```
#include<stdio.h>        #include<pthread.h>
#include<semaphore.h>            sem_t  mutex, writeblock;
int data = 0,rcount = 0;
int main()
{   int i,b;
  pthread_t  rtid[5],wtid[5];
  sem_init(&mutex,0,1);     sem_init(&writeblock,0,1);
  for(i=0;i<=2;i++)
  {    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++)
  {pthread_join(wtid[i],NULL);
    pthread_join(rtid[i],NULL);
  }   return 0; }
```

```
void *reader(void *arg)
{   int f;
  f = ((int)arg);
  sem_wait(&mutex);
rcount = rcount + 1;
  if(rcount==1)
 sem_wait(&writeblock);
  sem_post(&mutex);
  printf("Data read by the reader%d is %d\n",f,data);
  sleep(1);
  sem_wait(&mutex);
  rcount = rcount - 1;
  if(rcount==0)
   sem_post(&writeblock);
  sem_post(&mutex);
}
```

```
void *writer(void *arg)
{
  int f;
  f = ((int) arg);
  sem_wait(&writeblock);
  data++;
  printf("Data writen by the writer%d is %d\n",f,data);
  sleep(1);
  sem_post(&writeblock);
}
```

```
typedef int buffer_item;        #define BUFFER_SIZE 5
#include <stdlib.h>    #include <stdio.h>
#include <pthread.h>           #include <semaphore.h>
#include "buffer.h"     #define RAND_DIVISOR 100000000
#define TRUE 1
pthread_mutex_t mutex;      sem_t full, empty;
buffer_item buffer[BUFFER_SIZE];
int counter;
pthread_t tid;       //Thread ID
pthread_attr_t attr; //Set of thread attributes
void *producer(void *param); /* the producer thread */
void *consumer(void *param); /* the consumer thread */
```

```
void initializeData() {
   /* Create the mutex lock */
   pthread_mutex_init(&mutex, NULL);
    /* Create the full semaphore and initialize to 0 */
   sem_init(&full, 0, 0);
    /* Create the empty semaphore and initialize to BUFFER_SIZE
    */
   sem_init(&empty, 0, BUFFER_SIZE);
    /* Get the default attributes */
   pthread_attr_init(&attr);

/* init buffer */
   counter = 0;
}
```

```
void *producer(void *param) {
   buffer_item item;
    while(TRUE) {
      /* sleep for a random period of time */
      int rNum = rand() / RAND_DIVISOR;
      sleep(rNum);
    item = rand();
sem_wait(&empty);
pthread_mutex_lock(&mutex);
       if(insert_item(item)) {
  fprintf(stderr, " Producer report error condition\n");
     }
  else {         printf("producer produced %d\n", item);}
pthread_mutex_unlock(&mutex);
sem_post(&full);
   }
}
```

```
void *consumer(void *param) {
   buffer_item item;
    while(TRUE) {
    int rNum = rand() / RAND_DIVISOR;
      sleep(rNum);
    sem_wait(&full);
   pthread_mutex_lock(&mutex);
      if(remove_item(&item)) {
         fprintf(stderr, "Consumer report error condition\n");
      }
      else {printf("c onsumer consumed %d\n", item);      }
      /* release the mutex lock */
      pthread_mutex_unlock(&mutex);
      /* signal empty */
      sem_post(&empty);
   } }
```

```
int insert_item(buffer_item item) {
if(counter < BUFFER_SIZE) {
      buffer[counter] = item;
      counter++;
      return 0;
   }    else {       return -1;   }}

int remove_item(buffer_item *item) {
if(counter > 0) {
      *item = buffer[(counter-1)];
      counter--;
      return 0;
   }
   else {       return -1;
   }
}
```

```
int main(int argc, char *argv[]) {
   /* Loop counter */
   int i;
   /* Verify the correct number of
arguments were passed in */
   if(argc != 4) {
      fprintf(stderr, "USAGE:./main.out
<INT><INT><INT>\n");
   }
   int mainSleepTime = atoi(argv[1]); /*
Time in seconds for main to sleep */
   int numProd = atoi(argv[2]); /* Number
of producer threads */
   int numCons = atoi(argv[3]); /* Number
of consumer threads */
   /* Initialize the app */
```

```
    initializeData();
   /* Create the producer threads */
   for(i = 0; i < numProd; i++) {
      /* Create the thread */
      pthread_create(&tid,&attr,producer,NULL);
    }
   /* Create the consumer threads */
   for(i = 0; i < numCons; i++) {
      /* Create the thread */
      pthread_create(&tid,&attr,consumer,NULL);
   }
   /* Sleep for the specified amount of time in
milliseconds */
   sleep(mainSleepTime);
   /* Exit the program */
   printf("Exit the program\n");
   exit(0);
}
```

