OSD 19CS2106S

Session – 31

Implementation of spin Locks

Sequence-number-increment example:

When multiple process shares the same file, we need to make sure that each process sees a consistent view of its data. If each process uses files that other process don't read or modify, no consistency problems will exist. Similarly, if a file is read-only, there is no consistency problem with more than one process reading its value at the same time. However, when one process can modify a file that other process can read or modify, we need to synchronize. When one process modifies a file, other process can potentially see inconsistencies when reading the file

Consider the following scenario, which comes from the Unix print spoolers (the BSD **`lpr`** command and the System V **`lp`** command). The process that adds a job to the print queue (to be printed at a later time by another process) must assign a unique sequence number to each print job. The process ID, which is unique while the process is running, cannot be used as the sequence number, because a print job can exist long enough for a given process ID to be reused. A given process can also add multiple print jobs to a queue, and each job needs a unique number. The technique used by the print spoolers is to
have a file for each printer that contains the next sequence number to be used. The file is just a single line containing the sequence number in ASCII. Each process that needs to assign a sequence number goes through three steps:

1. it reads the sequence number file,
2. it uses the number, and it increments the number
3. and writes it back.

The problem is that in the time a single process takes to execute these three steps, another process can perform the same three steps. Chaos can result, as we will see in some examples that follow.

What is needed is for a process to be able to set a lock to say that no other process can access the file until the first process is done. Program:1 does these three steps. The functions **`my-lock`** and **`my-unlock`** are called to lock the file at the beginning and unlock the file when the process is done with the sequence number. We print the name by which the program is being run **`(argv`** `[0 ]`) each time around the loop when we print the sequence number.

```
void
my_lock(int fd)
{
        return;
}

void
my_unlock(int fd)
{
        return;
}
```

If the sequence number in the file is initialized to one, and a single copy of the program is run, we get the following output:

```
[vishnu@team-osd ~]$ cc seqnumnolock.c
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out
./a.out: pid = 5448, seq# = 1
./a.out: pid = 5448, seq# = 2
./a.out: pid = 5448, seq# = 3
./a.out: pid = 5448, seq# = 4
./a.out: pid = 5448, seq# = 5
./a.out: pid = 5448, seq# = 6
./a.out: pid = 5448, seq# = 7
./a.out: pid = 5448, seq# = 8
./a.out: pid = 5448, seq# = 9
./a.out: pid = 5448, seq# = 10
./a.out: pid = 5448, seq# = 11
./a.out: pid = 5448, seq# = 12
./a.out: pid = 5448, seq# = 13
./a.out: pid = 5448, seq# = 14
./a.out: pid = 5448, seq# = 15
./a.out: pid = 5448, seq# = 16
./a.out: pid = 5448, seq# = 17
```

./a.out: pid = 5448, seq# = 18
./a.out: pid = 5448, seq# = 19
./a.out: pid = 5448, seq# = 20

When the sequence number is again initialized to one, and the program is run twice in the background, we have the following output:

```
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out & ./a.out &
[1] 7891
[2] 7892
[vishnu@team-osd ~]$ ./a.out: pid = 7892, seq# = 1
./a.out: pid = 7892, seq# = 2
./a.out: pid = 7892, seq# = 3
./a.out: pid = 7892, seq# = 4
./a.out: pid = 7892, seq# = 5
./a.out: pid = 7892, seq# = 6
./a.out: pid = 7892, seq# = 7
./a.out: pid = 7892, seq# = 8
./a.out: pid = 7892, seq# = 9
./a.out: pid = 7892, seq# = 10
./a.out: pid = 7892, seq# = 11
./a.out: pid = 7892, seq# = 12
./a.out: pid = 7892, seq# = 13
./a.out: pid = 7892, seq# = 14
./a.out: pid = 7891, seq# = 8
./a.out: pid = 7892, seq# = 15
./a.out: pid = 7892, seq# = 16
./a.out: pid = 7892, seq# = 17
./a.out: pid = 7891, seq# = 17
./a.out: pid = 7892, seq# = 18
./a.out: pid = 7891, seq# = 19
./a.out: pid = 7892, seq# = 19
./a.out: pid = 7892, seq# = 20
./a.out: pid = 7891, seq# = 20
./a.out: pid = 7891, seq# = 21
./a.out: pid = 7891, seq# = 22
./a.out: pid = 7891, seq# = 23
./a.out: pid = 7891, seq# = 24
./a.out: pid = 7891, seq# = 25
./a.out: pid = 7891, seq# = 26
./a.out: pid = 7891, seq# = 27
./a.out: pid = 7891, seq# = 28
./a.out: pid = 7891, seq# = 29
./a.out: pid = 7891, seq# = 30
./a.out: pid = 7891, seq# = 31
./a.out: pid = 7891, seq# = 32
./a.out: pid = 7891, seq# = 33
./a.out: pid = 7891, seq# = 34
./a.out: pid = 7891, seq# = 35
./a.out: pid = 7891, seq# = 36

[1]-  Done                    ./a.out
[2]+  Done                    ./a.out
```

This is not what we want. Each process reads, increments, and writes the sequence number file 20 times (there are exactly 40 lines of output), so the ending value of the sequence number should be 40.

What we need is some way to allow a process to prevent other processes from accessing the sequence number file while the three steps are being performed. That is, we need these three steps to be performed as an *atomic operation* with regard to other processes. Another way to look at this problem is that the lines of code between the calls to **`my-lock`** and **`my-unlock`** in program form a *critical region*

When we run two instances of the program in the background as just shown, the output is *nondeterministic.* There is no guarantee that each time we run the two programs we get the same output. This is OK if the three steps listed earlier are handled atomically with regard to other processes, generating an ending value of 40. But this is not OK if the three steps are not handled atomically, often generating an ending value less than 40, which is an error. For example, we do not care whether the first process increments the sequence number from 1 to 20, followed by the second process incrementing

it from 21 to 40, or whether each process runs just long enough to increment the sequence number by two (the first process would print 1 and 2, then the next process would print 3 and 4, and so on).
Being nondeterministic does not make it incorrect. Whether the three steps are performed atomically is what makes the program correct or incorrect. Being nondeterministic, however, usually makes debugging these types of programs harder.

Lock Files

Posix guarantees that if the **open** function is called with the **O-CREAT** (create the file if it does not already exist) and **O-EXCL** flags (exclusive open), the function returns an error if the file already exists. Furthermore, the check for the existence of the file and the creation of the file (if it does not already exist) must be ***atomic*** with regard to other processes. We can therefore use the file created with this technique as a lock. We are guaranteed that only one process at a time can create the file (i.e., obtain the lock), and to release the lock, we just **unlink** the file. Figure 9.12 shows a version of our locking functions using this technique. If the
open succeeds, we have the lock, and the **my-lock** function returns. We **close** the file because we do not need its descriptor: the lock is the existence of the file, regardless of whether the file is open or not. If **open** returns an error of **EEXIST,** then the file exists and we try the **open** again

```
/* Lock functions using open()with O_CREAT and O_EXCL flags. */
#include        <errno.h>
#include        <fcntl.h>               /* for nonblocking */
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <unistd.h>
#define MAXLINE 4096                    /* max text line length */
#define SEQFILE "seqno"                 /* filename */
#define LOCKFILE "seqno.lock"
void    my_lock(int), my_unlock(int);
int
main(int argc, char **argv)
{
        int     fd;
        long    i, seqno;
        pid_t   pid;
        ssize_t n;
        char    line[MAXLINE + 1];
        pid = getpid();
        fd = open(SEQFILE, O_RDWR, 0666);
        for (i = 0; i < 20; i++) {
                my_lock(fd);                    /* lock the file */
                lseek(fd, 0L, SEEK_SET);        /* rewind before read */
                n = read(fd, line, MAXLINE);
                line[n] = '\0';                 /* null terminate for sscanf */
                n = sscanf(line, "%ld\n", &seqno);
                printf("%s: pid = %ld, seq# = %ld\n", argv[0], (long) pid, seqno);
                seqno++;                        /* increment sequence number */
                snprintf(line, sizeof(line), "%ld\n", seqno);
                lseek(fd, 0L, SEEK_SET);        /* rewind before write */
                write(fd, line, strlen(line));
                my_unlock(fd);                  /* unlock the file */
        }
        exit(0);
}
void
my_lock(int fd)
{
        int     tempfd;
        while ( (tempfd = open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, 0644)) < 0) {
                if (errno != EEXIST)
                        printf("open error for lock file");
                /* someone else has the lock, loop around and try again */
        }
        close(tempfd);                  /* opened the file, we have the lock */
}
void
my_unlock(int fd)
{
        unlink(LOCKFILE);               /* release lock by removing file */
}
[vishnu@team-osd ~]$ vi seqno
```

```
[vishnu@team-osd ~]$ cc seqnum.c
[vishnu@team-osd ~]$ vi seqno
[vishnu@team-osd ~]$ ./a.out
./a.out: pid = 5140, seq# = 1
./a.out: pid = 5140, seq# = 2
./a.out: pid = 5140, seq# = 3
./a.out: pid = 5140, seq# = 4
./a.out: pid = 5140, seq# = 5
./a.out: pid = 5140, seq# = 6
./a.out: pid = 5140, seq# = 7
./a.out: pid = 5140, seq# = 8
./a.out: pid = 5140, seq# = 9
./a.out: pid = 5140, seq# = 10
./a.out: pid = 5140, seq# = 11
./a.out: pid = 5140, seq# = 12
./a.out: pid = 5140, seq# = 13
./a.out: pid = 5140, seq# = 14
./a.out: pid = 5140, seq# = 15
./a.out: pid = 5140, seq# = 16
./a.out: pid = 5140, seq# = 17
./a.out: pid = 5140, seq# = 18
./a.out: pid = 5140, seq# = 19
./a.out: pid = 5140, seq# = 20
[vishnu@team-osd ~]$ ./a.out & ./a.out &
[1] 5263
[2] 5264
[vishnu@team-osd ~]$ ./a.out: pid = 5263, seq# = 1
./a.out: pid = 5264, seq# = 2
./a.out: pid = 5263, seq# = 3
./a.out: pid = 5264, seq# = 4
./a.out: pid = 5263, seq# = 5
./a.out: pid = 5264, seq# = 6
./a.out: pid = 5263, seq# = 7
./a.out: pid = 5264, seq# = 8
./a.out: pid = 5263, seq# = 9
./a.out: pid = 5264, seq# = 10
./a.out: pid = 5263, seq# = 11
./a.out: pid = 5264, seq# = 12
./a.out: pid = 5263, seq# = 13
./a.out: pid = 5264, seq# = 14
./a.out: pid = 5263, seq# = 15
./a.out: pid = 5264, seq# = 16
./a.out: pid = 5263, seq# = 17
./a.out: pid = 5264, seq# = 18
./a.out: pid = 5263, seq# = 19
./a.out: pid = 5264, seq# = 20
./a.out: pid = 5263, seq# = 21
./a.out: pid = 5263, seq# = 22
./a.out: pid = 5264, seq# = 23
./a.out: pid = 5263, seq# = 24
./a.out: pid = 5264, seq# = 25
./a.out: pid = 5263, seq# = 26
./a.out: pid = 5264, seq# = 27
./a.out: pid = 5263, seq# = 28
./a.out: pid = 5264, seq# = 29
./a.out: pid = 5263, seq# = 30
./a.out: pid = 5264, seq# = 31
./a.out: pid = 5263, seq# = 32
./a.out: pid = 5264, seq# = 33
./a.out: pid = 5263, seq# = 34
./a.out: pid = 5264, seq# = 35
./a.out: pid = 5263, seq# = 36
./a.out: pid = 5264, seq# = 37
./a.out: pid = 5263, seq# = 38
./a.out: pid = 5264, seq# = 39
./a.out: pid = 5264, seq# = 40
```

