record the sender of the message as the *parent* of the destination node in a changing tree structure. Each message has a complementary *signal* that will eventually be sent in the reverse direction. If the original message awakes a node, the return signal will not be sent until the destination *and all of its children* (as counted by the variable defOut) have gone idle. If a message is sent to a node that is not idle (i.e. the node currently has a parent), then the return signal may be sent immediately, since termination of the subtree rooted by the destination will be detected when the signal is eventually sent to the parent.

```
// MPI for Dijkstra & Scholten's termination detection
// for diffusing computations.
// BPW, 5/2001

#include <stdio.h>
#include "mpi.h"

MPI_Status status;
int rank,numProcesses;

#define MESSAGE 1
#define SIGNAL 2
#define GLOBALTERMINATION 3

int generateRandom(minRange,maxRange)
```

```
int minRange,maxRange;
{
/* returns integer in range minRange <= x <= maxRange*/
return minRange+abs(random()) % (maxRange-minRange+1);
}

void rootProcess(int iterNumber,int descendants)
{
int i,k;

printf("0: round %d started\n",iterNumber);
for (i=1;i<=descendants;i++)
  MPI_Send(&iterNumber,1,MPI_INT,i,MESSAGE,MPI_COMM_WORLD);
for (i=1;i<=descendants;i++)
{
  MPI_Recv(&k,1,MPI_INT,MPI_ANY_SOURCE,SIGNAL,
    MPI_COMM_WORLD,&status);
  if (k!=iterNumber)
  {
    printf("0(%d): rounds out of synch\n",__LINE__);
    MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
}
for (i=1;i<numProcesses;i++)
  MPI_Send(&iterNumber,1,MPI_INT,i,GLOBALTERMINATION,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
printf("0: round %d completed\n");
}

void slaveProcess(int iterNumber,float prob)
{
int parent=(-1);
int defIn=0,defOut=0;
int i,k;

//printf("%d: round %d started\n",rank,iterNumber);
do
{
  MPI_Recv(&k,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
    MPI_COMM_WORLD,&status);
  if (k!=iterNumber)
  {
    printf("%d(%d): rounds out of synch\n",rank,__LINE__);
    MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
  switch (status.MPI_TAG)
  {
    case (MESSAGE):
      defIn++;
      if (parent==(-1))
      {
        parent=status.MPI_SOURCE;
        if (defIn!=1)
        {
          printf("%d(%d): parent error\n",rank,__LINE__);
          MPI_Abort(MPI_COMM_WORLD,__LINE__);
        }
      }
      else
      {
        defIn--;
        if (defIn<1)
        {
          printf("%d(%d): defIn incorrectly < 1\n",rank,__LINE__);
          MPI_Abort(MPI_COMM_WORLD,__LINE__);
        }
        MPI_Send(&iterNumber,1,MPI_INT,status.MPI_SOURCE,
          SIGNAL,MPI_COMM_WORLD);
      }
      // Randomly diffuse (or replace with real work that may diffuse)
      while (1)
      {
        k=generateRandom(0,100000);
        if (k>prob*100000)
```

```
          break;
        k=generateRandom(1,numProcesses-1);
        defOut++;
        MPI_Send(&iterNumber,1,MPI_INT,k,MESSAGE,MPI_COMM_WORLD);
      }
      if (defOut==0)
      {
        defIn--;
        if (defIn!=0)
        {
          printf("%d(%d): defIn should be 0!\n",rank,__LINE__);
          MPI_Abort(MPI_COMM_WORLD,__LINE__);
        }
        MPI_Send(&iterNumber,1,MPI_INT,parent,SIGNAL,MPI_COMM_WORLD);
        parent=(-1);
      }
      break;
    case (SIGNAL):
      defOut--;
      if (defOut==0)
      {
        defIn--;
        if (defIn!=0)
        {
          printf("%d(%d): defIn should be 0!\n",rank,__LINE__);
          MPI_Abort(MPI_COMM_WORLD,__LINE__);
        }
        MPI_Send(&iterNumber,1,MPI_INT,parent,SIGNAL,MPI_COMM_WORLD);
        parent=(-1);
      }
      break;
    case (GLOBALTERMINATION):
      break;
    default:
      printf("%d(%d): bad slaveProcess request\n",rank,__LINE__);
      MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
} while (status.MPI_TAG!=GLOBALTERMINATION);
if (defIn!=0 || defOut!=0)
{
  printf("%d(%d): defIn/defOut invariant problem\n",rank,__LINE__);
  MPI_Abort(MPI_COMM_WORLD,__LINE__);
}
MPI_Barrier(MPI_COMM_WORLD);
//printf("%d: round %d completed\n",rank,iterNumber);
}

main(int argc, char** argv)
{
int seed,iterations,descendants,i;
float prob;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0)
{
  printf("Enter seed: ");
  fflush(stdout);
  scanf("%d",&seed);
  printf("Enter number of iterations: ");
  fflush(stdout);
  scanf("%d",&iterations);
  printf("Enter probability: ");
  fflush(stdout);
  scanf("%f",&prob);
  printf("Enter # of root descendants: ");
  fflush(stdout);
  scanf("%d",&descendants);
}
MPI_Bcast(&seed,1,MPI_INT,0,MPI_COMM_WORLD);
srandom(seed+rank);
MPI_Bcast(&iterations,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&prob,1,MPI_FLOAT,0,MPI_COMM_WORLD);
```

```
for (i=0;i<iterations;i++)
  if (rank==0)
    rootProcess(i,descendants);
  else
    slaveProcess(i,prob);
MPI_Finalize();
}
```

Problems:

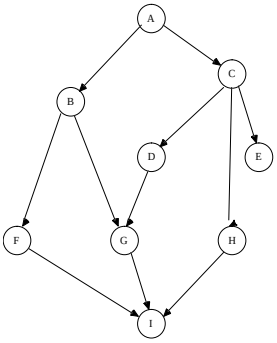
1. Two threads perform the following processing:

Thread A:
while true do
begin
<miscellaneous>
send value of integer variable i to thread B
<miscellaneous>
receive value of integer variable i from thread B
<miscellaneous>
end

Thread B:
while true do
begin
<miscellaneous>
receive value of integer variable j from thread A
<miscellaneous>
send value of integer variable j to thread A
<miscellaneous>
end

Sketch the pthreads code for implementing these threads. Use two mutexes and a *single* shared integer to implement the sends/receives. Do not use any barriers. A thread performing a send does not wait for the other thread to receive. Of course, a thread will block waiting to receive. Since the threads execute sporadically, you must consider race conditions on the shared integer.

2. The following task graph shows data dependencies among sections of code to be included in a single function that will be coded as a single concurrent function. Each section of code will be modified by being executed across the participating threads. Thus, if there is a path from node x to node y, there must be a barrier in the code to assure that all threads have completed the processing for node x before the processing for node y commences. Show how the code should be ordered to minimize the number of barriers. Be sure to indicate the positions of the barriers.



3. What is wrong with the following implementation of a (reusable) barrier? Give the subtle scenario that breaks it and explain how to fix it. count is initialized to 0 and all locks are initialized before any thread encounters this barrier.

lock countLock;
count++;
unlock countLock;

```
lock (mutexTable[threadId]);
if (threadId == 0)
        while (count != numThreads)
                ;
        count = 0;
        for (i=0; i<numThreads; i++)
                unlock mutexTable[i];
else
        lock (mutexTable[threadId]);
        unlock (mutexTable[threadId]);
```

4. Given n threads numbered 0, 1, . . ., n-1, an *ordered critical section* is a piece of code that must be executed *completely* by thread i before thread j (i < j) enters this piece of code. Suggest an implementation for an ordered critical section.

5. Explain how each of the four forms of message passing in SR may be simulated in MPI.

Problem Solutions

1. Two threads . . .

Sketch the pthreads code for implementing these threads. Use two mutexes and a *single* shared integer to implement the sends/receives. Do not use any barriers. A thread performing a send does not wait for the other thread to receive. Of course, a thread will block waiting to receive. Since the threads execute sporadically, you must consider race conditions on the shared integer.

Initialization before spawning threads:

```
pthread_mutex_lock(&Amutex);
pthread_mutex_lock(&Bmutex);
```

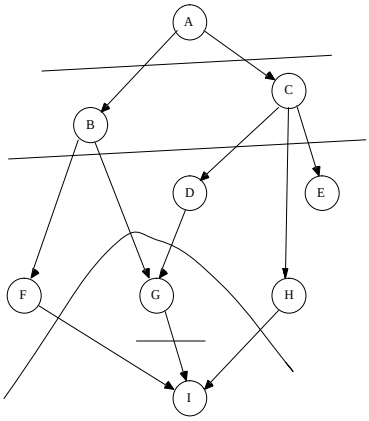
Thread A:

```
while true do
begin
    <miscellaneous>
    /* send value of integer variable i to thread B */
    buffer = i;
    pthread_mutex_unlock(&Bmutex);
    <miscellaneous>
    /* receive value of integer variable i from thread B */
    pthread_mutex_lock(&Amutex);
    i = buffer;
    <miscellaneous>
end
```

Thread B:

```
while true do
begin
    <miscellaneous>
    /* receive value of integer variable j from thread A */
    pthread_mutex_lock(&Bmutex);
    j = buffer;
    <miscellaneous>
    /* send value of integer variable j to thread A */
    buffer = j;
    pthread_mutex_unlock(&Amutex);
    <miscellaneous>
    end
```

2. The following task graph . . .



```
A;
<barrier>
B;
C;
<barrier>
D;
E;
F;
H;
<barrier>
G;
<barrier>
I;
```

3. What is wrong . . .

Scenario that breaks:

Suppose that thread 0 runs ahead of the others.
Some other thread increments count to reach count ==numThreads, but is delayed in setting its lock.
Thread 0 does unlock for some thread *before* that thread sets its lock the first time.

Patch - don't increment count until thread has set its lock

```
lock (mutexTable[threadId]);            /* Sets lock to cause delay for others to catch up */
lock countLock;
count++;
unlock countLock;
if (threadId == 0)
        while (count != numThreads)
                ;
        count = 0;
        for (i=0; i<numThreads; i++)
                unlock mutexTable[i];
else
        lock (mutexTable[threadId]);    /* Thread 0 will unlock to let this continue */
        unlock (mutexTable[threadId]);  /* Ready for next round */
```

4. Given n threads numbered . . .

Use n locks. Initially locks 1, 2, . . ., n-1 are set and lock 0 is cleared. Process i must acquire lock i before processing the critical section and then clears lock (i+1) mod n upon exiting. Indefinite reuse is allowed.

5. Explain how each of the four forms . . .

a. Remote procedure call - receiving MPI process is ''stateless'': after processing RPC, receiving process is ready for another RPC.

   1. Caller sends message(s) with arguments (MPI_Send). Becomes messy if arguments are interesting data structures.
   2. Caller blocks on MPI_Recv for returned values.
   3. Server receives message with arguments.
   4. Server processes.
   5. Server sends back returned values to unblock caller.
   6. Server is ready for another RPC.

b. Dynamic process creation - NOT SUPPORTED by MPI.

c. Asynchronous message sending - this is the usual MPI_Send/MPI_Recv.

d. Rendezvous - similar to a., but receiving process may be more flexible, i.e. does more than service RPCs.