DEADLOCKS

- system can be modeled as a collection of limited resources, which can be of different categories, and is to be allocated to a number of processes, **each having different needs**.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- all the resources within a category are equivalent, a request of this category can be equally satisfied by any one of the resources in that category. Otherwise they are sub categorised.
- For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.

DEADLOCKS

- Normally a process must request a resource before using it, and release it when it is done, in the following sequence:
- **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. open( ), malloc( ), new( ), and request( ).
- **Use** - process uses the resource, e.g. prints to the printer or reads from the file.
- **Release** - The process relinquishes the resource. so that it becomes available for other processes. close( ), free( ), delete( ), and release( ).

DEADLOCKS

- kernel-managed resources, kernel tracks resources - free and allocated,
- to which process they are allocated, and a queue of processes waiting for this resource to become available.
- Application-managed resources - using mutexes or wait( ) and signal( ) calls, (i.e. binary /counting semaphores.)
- A set of processes is said to be deadlocked : **every process in the set is waiting for a resource** that **is currently allocated to another process in the set** (and which can only be released when that other waiting process makes progress.)

DEADLOCKS code

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—thread_one and thread_two—are created, and both these threads have access to both mutex locks. thread_one and thread_two run in the functions do_work_one() and do_work_two(), respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Necessary Conditions for Deadlock

- 4 conditions that are necessary for deadlock 2 occur:
  - **Mutual Exclusion** - At least one resource must be held in a **non-sharable mode**;
  - If any other process requests this resource, then that process must wait for the resource to be released.
  - **Hold and Wait** - A process must be **simultaneously holding at least one resource and waiting for at least one resource** that is currently being held by some other process.

- **No preemption** - Once a process is holding a resource (i.e. once its request has been granted),
- then that resource **cannot be taken away** from that process until the process voluntarily releases it.
- **Circular Wait or Cyclic Wait**
- - A set of processes { P0, P1, P2, . . ., PN } must exist such that:
- every **P[i] is waiting for P[(i + 1) % (N + 1)].** (
- implies the hold-and-wait condition,

Resource-Allocation Graph

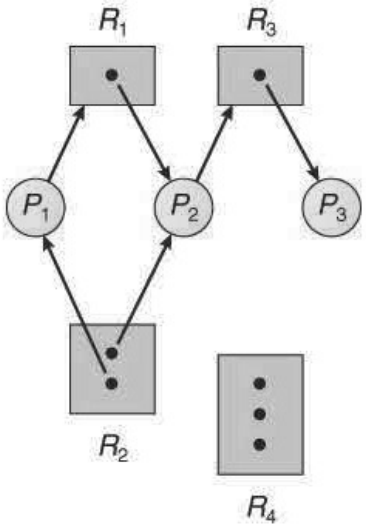
- Deadlocks are best understood using **Resource-Allocation Graphs,** which satisfy the following properties
  - A set of **resource categories**, { R1, R2, R3, . . ., RN }, which appear as **square nodes** on the graph.
  - **Dots inside the resource nodes** indicate specific **instances of the resource**.
  - (E.g. two dots might represent two laser printers.)
  - A set of processes, { P1, P2, P3, . . ., PN }

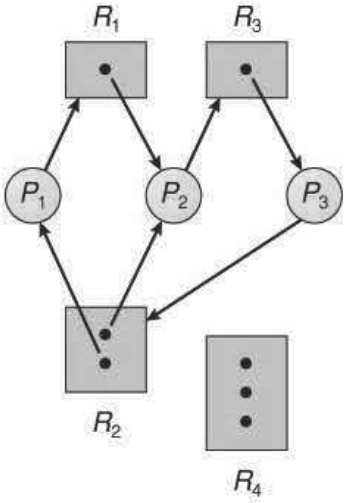
- **Request Edges -**
- A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
- **Assignment Edges -**
- A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.