[1]- Done ./a.out
[2]+ Done ./a.out

Each file type has its own unique data structure that records all the information in a format only the software that created it can understand. Databases store metadata in fixed tables with rigid schemas and pointers linking multiple tables together. This enables data to be structured and categorized so that it can be indexed, searched and easily manipulated using Structured Query Language (SQL) transactions. A simple SQL query is able to extract all the information about a file and its relational hierarchy to all the other files in your database.

how AUTO_INCREMENT and concurrent insert works using locks in DBMS

Terminal – 1	Terminal – 2

`$mysql -u root -p`
`MariaDB [(none)]> create database testdb;`
`MariaDB [(none)]> create user 'testuser'@localhost identified by 'password';`
`MariaDB [(none)]> grant all on testdb.* to 'testuser' identified by 'password';`
`MariaDB [(none)]> quit;`
`$mysql -u testuser -p`
 	
`MariaDB [(none)]> SHOW DATABASES;`
`MariaDB [(none)]> use testdb;`
 	
`MariaDB [testdb]> CREATE TABLE cities (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, name VARCHAR(100), pop MEDIUMINT, founded DATE, owner VARCHAR(100));`
`MariaDB [testdb]> show tables;`
`MariaDB [testdb]> describe cities;`
 	
`MariaDB [testdb]> DELIMITER //`
 	
`CREATE PROCEDURE dorepeatt(p1 INT)`
`  BEGIN`
`    SET @x = 1;`
`    REPEAT SET @x = @x + 1;`
`INSERT INTO cities (id, name, pop, founded, owner)`
`VALUES`
`(NULL, 'a', 20, "2010-01-12",'testuser'),`
`(NULL, 'b', 56, "2011-2-28",'testuser'),`
`(NULL, 'c', 3, "120314",'testuser'),`
`(NULL, 'd', 34, "13*04*21",'testuser'),`
`(NULL, 'e', 12, "2010-09-19",'testuser'),`
`(NULL, 'f', 09, "2010-12-13",'testuser');`
` UNTIL @x > p1 END REPEAT;`
`  END`
`//`
 	
`MariaDB [testdb]> DELIMITER ;`
	`$mysql -u root -p`
	`MariaDB [(none)]> create user 'vishnu'@localhost identified by 'password';`
	`MariaDB [(none)]> grant all on testdb.* to 'vishnu' identified by 'password';`
	`MariaDB [(none)]> quit;`
	
	`$mysql -u vishnu -p`
	
	`MariaDB [(none)]> SHOW DATABASES;`
	`MariaDB [(none)]> use testdb;`
	`MariaDB [testdb]>show tables;`
	`MariaDB [testdb]>describe cities;`
	
	`MariaDB [testdb]>DELIMITER //`
	
	`CREATE PROCEDURE dorepeatv(p1 INT)`
	`  BEGIN`
	`    SET @x = 1;`
	`    REPEAT SET @x = @x + 1;`
	`INSERT INTO cities (id, name, pop, founded, owner)`
	`VALUES`
	`(NULL, 'A', 120, "2020-09-30",'vishnu'),`
	`(NULL, 'B', 156, "2001-8-22",'vishnu'),`
	`(NULL, 'C', 13, "180924",'vishnu'),`
	`(NULL, 'D', 134, "17*09*13",'vishnu'),`
	`(NULL, 'E', 112, "2018-11-29",'vishnu'),`
	`(NULL, 'F', 109, "2000-02-17",'vishnu');`
	` UNTIL @x > p1 END REPEAT;`
	`  END`
	`//`
	
	`MariaDB [testdb]>DELIMITER ;`

Execute in two terminals in parallel	
`MariaDB [testdb]> CALL dorepeatt(100);`	`MariaDB [testdb]>CALL dorepeatv(100);`

```
MariaDB [testdb]> select * from cities;
+------+------+------+------------+----------+
| id   | name | pop  | founded    | owner    |
+------+------+------+------------+----------+
|    1 | a    |   20 | 2010-01-12 | testuser |
|    2 | b    |   56 | 2011-02-28 | testuser |
|    3 | c    |    3 | 2012-03-14 | testuser |
|    4 | d    |   34 | 2013-04-21 | testuser |
|    5 | e    |   12 | 2010-09-19 | testuser |
.
.
.
.

|  364 | d    |   34 | 2013-04-21 | testuser |
|  365 | e    |   12 | 2010-09-19 | testuser |
|  366 | f    |    9 | 2010-12-13 | testuser |
|  367 | A    |  120 | 2020-09-30 | vishnu   |
|  368 | B    |  156 | 2001-08-22 | vishnu   |
.
.
.


|  399 | c    |    3 | 2012-03-14 | testuser |
|  400 | d    |   34 | 2013-04-21 | testuser |
|  401 | e    |   12 | 2010-09-19 | testuser |
```

```
|  402 | f    |    9 | 2010-12-13 | testuser |
|  403 | A    |  120 | 2020-09-30 | vishnu   |
|  404 | B    |  156 | 2001-08-22 | vishnu   |
|  405 | C    |   13 | 2018-09-24 | vishnu   |
|  406 | D    |  134 | 2017-09-13 | vishnu   |
|  407 | E    |  112 | 2018-11-29 | vishnu   |
|  408 | F    |  109 | 2000-02-17 | vishnu   |
|  409 | a    |   20 | 2010-01-12 | testuser |
|  410 | b    |   56 | 2011-02-28 | testuser |

.
.
.

|  634 | d    |   34 | 2013-04-21 | testuser |
|  635 | e    |   12 | 2010-09-19 | testuser |
|  636 | f    |    9 | 2010-12-13 | testuser |

.
.
.

| 1191 | C    |   13 | 2018-09-24 | vishnu   |
| 1192 | D    |  134 | 2017-09-13 | vishnu   |
| 1193 | E    |  112 | 2018-11-29 | vishnu   |
| 1194 | F    |  109 | 2000-02-17 | vishnu   |
| 1195 | A    |  120 | 2020-09-30 | vishnu   |
| 1196 | B    |  156 | 2001-08-22 | vishnu   |
| 1197 | C    |   13 | 2018-09-24 | vishnu   |
| 1198 | D    |  134 | 2017-09-13 | vishnu   |
| 1199 | E    |  112 | 2018-11-29 | vishnu   |
| 1200 | F    |  109 | 2000-02-17 | vishnu   |
+------+------+------+------------+----------+
1200 rows in set (0.00 sec)
```

19CS2106S

Operating Systems Design

Session: 32

Solution that uses the exchange primitive to build a lock

```
/* Example: Creating a Thread and waiting for Thread Completion */
#include <stdio.h>
 #include <pthread.h>
 #include <stdlib.h>

 typedef struct __myarg_t {
     int a;
     int b;
 } myarg_t;

 typedef struct __myret_t {
     int x;
     int y;
 } myret_t;

 void *mythread(void *arg) {
     myarg_t *m = (myarg_t *) arg;
     printf("%d %d\n", m->a, m->b);
     myret_t *r = malloc(sizeof(myret_t));
     r->x = 1;
     r->y = 2;
     return (void *) r;
 }
int main(int argc, char *argv[]) {
     int rc;
     pthread_t p;
     myret_t *m;

     myarg_t args;
     args.a = 10;
     args.b = 20;
     pthread_create(&p, NULL, mythread, &args);
     pthread_join(p, (void **) &m);  // this thread has been waiting inside of the
pthread_join() routine.
     printf("returned %d %d\n", m->x, m->y);
     return 0;
 }
/*
[vishnu@team-osd ~]$ cc thread.c -lpthread
[vishnu@team-osd ~]$ ./a.out
10 20
returned 1 2
*/

/* Example: Simpler Argument Passing to a Thread */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void *mythread(void *arg) {
     int m = (int) arg;
     printf("%d\n", m);
     return (void *) (arg + 1);
 }

 int main(int argc, char *argv[]) {
     pthread_t p;
     int rc, m;
     pthread_create(&p, NULL, mythread, (void *) 100);
     pthread_join(p, (void **) &m);
     printf("returned %d\n", m);
```

```
     return 0;
 }
/*
[vishnu@team-osd ~]$ cc threadsinglevalue.c -lpthread
threadsinglevalue.c: In function ‘mythread’:
threadsinglevalue.c:7:14: warning: cast from pointer to integer of different size [-
Wpointer-to-int-cast]
      int m = (int) arg;
              ^
[vishnu@team-osd ~]$ ./a.out
100
returned 101
*/

/* Why it gets worse while Shared Data, nondeterministic start-up and uncontrolled
scheduling - A solution that uses the exchange primitive to build a lock. main-thread-
5.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int max;
volatile int balance = 0;

//
// xchg(int *addr, int newval)
// return what is pointed to by addr
// at the same time, store newval into addr
//
static inline uint
xchg(volatile unsigned int *addr, unsigned int newval)
{
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr), "=a" (result) : "1" (newval) :
"cc");
    return result;
}

volatile unsigned int mutex = 0;

void
SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
      ; // spin
}

void
SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}


void *
mythread(void *arg)
{
    char *letter = arg;
    //cpubind();
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      SpinLock(&mutex);
      balance = balance + 1;
      SpinUnlock(&mutex);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    if (argc != 2) {
      fprintf(stderr, "usage: main-first <loopcount>\n");
      exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [balance = %d]\n", balance);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
         balance, max*2);
    return 0;
}
/*
[vishnu@localhost threads]$ cc main-thread-5.c -lpthread
[vishnu@localhost threads]$ ./a.out
usage: main-first <loopcount>
[vishnu@localhost threads]$ ./a.out 4
main: begin [balance = 0]
A: begin
B: begin
B: done
A: done
main: done
 [balance: 8]
 [should: 8]
*/
```

OSD 19CS2106S
Session – 33
Lecture Notes

UNIX provides several different IPC mechanisms.

Interprocess interactions have several distinct purposes:

- **Data transfer —** One process may wish to send data to another process. The amount of data sent may vary from one byte to several megabytes.
- **Sharing data —** Multiple processes may wish to operate on shared data, such that if a process modifies the data, that change will be immediately visible to other processes sharing it.
- **Event notification —** A process may wish to notify another process or set of processes that some event has occurred. For instance, when a process terminates, it may need to inform its parent process. The receiver may be notified asynchronously, in which case its normal processing is interrupted. Alternatively, the receiver may choose to wait for the notification.
- **Resource sharing —** Although the kernel provides default semantics for resource allocation, they are not suitable for all applications. A set of cooperating processes may wish to define their own protocol for accessing specific resources. Such rules are usually implemented by a locking and synchronization scheme, which must be built on top of the basic set of primitives provided by the kernel.
- **Process control —** A process such as a debugger may wish to assume complete control over the execution of another (target) process. The controlling process may wish to intercept all traps and exceptions intended for the target and be notified of any change in the target's state.

IPC structure

The three types of IPC that we call XSI IPC - message queues, semaphores, and shared memory have many similarities. The XSI IPC functions are based closely on the System V IPC functions. message queues, semaphores, and shared memory are defined as XSI extensions in the Single UNIX Specification.

Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier. To send or fetch a message to or from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name. Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
   The `IPC_PRIVATE` key is also used in a parent child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.
2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
3. The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.
4. The three `get` functions (`msgget`, `semget`, and `shmget`) all have two similar arguments: a key and an integer flag. A new IPC structure is created (normally, by a server) if either key is `IPC_PRIVATE` or key is not currently associated with an IPC structure of the particular type and the `IPC_CREAT` bit of flag is specified. To reference an existing queue (normally done by a client), key must equal the key that was specified when the queue was created, and `IPC_CREAT` must not be specified.
5. Note that it's never possible to specify `IPC_PRIVATE` to reference an existing queue, since this special key value always creates a new queue. To reference an existing queue that was created with a key of `IPC_PRIVATE`, we must know the associated identifier and then use that identifier in the other IPC calls (such as `msgsnd` and `msgrcv`), bypassing the `get` function.
6. If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a flag with both the `IPC_CREAT` and `IPC_EXCL` bits set. Doing this causes an error return of `EEXIST` if the IPC structure already exists. (This is similar to an `open` that specifies the `O_CREAT` and `O_EXCL` flags.)

Permission Structure

XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user id */
  gid_t  gid;  /* owner's effective group id */
  uid_t  cuid; /* creator's effective user id */
  gid_t  cgid; /* creator's effective group id */
  mode_t mode; /* access modes */
  .
  .
  .
};
```

Each implementation includes additional members. See `<sys/ipc.h>` on your system for the complete definition.
All the fields are initialized when the IPC structure is created. At a later time, we can modify the `uid`, `gid`, and `mode` fields by calling `msgctl`, `semctl`, or `shmctl`. To change these values, the calling process must be either the creator of the IPC structure or the superuser.

Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are system wide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.
Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in File I/O. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands `ipcs`(1) and `ipcrm`(1)were added.

Interprocess Communication

Processes have to communicate to exchange data and to synchronize operations. Several forms of interprocess communication are: pipes, named pipes and signals. But each one has some limitations.

System V IPC

The UNIX System V IPC package consists of three mechanisms:

1. Messages allow process to send formatted data streams to arbitrary processes.
2. Shared memory allows processes to share parts of their virtual address space.
3. Semaphores allow processes to synchronize execution.

The share common properties:

- Each mechanism contains a table whose entries describe all instances of the mechanism.
- Each entry contains a numeric *key*, which is its user-chosen name.
- Each mechanism contains a "get" system call to create a new entry or to retrieve an existing one, and parameters to the calls include a key and flags.
- The kernel uses the following formula to find the index into the table of data structures from the descriptor: `index = descriptor modulo (number of entries in table);`
- Each entry has a permissions structure that includes the user ID and group ID of the process that created the entry, a user and group ID set by the "control" system call (studied below), and a set of read-write-execute permissions for user, group, and others, similar to the file permission modes.
- Each entry contains other information such as the process ID of the last process to update the entry, and time of last access or update.
- Each mechanism contains a "control" system call to query status of an entry, to set status information, or to remove the entry from the system.

Messages

There are four system calls for messages:

1. *msgget* returns (and possibly creates) a message descriptor.
2. *msgctl* has options to set and return parameters associated with a message descriptor and an option to remove descriptors.
3. *msgsnd* sends a message.
4. *msgrcv* receives a message.

Syntax of *msgget*:

```
msgqid = msgget(key, flag);
```

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

- Pointers to the first and last messages on a linked list.

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

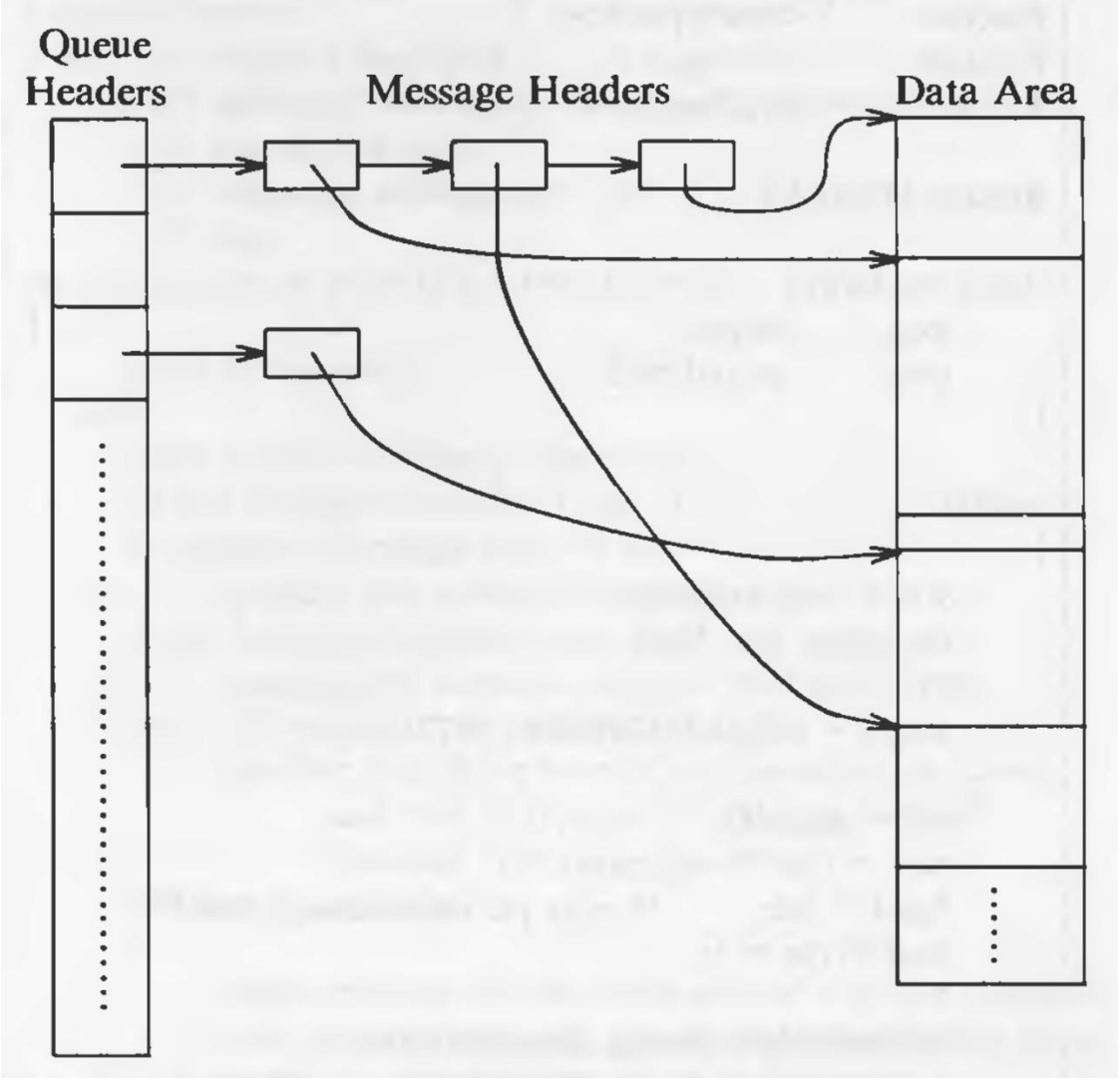
Syntax of *msgsnd*:

```
msgsnd(msgqid, msg, count, flag);
```

`flag` describes the action the kernel should take if it runs out of internal buffer space. The algorithm is given below:

```
/*  Algorithm: msgsnd
 *  Input: message queue descriptor
 *         address of message structure
 *         size of message
 *         flags
 *  Output: number of bytes send
 */
{
        check legality of descriptor, permissions;
        while (not enough space to store message)
        {
                if (flags specify not to wait)
                        return;
                sleep(event: enough space is available);
        }
        get message header;
        read message text from user space to kernel;
        adjust data structures: enqueue message header,
                message header points to data, counts,
                time stamps, process ID;
        wakeup all processes waiting to read message from queue;
}
```

The diagram shows the structure of message queues:



Syntax for *msgrcv*:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

The algorithm is given below:

```
/*  Algorithm: msgrcv
 *  Input: message descriptor
 *         address of data array for incoming message
 *         size of data array
 *         requested message type
 *         flags
 *  Output: number of bytes in returned message
 */

{
        check permissions;
 loop:
        check legality of message descriptor;
        // find message to return to user
        if (requested message type == 0)
                consider first message on queue;
        else if (requested message type > 0)
                consider first message on queue with given type;
        else                    // requested message type < 0
                consider first of the lowest typed messages on queue,
                        such that its type is <= absolute value of requested type;
        if (there is a message)
        {
                adjust message size or return error if user size too small;
                copy message type, text from kernel space to user space;
                unlink message from queue;
                return;
        }
        // no message
        if (flags specify not to sleep)
                return with error;
        sleep(event: message arrives on queue);
        goto loop;
}
```

If processes were waiting to send messages because there was no room on the list, the kernel awakens them after it removes a message from the message queue. If a message is bigger than *maxcount*, the kernel returns an error for the system call leaves the message on the queue. If a process ignores the size constraints (*MSG_NOERROR* bit is set in *flag*), the kernel truncates the message, returns the requested number of bytes, and removes the entire message from the list.

If the *type* is a positive integer, the kernel returns the first message of the given type. If it is a negative, the kernel finds the lowest type of all message on the queue, provided it is less than or equal to the absolute value of the *type*, and returns the first message of that type. For example, if a queue contains three messages whose types are 3, 1, and 2, respectively, and a user requests a message with type -2, the kernel returns the message of type 1.

The syntax of *msgctl*:

```
msgctl(id, cmd, mstatbuf);
```

where `cmd` specifies the type of command, and `mstatbuf` is the address of a user data structure that will contain control parameters or the results of a query.

Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

The Single UNIX Specification includes an alternate IPC message queue implementation in the message-passing option of its real-time extensions. We do not cover the real-time extensions in this text.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
   struct msqid_ds {
     struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
     msgqnum_t        msg_qnum;     /* # of messages on queue */
     msglen_t         msg_qbytes;   /* max # of bytes on queue */
     pid_t            msg_lspid;    /* pid of last msgsnd() */
     pid_t            msg_lrpid;    /* pid of last msgrcv() */
     time_t           msg_stime;    /* last-msgsnd() time */
     time_t           msg_rtime;    /* last-msgrcv() time */
```

```
    time_t           msg_ctime;    /* last-change time */
    .
    .
    .
};
```

This structure defines the current status of the queue.
The cmd argument specifies the command to be performed on the queue specified by msqid.

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by buf.

`IPC_SET` Copy the following fields from the structure pointed to by buf to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

POSIX:XSI Interprocess Communication

The POSIX interprocess communication (IPC) is part of the POSIX:XSI Extension and has its origin in UNIX System V interprocess communication. IPC, which includes message queues, semaphore sets and shared memory, provides mechanisms for sharing information among processes on the same system. These three communication mechanisms have a similar structure, and this chapter emphasizes the common elements of their use. Table given below summarizes the POSIX:XSI interprocess communication functions.

POSIX:XSI interprocess communication functions.

mechanism	POSIX function	meaning
message queues	`msgctl`	control
	`msgget`	create or access
	`msgrcv`	receive message
	`msgsnd`	send message
semaphores	`semctl`	control
	`semget`	create or access
	`semop`	execute operation (wait or post)
shared memory	`shmat`	attach memory to process
	`shmctl`	control
	`shmdt`	detach memory from process
	`shmget`	create and initialize or access

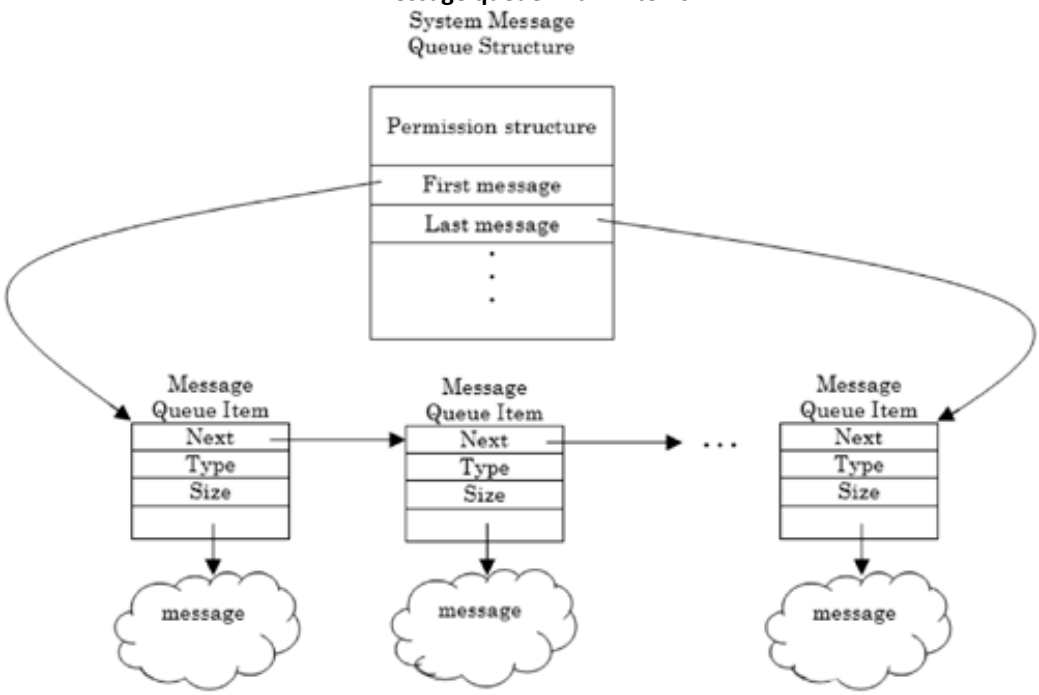
The designers of UNIX found the types of interprocess communications that could be implemented using signals and pipes to be restrictive. To increase the flexibility and range of interprocess communication, supplementary communication facilities were added. These facilities, added with the release of System V in the 1970s, are grouped under the heading IPC (Interprocess Communication). In brief, these are

- Message queues— Information to be communicated is placed in a predefined message structure. The process generating the message specifies its type and places the message in a system-maintained message queue. Processes accessing the message queue can use the message type to selectively read messages of specific types in a first in first out (FIFO) manner. Message queues provide the user with a means of asynchronously multiplexing data from multiple processes.
- Semaphores— Semaphores are system-implemented data structures used to communicate small amounts of data between processes. Most often, semaphores are used for process synchronization.
- Shared memory— Information is communicated by accessing shared process data space. This is the fastest method of interprocess communication. Shared memory allows participating processes to randomly access a shared memory segment. Semaphores are often used to synchronize the access to the shared memory segments.

All three of these facilities can be used by related and unrelated processes, but these processes must be on the same system (machine).
Like a file, an IPC resource must be generated before it can be used. Each IPC resource has a creator, owner, and access permissions. These attributes, established when the IPC is created, can be modified using the proper system calls. At a system level, information about the IPC facilities supported by the system can be obtained with the `ipcs` command.

Summary of the System V IPC Calls.			
Functionality	Message Queue	System Call Semaphore	Shared Memory
Allocate an IPC resource; gain access to an existing IPC resource.	`msgget`	`semget`	`shmget`
Control an IPC resource: obtain/modify status information, remove the resource.	`msgctl`	`semctl`	`shmctl`
IPC operations: send/receive messages,perform semaphore operations, attach/free a shared memory segment.	`msgsnd` `msgrcv`	`semop`	`shmat` `shmdt`

A message queue with N items



Message Queues IPC

Where's my queue?