An Example RAG

•If a resource-allocation graph contains no cycles, then the system is not deadlocked

•If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, **then a deadlock exists**



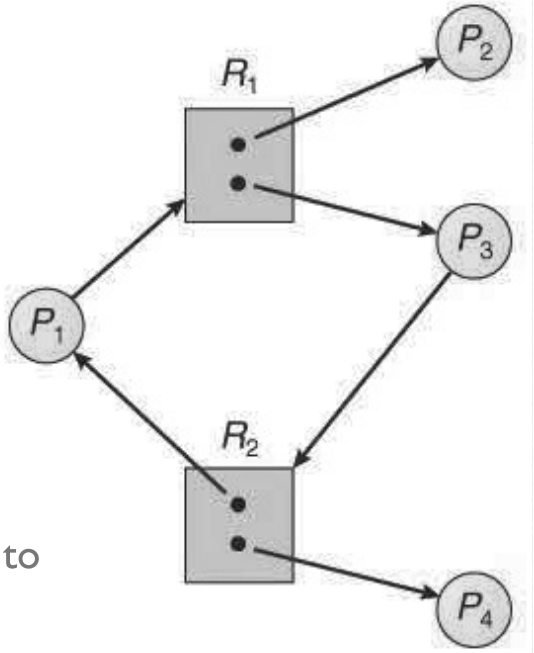


An Example RAG

•If a resource category contains more than one instance,

•then the presence of a cycle in the resource-allocation **graph indicates the *possibility* of a deadlock, but does not guarantee one.**

(next graph is a case for this!)



•3 approaches to Deadlocks

•**Deadlock Detection**

•**Deadlock Prevention**

•**Deadlock Avoidance**

• Infact the RAG model is one approach to Deadlock Detection

•Methods for Handling Deadlocks

1. Deadlock prevention or avoidance - system is not allowed to get into a deadlocked state.
2. Deadlock detection and recovery - system gets into a deadlocked state and then recovery solutions are implemented.
3. Ignore the problem all together – Deadlocks are handled in a crude fashion such as System Reset! (most real time scenarios this proves to be the best and pragmatic solution! Avoiding the time spent in detecting the point of deadlock or worrying about futuristic combination of requests as with an avoidance setup!
4. This is the approach that both Windows and UNIX take.

•Deadlock Prevention

1. Prevention can be achieved by negating the AND of the necessary conditions discussed earlier.
2. **NOT (A && B && C && D)** implies violation of any one of the four conditions would prevent deadlock from occuring!
3. **NO MUTEX** implies all resources are shared by all processes! In Reality some resources such as printers, tape drives, etc need to be accessed in a NON Shared fashion.
4. **NO HOLD and WAIT**: Start execution only when a process gets all the resources it needs (not efficient!) implies all request calls preceded any release call.
5. Alternatively a new process is allowed to request a new resource only if releases all it has . Both 4 & 5 can result in starvation

•Deadlock Prevention

PREMPTION of resources is allowed!

Wait and Die and Wound – Wait Schemes of Transaction Processing Systems

i) a process that needs additional resources is prempted of other resources it holds (waits and dies..) and later reissued again to gain all resources all over again

ii) a process that needs resources held by other process (say P1); resources are prempted from P1 and assigned to this process. Wounds other processes.

iii) Above solutions are unrealistic with MUTEX and Semaphores (to remember state changes..) possible with CPU registers, memory etc.

•Deadlock Prevention

NO CIRCULAR WAIT

- **One way to realize this is thru a total ordering of resources and requests**
- number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- Essence ordering is a one to one function F: R -> N
- F(Tape drive) = 1 ; F(disk drive) = 5 ; F (Printer) = 12
- A Process that holds disk drive cannot request for tape drive or should do so after releasing Tape drive
- Request allowed for Rj only if F(Rj) > F (Ri)
- Release any resource Ri such that F(Ri) >=F(Rj)

- How to prove that above ordering ensures no cyclic wait!
- Proof by contradiction – assume Circular wait holds;
- $P_o, P_1, \ldots P_n$ be the process in the cyclic wait such that P_i waits for R_i held by P_{i+1};
- P_{i+1} is holding R_i which requesting for R_{i+1};
- Implies $F(R_i) < F(R_{i+1})$ – this means
- $F(R_0) < F(R_1) < F(R_2) \ldots < F(R_n) < F(R_0)$
- Which implies $F(R_0) < F(R_0)$ which is not possible. Hence the assumption was wrong.