First of all, you want to connect to a queue, or create it if it doesn't exist. The call to accomplish this is the **`msgget()`** system call: **`int msgget(key_t key, int msgflg);`**

`msgget()` returns the message queue ID on success, or `-1` on failure (and it sets *`errno`*, of course.)

The first arguments, *`key`* is a system-wide unique identifier describing the queue you want to connect to (or create). Every other process that wants to connect to this queue will have to use the same *`key`*.

The other argument, *`msgflg`* tells **`msgget()`** what to do with queue in question. To create a queue, this field must be set equal to `IPC_CREAT` bit-wise OR'd with the permissions for this queue. (The queue permissions are the same as standard file permissions—queues take on the user-id and group-id of the program that created them.)

A sample call is given in the following section.

"Are you the Key Master?"

`key_t ftok(const char *path, int id);`

The **`ftok()`** function uses information about the named file (like inode number, etc.) and the *`id`* to generate a probably-unique *`key`* for **`msgget()`**. Programs that want to use the same queue must generate the same *`key`*, so they must pass the same parameters to **`ftok()`**.

Finally, it's time to make the call:

```
#include <sys/msg.h>
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

In the above example, I set the permissions on the queue to `666` (or `rw-rw-rw-`). And now we have *`msqid`* which will be used to send and receive messages from the queue.

Sending to the queue

Once you've connected to the message queue using **`msgget()`**, you are ready to send and receive messages. First, the sending: Each message is made up of two parts, which are defined in the template structure `struct msgbuf`, as defined in *`sys/msg.h`*:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

The field *mtype* is used later when retrieving messages from the queue, and can be set to any positive number. *mtext* is the data this will be added to the queue.
You can use any structure you want to put messages on the queue, as long as the first element is a long. For instance, we could use this structure to store all kinds of goodies:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
        int booty_value;
    } info;
};
```

msgsnd():

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

msqid is the message queue identifier returned by **msgget()**. The pointer *msgp* is a pointer to the data you want to put on the queue. *msgsz* is the size in bytes of the data to add to the queue (not counting the size of the *mtype* member). Finally, *msgflg* allows you to set some optional flag parameters, which we'll ignore for now by setting it to 0. When to get the size of the data to send, just take the size of the second field:

```
struct cheese_msgbuf {
    long mtype;
    char name[20];
};
/* calculate the size of the data to send: */
struct cheese_msgbuf mbuf;
int size;
size = sizeof mbuf.name;

#include <sys/msg.h>
#include <stddef.h>
key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, { "vishnu", 'S', 80, 10, 12035 } };
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
/* stick him on the queue */
/* struct pirate_info is the sub-structure */
msgsnd(msqid, &pmb, sizeof(struct pirate_info), 0);
```

Receiving from the queue

A call to **msgrcv()** that would do it looks something like this:

```
#include <sys/msg.h>
#include <stddef.h>
key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where vishnu is to be kept */
key = ftok("/home/vishnu/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
/* get him off the queue! */
msgrcv(msqid, &pmb, sizeof(struct pirate_info), 2, 0);
```

There is something new to note in the **msgrcv()** call: the 2! What does it mean? Here's the synopsis of the call:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

The 2 we specified in the call is the requested *msgtyp*. Recall that we set the *mtype* arbitrarily to 2 in the **msgsnd()** section of this document, so that will be the one that is retrieved from the queue.
Actually, the behavior of **msgrcv()** can be modified drastically by choosing a *msgtyp* that is positive, negative, or zero:

msgtyp	**Effect on msgrcv()**
Zero	Retrieve the next message on the queue, regardless of its *mtype*.
Positive	Get the next message with an *mtype* ***equal to*** the specified *msgtyp*.
Negative	Retrieve the first message on the queue whose *mtype* field is less than or equal to the absolute value of the *msgtyp* argument.

So, what will often be the case is that you'll simply want the next message on the queue, no matter what *mtype* it is. As such, you'd set the *msgtyp* parameter to 0.

Destroying a message queue

There comes a time when you have to destroy a message queue. There are two ways:

1. Use the Unix command **ipcs** to get a list of defined message queues, then use the command **ipcrm** to delete the queue.

2. Write a program to do it for you.

Often, the latter choice is the most appropriate, since you might want your program to clean up the queue at some time or another. To do this requires the introduction of another function: **`msgctl()`**.

The synopsis of **`msgctl()`** is:

`int msgctl(int msqid, int cmd, struct msqid_ds *buf);`

Of course, *`msqid`* is the queue identifier obtained from **`msgget()`**. The important argument is *`cmd`* which tells **`msgctl()`** how to behave. It can be a variety of things, but we're only going to talk about `IPC_RMID`, which is used to remove the message queue. The *`buf`* argument can be set to `NULL` for the purposes of `IPC_RMID`.

Say that we have the queue we created above to hold the pirates. You can destroy that queue by issuing the following call:

```
#include <sys/msg.h>
.
msgctl(msqid, IPC_RMID, NULL);
```

And the message queue is no more.

`/*** kirk.c --demonstrate Message Queues IPC - adds messages to the message queue */`

```
#include  <stdio.h>
#include  <stdlib.h>

#include  <errno.h>
#include  <string.h>
#include  <sys/types.h>

#include  <sys/ipc.h>
#include  <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];

};
int main(void)

{
    struct my_msgbuf buf;

    int msqid;

    key_t key;
    if ((key = ftok("kirk.c", 'B')) == -1) {

         perror("ftok");
         exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
         perror("msgget");

         exit(1);
    }
    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */
    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {

      int len = strlen(buf.mtext);
      /* ditch newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
          perror("msgsnd");

   }
```

```
   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");

      exit(1);
   }

  return 0;
}
/*[vishnu@mannava ~]$ cc kirk.c -o kirk

[vishnu@mannava ~]$ ./kirk
Enter lines of text, ^D to quit:

hello how are you
type ipcs -q and check
third line

hello
*/

/*
[vishnu@mannava ~]$ ipcs -q
------ Message Queues --------

key        msqid      owner      perms      used-bytes   messages
0x42004205 0          vishnu     644        43           2
```

/*spock.c - demonstrate Message Queues IPC - retrieves messages from the message queue */

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msgbuf {
    long mtype;

    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;

    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {   /* same key as kirk.c */
         perror("ftok");
         exit(1);

    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue
         */ perror("msgget");
         exit(1);
    }

    printf("spock: ready to receive messages,
    captain.\n"); for(;;) { /* Spock never quits! */
```

```
        if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1)
             { perror("msgrcv");
             exit(1);

        }
        printf("spock: \"%s\"\n", buf.mtext);

    }
    return 0;
}

/*[vishnu@mannava ~]$ ./spock

spock: ready to receive messages, captain. spock: "hello how are you"
spock: "type ipcs -q and check"
spock: "third line"
spock: "hello"
msgrcv: Identifier
removed */

[vishnu@team-osd ~]$ ipcs -l

------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 1940588
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 100
semaphore max value = 32767

[vishnu@team-osd ~]$ ipcs -u

------ Messages Status --------
allocated queues = 9
used headers = 4
used space = 378 bytes

------ Shared Memory Status --------
segments allocated 96
pages allocated 14011
pages resident  3191
pages swapped   6271
Swap performance: 0 attempts     0 successes

------ Semaphore Status --------
used arrays = 0
allocated semaphores = 0

[vishnu@team-osd ~]$ ipcs -q -c

------ Message Queues Creators/Owners --------
msqid      perms      cuid       cgid       uid        gid
12         644        osd-kiran3168 osd-kiran3168 osd-kiran3168 osd-kiran3168
13         644        vishnu     vishnu     vishnu     vishnu
```

OSD 19CS2106S
Session – 34
Shared Memory IPC
Lecture Notes

Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking can also be used.)

The Single UNIX Specification includes an alternate set of interfaces to access shared memory in the shared memory objects option of its real-time extensions. We do not cover the real-time extensions in this text.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Earlier Section */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

- The `ipc_perm` structure is initialized as described earlier. The `mode` member of this structure is set to the corresponding permission bits of flag. These permissions are specified with the values given earlier.
- `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the size requested.

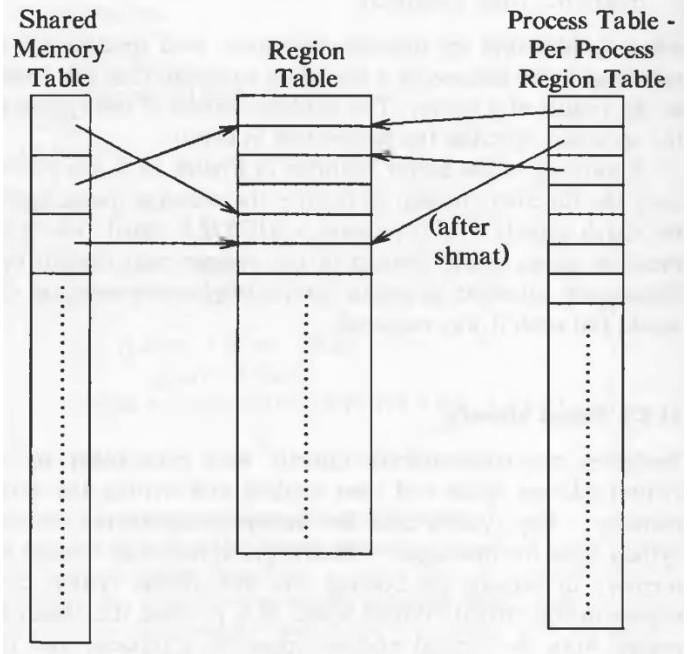
Shared Memory

Sharing the part of virtual memory and reading to and writing from it, is another way for the processes to communicate. The system calls are:

- *shmget* creates a new region of shared memory or returns an existing one.
- *shmat* logically attaches a region to the virtual address space of a process.
- *shmdt* logically detaches a region.
- *shmctl* manipulates the parameters associated with the region.

```
shmid = shmget(key, size, flag);
```

where `size` is the number of bytes in the region. If the region is to be created, *allocreg* is used. It sets a flag in the shared memory table to indicate that the memory is not allocated to the region. The memory is allocated only when the region gets attached. A flag is set in the region table which indicates that the region should not be freed when the last process referencing it, *exit*s. The data structure are shown below:



Syntax for *shmat*:
`virtaddr = shmat(id, addr, flags);`
where `addr` is the virtual address where the user wants to attach the shared memory. And the `flags` specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value *virtaddr* is the virtual address where the kernel attached the region, not necessarily the value requested by the process.
The algorithm is given below:

```
/*  Algorithm: shmat
 *  Input: shared memory descriptor
 *         virtual addresses to attach memory
 *         flags
 *  Output: virtual address where memory was attached
 */
{
        check validity of descriptor, permissions;
        if (user specified virtual address)
        {
                round off virtual address, as specified by flags;
                check legality of virtual address, size of region;
        }
        else                    // user wants kernel to find good address
                kernel picks virtual address: error if none available;
        attach region to process address space (algorithm: attachreg);
        if (region being attached for first time)
                allocate page tables, memory for region (algorithm: growreg);
        return (virtual address where attached);
}
```

If the address where the region is to be attached is given as 0, the kernel chooses a convenient virtual address. If the calling process is the first process to attach that region, it means that page tables and memory are not allocated for that region, therefore, the kernel allocated both using *growreg*.
The syntax for *shmdt*:
`shmdt(addr);`
where `addr` is the virtual address returned by a prior *shmat* call. The kernel searches for the process region attached at the indicated virtual address and detaches it using *detachreg*. Because the region tables have no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.
Syntax of *shmctl*
`shmctl(id, cmd, shmstatbuf);`
which is similar to *msgctl*

Shared Memory IPC

A segment of memory is shared between processes. You just connect to the shared memory segment, and get a pointer to the memory. You can read and write to this pointer and all changes you make will be visible to everyone else connected to the segment.

Creating the segment and connecting

Similarly to other forms of System V IPC, a shared memory segment is created and connected to via the **`shmget()`** call:

```
int shmget(key_t key, size_t size, int shmflg);
```

Upon successful completion, **`shmget()`** returns an identifier for the shared memory segment. The *`key`* argument should be created the same was as shown in the Message Queues document, using **`ftok()`**. The next argument, *`size`*, is the size in bytes of the shared memory segment. Finally, the *`shmflg`* should be set to the permissions of the segment bitwise-ORd with `IPC_CREAT` if you want to create the segment, but can be `0` otherwise. (It doesn't hurt to specify `IPC_CREAT` every time—it will simply connect you if the segment already exists.)

```
key_t key;
int shmid;
char *data;
key = ftok("/home/vishnu/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Here's an example call that creates a 1K segment with `644` permissions (`rw-r--r--`):
But how do you get a pointer to that data from the *`shmid`* handle? The answer is in the call **`shmat()`**, in the following section.

Attach me—getting a pointer to the segment

Before you can use a shared memory segment, you have to attach yourself to it using the **`shmat()`** call:

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

shmid is the shared memory ID you got from the call to **shmget()**. Next is *shmaddr*, which you can use to tell **shmat()** which specific address to use but you should just set it to `0` and let the OS choose the address for you. Finally, the *shmflg* can be set to `SHM_RDONLY` if you only want to read from it, `0` otherwise.

Here's a more complete example of how to get a pointer to a shared memory segment:

```
data = shmat(shmid, (void *)0, 0);
```

Now you have the pointer to the shared memory segment! Notice that **shmat()** returns a `void` pointer, and we're treating it, in this case, as a `char` pointer. You can treat is as anything you like, depending on what kind of data you have in there. Pointers to arrays of structures are just as acceptable as anything else.

Also, it's interesting to note that **shmat()** returns `-1` on failure. But how do you get `-1` in a `void` pointer? Just do a cast during the comparison to check for errors:

```
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1))
    perror("shmat");
```

Reading and Writing

Lets say you have the *data* pointer from the above example. It is a `char` pointer, so we'll be reading and writing chars from it. Furthermore, for the sake of simplicity, lets say the 1K shared memory segment contains a null-terminated string.

It couldn't be easier. Since it's just a string in there, we can print it like this:

```
printf("shared contents: %s\n", data);
```

And we could store something in it as easily as this:

```
printf("Enter a string: ");
gets(data);
```

Detaching from and deleting segments

When you're done with the shared memory segment, your program should detach itself from it using the **shmdt()** call:

```
int shmdt(void *shmaddr);
```

The only argument, *shmaddr*, is the address you got from **shmat()**. The function returns `-1` on error, `0` on success.

When you detach from the segment, it isn't destroyed. Nor is it removed when *everyone* detaches from it. You have to specifically destroy it using a call to **shmctl()**, similar to the control calls for the other System V IPC functions:

```
shmctl(shmid, IPC_RMID, NULL);
```

The above call deletes the shared memory segment, assuming no one else is attached to it. The **shmctl()** function does a lot more than this, though, and it worth looking into. As always, you can destroy the shared memory segment from the command line using the **ipcrm** Unix command. Also, be sure that you don't leave any usused shared memory segments sitting around wasting system resources. All the System V IPC objects you own can be viewed using the **ipcs** command.

```
/*** shmdemo.c --demonstrate shared memory IPC - execute with
and without command line arguments */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024 /* make it a 1K shared memory segment */
int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;
    if (argc > 2) {
         fprintf(stderr, "usage: shmdemo
         [data_to_write]\n"); exit(1);
    }
    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1)
         { perror("ftok");
         exit(1);
    }
    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
         { perror("shmget");
         exit(1);
    }
    /* attach to the segment to get a pointer to it:
    */ data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1))
         { perror("shmat");
         exit(1);
```

```
    }
    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
         printf("writing to segment: \"%s\"\n", argv[1]);
         strncpy(data, argv[1], SHM_SIZE);
    } else
         printf("segment contains: \"%s\"\n", data);
    /* detach from the segment: */
    if (shmdt(data) == -1) {
         perror("shmdt");
         exit(1);
    }
}
/*
[vishnu@mannava ~]$ cc shmdemo.c -o shmdemo
[vishnu@mannava ~]$ ./shmdemo hello
writing to segment: "hello"
[vishnu@mannava ~]$ ./shmdemo
segment contains: "hello"
*/
*/[vishnu@mannava ~]$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 98304      vishnu     600        393216     2          dest
0x00000000 131073     vishnu     600        393216     2          dest

0x00000000 163842     vishnu      600        393216     2         dest
*/
```

Program to Demonstrate System V Shared memory

Writing a message into shared memory

In this part of this recipe, we will learn how a message is written into shared memory.

How to do it…

1. Invoke the ftok function to generate an IPC key by supplying a filename and an ID.
2. Invoke the shmget function to allocate a shared memory segment that is associated with the key that was generated in step 1.
3. The size that's specified for the desired memory segment is 1024. Create a new memory segment with read and write permissions.
4. Attach the shared memory segment to the first available address in the system.
5. Enter a string that is then assigned to the shared memory segment.
6. The attached memory segment will be detached from the address space.

The writememory.c program for writing data into the shared memory is as follows:

/* writememory.c – Program to write data into the attached shared memory segment */

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
      char *str;
      int shmid;

      key_t key = ftok("sharedmem",'a');
      if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                  perror("shmget");
                  exit(1);
      }
            if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                  perror("shmat");
                  exit(1);
      }
      printf("Enter the string to be written in memory : ");
      gets(str);
            printf("String written in memory: %s\n",str);
```

```
        shmdt(str);
        return 0;
}
[vishnu@team-osd ~]$ vi writememory.c
[vishnu@team-osd ~]$ cc writememory.c -o writememory
[vishnu@team-osd ~]$ ./writememory
Enter the string to be written in memory : hello
String written in memory: hello
```

Reading a message from shared memory

In this part of the recipe, we will learn how the message that was written into shared memory is read and displayed on screen.

How to do it…

1. Invoke the ftok function to generate an IPC key. The filename and ID that are supplied should be the same as those in the program for writing content into shared memory.
2. Invoke the shmget function to allocate a shared memory segment. The size that's specified for the allocated memory segment is 1024 and is associated with the IPC key that was generated in step 1. Create the memory segment with read and write permissions.
3. Attach the shared memory segment to the first available address in the system.
4. The content from the shared memory segment is read and displayed on screen.
5. The attached memory segment is detached from the address space.
6. The shared memory identifier is removed from the system, followed by destroying the shared memory segment.

The readmemory.c program for reading data from shared memory is as follows:

```
/* readmemory.c – Program to read data from the attached shared memory segment */
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int shmid;
        char * str;
        key_t key = ftok("sharedmem",'a');
                if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                        perror("shmget");
                        exit(1);
        }
                if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                        perror("shmat");
                        exit(1);
        }
        printf("Data read from memory: %s\n",str);
                shmdt(str);
        shmctl(shmid,IPC_RMID,NULL);
        return 0;
}
[vishnu@team-osd ~]$ vi readmemory.c
[vishnu@team-osd ~]$ cc readmemory.c -o readmemory
[vishnu@team-osd ~]$ ./readmemory
Data read from memory: hello
```

Introduction to Pthreads

A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task.

In the traditional UNIX model, when a process needs something performed by another entity, it forks a child process and lets the child perform the processing. Most network servers under Unix, for example, are written this way. Although this paradigm has served well for many years, there are problems with fork:

- fork is expensive. Memory is copied from the parent to the child, all descriptors are duplicated in the child, and so on.
- Interprocess communication (IPC) is required to pass information between the parent and child after the fork. Information from the parent to the child before the fork is easy, since the child starts with a copy of the parent's data space and with a copy of all the parent's descriptors. But returning information from the child to the parent takes more work.

Threads help with both problems. Threads are sometimes called lightweight processes, since a thread is "lighter weight" than a process. That is, thread creation can be 10-100 times faster than process creation.

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes the problem of ***synchronization.*** But more than just the global variables are shared.

Benefits of threads:

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type. Each thread can then handle its event using a synchronous programming model. A synchronous programming model is much simpler than an asynchronous one.
- Some problems can be partitioned so that overall program throughput can be improved. A single process that has multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control. With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on the processing performed by each other.
- Similarly, interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

All threads within a process share:

- process instructions,
- most data,
- open files (e.g., descriptors),
- signal handlers and signal dispositions,
- current working directory, and
- user and group IDS.

But each thread has its own:

- thread ID,
- set of registers, including program counter and stack pointer,
- stack (for local variables and return addresses),
- errno,
- signal mask, and
- priority.

pthread_create Function:

When a program is started by exec, a single thread is created, called the ***initial thread*** or ***main thread.*** The pthread_create() function creates a new thread. The new thread commences execution by calling the function identified by start with the argument arg (i.e., start(arg)). The thread that calls pthread_create() continues execution with the next statement that follows the call.

The **arg argument** is declared as void *, meaning that we can pass a pointer to any type of object to the start function. Typically, arg points to a global or heap variable, but it can also be specified as NULL. If we need to pass multiple arguments to start, then arg can be specified as a pointer to a structure containing the arguments as separate fields. With judicious casting, we can even specify arg as an int.

```
#include<stdio.h>
main()
{
int myval = 5;
void* ptr = (void*)myval;
printf("%d",(int)ptr);
}
/*[root@ParallelProcessingLab klupplab]# ./a.out
5 */
```

The **return value of start** is likewise of type void *, and it can be employed in the same way as the arg argument.

The **thread argument** points to a buffer of type pthread_t into which the unique identifier for this thread is copied before pthread_create() returns. This identifier can be used in later Pthreads calls to refer to the thread.

Each thread has numerous ***attributes:*** its priority, its initial stack size, whether it should be a daemon thread or not, and so on. When a thread is created, we can specify these attributes by initializing a pthread_attr_t variable that overrides the default. We normally take the default, in which case, we specify the ***attr*** argument as a null pointer.
The **return value** from the Pthread functions is normally 0 if OK or nonzero on an error.

Pthread_self Function:

Each thread has an ID that identifies it within a given process. The thread ID is returned by **pthread_create,** and we saw that it was used by **pthread_join.** A thread fetches this value for itself using **pthread_self.**

Pthread_j oin Function:

We can wait for a given thread to terminate by calling **pthread_join.** Comparing threads to Unix processes, **pthread_create** is similar to **fork,** and **pthread_join** is similar to **waitpid.**
We must specify the tid of the thread for which we wish to wait. Unfortunately, we have no way to wait for any of our threads (similar to **waitpid** with a process ID argument of -1).
If the status pointer is nonnull, the return value from the thread (a pointer to some object) is stored in the location pointed to by status.

Pthread_detach Function:

A thread is either ***joinable*** (the default) or ***detached.*** When a joinable thread terminates, its thread ID and exit status are retained until another thread in the process calls pthread_join. But a detached thread is l i e a daemon process: when it terminates, all its resources are released, and we cannot wait for it to terminate. If one thread needs to know when another thread terminates, it is best to leave the thread as joinable. The pthread_detach function changes the specified thread so that it is detached. This function is commonly called by the thread that wants to detach itself, as in **Pthread_detach(pthread_self());**

pthread_exit Function:

One way for a thread to terminate is to call pthread_exit.
If the thread is not detached, its thread ID and exit status are retained for a later pthread_join by some other thread in the calling process. The pointer ***status*** must not point to an object that is local to the calling thread (e.g., an automatic variable in the thread start function), since that object disappears when **the** thread terminates.
A thread can terminate in two other ways:

- The function that started the thread (the third argument to pthread_create) can return. Since this function must be declared as returning a void pointer, that return value is the exit status of the thread.
- If the main function of the process returns or if any thread calls exit ( ) or _exit ( ), the process terminates immediately, including any threads that are still running.

```
#include <pthread.h>
void* worker( void* p ) {
  int* ip = (int*)p;
  printf("Hello world from worker
%i!\n",*ip);
}
int main() {
  pthread_t OtherThread[4];
  int i;
  for(i=0;i<4;i++) {
    pthread_create( &OtherThread[i],
NULL, worker, &i );
    sleep(1);
  }
}
/*
[vishnu@mannava PP]$ ./a.out
Hello world from worker 0!
Hello world from worker 1!
Hello world from worker 2!
Hello world from worker 3!
*/
```

```
#include <pthread.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
 printf("\n%d: Hello World!\n", threadid);
 pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0;t < NUM_THREADS;t++)
    {
         printf("Creating thread %d\n", t);
         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
         if (rc) {
            printf("ERROR; pthread_create() return code is %d\n", rc);
            exit(-1);
                  }
    }
     for(t=0;t < NUM_THREADS;t++)
     {
     pthread_join( threads[t], NULL);
     printf("Joining thread %d\n", t);
     }
    return 0;
}/*[vishnu@mannava PP]$ ./a.out
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
Creating thread 4
2: Hello World!
3: Hello World!
4: Hello World!
Joining thread 0
Joining thread 1
Joining thread 2
Joining thread 3
Joining thread 4*/
```

Comparison of process and thread primitives

Process primitive	Thread primitive	Description
fork	pthread_create	create a new flow of control

```
#include <unistd.h>
pid_t fork(void);
```
Returns: 0 in child, process ID of child in parent, -1 on error
```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);
```
Returns 0 on success, or a positive error number on error

exit	pthread_exit	exit from an existing flow of control

```
#include <stdlib.h>
void exit(int status);
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

waitpid	pthread_join	get exit status from flow of control

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```
Both return: process ID if OK, 0 (see later), or -1 on error
```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

getpid	pthread_self	get ID for flow of control

```
pid_t getpid(void);
```
Returns: process ID of calling process
```
pid_t getppid(void);
```
Returns: parent process ID of calling process
```
#include <pthread.h>
pthread_t pthread_self(void);
```
Returns: the thread ID of the calling thread

abort	pthread_cancel	request abnormal termination of flow of control

```
#void abort(void);
include <stdlib.h>
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

```
int pthread_detach(pthread_t tid);
int pthread_equal(pthread_t tid1, pthread_t tid2);
Returns: nonzero if equal, 0 otherwise
```

Thread Synchronization

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Pthreads data types	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_attr_t	Pthread attributes object

```
/* File:  pth_hello.c
Purpose: Illustrate basic use of pthreads:  create some threads, each of which prints a
message. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
const int MAX_THREADS = 64;
/* Global variable:  accessible to all threads */
int thread_count;
void Usage(char* prog_name);
void *Hello(void* rank);  /* Thread function */
/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
   /* Get number of threads from command line */
   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
   thread_handles = malloc (thread_count*sizeof(pthread_t));
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
   printf("Hello from the main thread\n");
   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   free(thread_handles);
   return 0;
}  /* main */
/*--------------------------------------------------------------------*/
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */
   printf("Hello from thread %ld of %d\n", my_rank, thread_count);
   return NULL;
}  /* Hello */
/*--------------------------------------------------------------------*/
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "0 < number of threads <= %d\n", MAX_THREADS);
   exit(0);
}  /* Usage */

/*[vishnu@mannava PP]$ cc -o pth_hello pth_hello.c -lpthread
[vishnu@mannava PP]$./pth_hello 4
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4 */
```

/*pthreads complex Passing arguments demo program */

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   8
char *messages[NUM_THREADS];
typedef struct thread_data {
   int thread_id;
   int sum;
   char *message;
} tdata_t;
void *PrintHello(void *threadarg) {
   int taskid, sum;
   char *hello_msg;
   struct tdata_t *my_data;
   //sleep(1);
   my_data = (tdata_t *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
```

```
   hello_msg = my_data->message;
   printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
   free(threadarg);
   pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t, sum;
    sum=0;
    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[2] = "Spanish: Hola al mundo";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";
  for(t=0;t<NUM_THREADS;t++) {
       tdata_t *tdata = (tdata_t *) malloc(sizeof(tdata_t));
      sum = sum + t;
      tdata->thread_id = t;
      tdata->sum = sum;
      tdata->message = messages[t];
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) tdata);
      if (rc) {
          printf("ERR; pthread_create() ret = %d\n", rc);
          exit(-1);
      }
  }
  return 0;
}
/*
[vishnu@mannava PP]$ ./a.out
Creating thread 0
Creating thread 1
Task 0: English: Hello World!
Creating thread 2
Creating thread 3
Task 2: Spanish: Hola al mundo
Task 1: French: Bonjour, le monde!
Creating thread 4
Task 3: Klingon: Nuq neH!
Creating thread 5
Creating thread 6
Creating thread 7
Task 6: Japan: Sekai e konnichiwa!
Task 4: German: Guten Tag, Welt!
/*  A simple child/parent signaling example. - main-signal.c     */
#include <stdio.h>
#include <pthread.h>
int done = 0;
void* worker(void* arg) {
    printf("this should print first\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create(&p, NULL, worker, NULL);
    while (done == 0)
      ;
    printf("this should print last\n");
    return 0;
}
/*
```

```
vishnu@mannava:~/threads$ cc main-signal.c -lpthread
vishnu@mannava:~/threads$ ./a.out
this should print first
this should print last
*/

/* A more efficient signaling via condition variables. - main-signal-cv.c    */
#include <stdio.h>
#include <pthread.h>
/* simple synchronizer: allows one thread to wait for another structure
"synchronizer_t" has all the needed data methods are:
   init (called by one thread)
   wait (to wait for a thread)
   done (to indicate thread is done)      */
typedef struct __synchronizer_t {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int done;
} synchronizer_t;

synchronizer_t s;

void signal_init(synchronizer_t *s) {
    pthread_mutex_init(&s->lock, NULL);
    pthread_cond_init(&s->cond, NULL);
    s->done = 0;
}

void signal_done(synchronizer_t *s) {
    pthread_mutex_lock(&s->lock);
    s->done = 1;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}

void signal_wait(synchronizer_t *s) {
    pthread_mutex_lock(&s->lock);
    while (s->done == 0)
      pthread_cond_wait(&s->cond, &s->lock);
    pthread_mutex_unlock(&s->lock);
}

void* worker(void* arg) {
    printf("this should print first\n");
    signal_done(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    signal_init(&s);
    pthread_create(&p, NULL, worker, NULL);
    signal_wait(&s);
    printf("this should print last\n");

    return 0;
}
/*
vishnu@mannava:~/threads$ cc main-signal-cv.c -lpthread
vishnu@mannava:~/threads$ ./a.out
this should print first
this should print last
*/
```

OSD 19CS2106S
Session – 36
Mutex and Concurrent Linked Lists
Lecture Notes

```
/*   Concurrent Linked Lists  */
#include <stdio.h>
#include <stdlib.h>
typedef struct __node_t {
    int             key;
    struct __node_t        *next;
} node_t;
// basic list structure (one used per list)
typedef struct __list_t {
node_t *head;
pthread_mutex_t lock;
} list_t;
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
       node_t *new = malloc(sizeof(node_t));
       if (new == NULL) {
              perror("malloc");
              pthread_mutex_unlock(&L->lock);
              return -1; // fail
       }
       new->key = key;
       new->next = L->head;
       L->head = new;
       pthread_mutex_unlock(&L->lock);
       return 0; // success
}
int List_Lookup(list_t *L, int key) {
       pthread_mutex_lock(&L->lock);
       node_t *curr = L->head;
       while (curr) {
              if (curr->key == key) {
              pthread_mutex_unlock(&L->lock);
              return 0; // success
              }
       curr = curr->next;
       }
       pthread_mutex_unlock(&L->lock);
       return -1; // failure
}
void List_Print(list_t *L) {
    node_t *tmp = L->head;
    while (tmp) {
       printf("%d ", tmp->key);
       tmp = tmp->next;
    }
    printf("\n");
}
int main(int argc, char *argv[])
{
    list_t mylist;
    List_Init(&mylist);
    List_Insert(&mylist, 10);
    List_Insert(&mylist, 30);
    List_Insert(&mylist, 5);
    List_Print(&mylist);
    printf("In List: 10? %d 20? %d\n",
         List_Lookup(&mylist, 10), List_Lookup(&mylist, 20));
    return 0;
}
/*
vishnu@mannava:~/threads$ ./a.out;./a.out
5 30 10
In List: 10? 0 20? -1
5 30 10
In List: 10? 0 20? -1
*/
```

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.

- If malloc() happens to *fail*, the code must also release the lock before failing the insert.
- This kind of exceptional control flow has been shown to be quite error prone.
- **Solution:** The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked List: Rewritten

```
1     void List_Init(list_t *L) {
2           L->head = NULL;
3           pthread_mutex_init(&L->lock, NULL);
4     }
5
```

```
6      void List_Insert(list_t *L, int key) {
7            // synchronization not needed
8            node_t *new = malloc(sizeof(node_t));
9            if (new == NULL) {
10                 perror("malloc");
11                 return;
12           }
13           new->key = key;
14
15           // just lock critical section
16           pthread_mutex_lock(&L->lock);
17           new->next = L->head;
18           L->head = new;
19           pthread_mutex_unlock(&L->lock);
20     }
```

```
21
22     int List_Lookup(list_t *L, int key) {
23           int rv = -1;
24           pthread_mutex_lock(&L->lock);
25           node_t *curr = L->head;
26           while (curr) {
27                 if (curr->key == key) {
28                       rv = 0;
29                       break;
30                 }
31                 curr = curr->next;
32           }
33           pthread_mutex_unlock(&L->lock);
34           return rv; // now both success and failure
35     }
```

Scaling Linked List:

- Hand-over-hand locking (lock coupling)
  - Add **a lock per node** of the list instead of having a single lock for the entire list.
  - When traversing the list,
    - First grabs the next node's lock.
    - And then releases the current node's lock.
  - Enable a high degree of concurrency in list operations.

However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*.

Pthreads Read-Write Locks

Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
The **first solution** only allows one thread to access the entire list at any instant
The **second** only allows one thread to access any given node at any instant.
A read-write lock is somewhat like a mutex except that it provides two lock functions.
The first lock function locks the read-write lock for reading, while the second locks it for writing.
So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.
If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.
Readerwriter locks are similar to mutexes, except that they allow for higher degrees of parallelism. With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time. Three states are possible with a readerwriter lock: locked in read mode, locked in write mode, and unlocked. Only one thread at a time can hold a readerwriter lock in write mode, but multiple threads can hold a readerwriter lock in read mode at the same time.
When a readerwriter lock is write-locked, all threads attempting to lock it block until it is unlocked. When a readerwriter lock is read-locked, all threads attempting to lock it in read mode are given access, but any threads attempting to lock it in write mode block until all the threads have relinquished their read locks. Although implementations vary, readerwriter locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode. This prevents a constant stream of readers from starving waiting writers.
Readerwriter locks are well suited for situations in which data structures are read more often than they are modified. When a readerwriter lock is held in write mode, the data structure it protects can be modified safely, since only one thread at a time can hold the lock in write mode. When the readerwriter lock is held in read mode, the data structure it protects can be read by multiple threads, as long as the threads first acquire the lock in read mode.
Readerwriter locks are also called sharedexclusive locks. When a readerwriter lock is read-locked, it is said to be locked in shared mode. When it is write-locked, it is said to be locked in exclusive mode.
As with mutexes, readerwriter locks must be initialized before use and destroyed before freeing their underlying memory.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
Both return: 0 if OK, error number on failure
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);          All return: 0 if OK, error number on failure
```

```
/* Pthreads - Increment Problem without
Synchronization*/
#include <pthread.h>
#include<stdio.h>
static int glob = 0;
/* Loop 'arg' times incrementing 'glob' */
static void *  threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;
   for (j = 0; j < loops; j++) {
      loc = glob;
      loc++;
      glob = loc;
   }
   return NULL;
}
Int main(int argc, char *argv[])
{
   pthread_t t1, t2;
   long loops, s;
   loops = strtol(argv[1], NULL, 10);
   s = pthread_create(&t1, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_create(&t2, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_join(t1, NULL);
   if (s != 0)
      perror("pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      perror("pthread_join");
   printf("glob = %d\n", glob);
   exit(0);
}
/*
 [vishnu@mannava PP]$ ./a.out 500
glob = 1000
[vishnu@mannava PP]$ ./a.out 50000
glob = 51227
[vishnu@mannava PP]$ ./a.out 5000000
glob = 6268601
[vishnu@mannava PP]$ ./a.out 50000000
glob = 59328262
*/
```

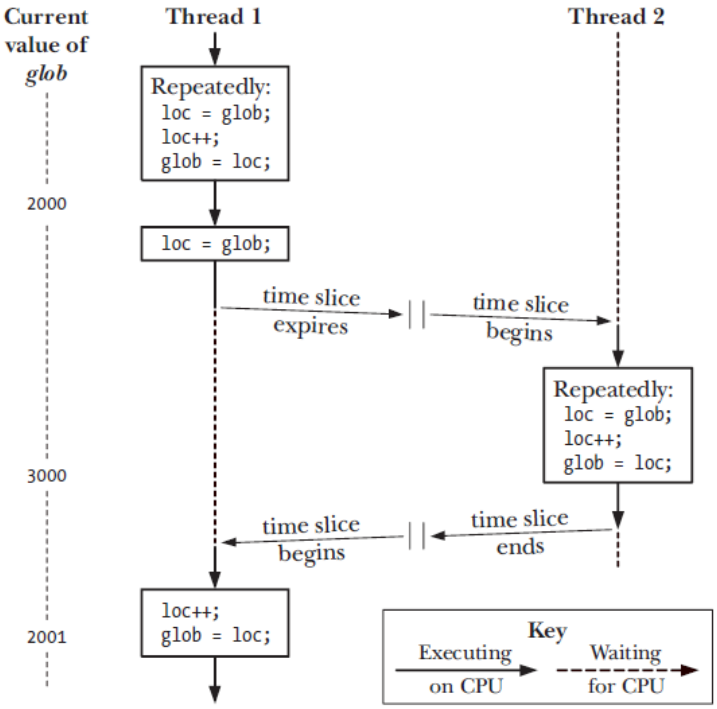
```
/* Pthreads - Increment Problem with Mutex -
Synchronization */
#include <pthread.h>
#include<stdio.h>
static int glob = 0;
static pthread_mutex_t mtx =
PTHREAD_MUTEX_INITIALIZER;
/* Loop 'arg' times incrementing 'glob' */
static void *  threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;
   for (j = 0; j < loops; j++) {
      pthread_mutex_lock(&mtx);
      loc = glob;
      loc++;
      glob = loc;
      pthread_mutex_unlock(&mtx);
   }
   return NULL;
}
int
main(int argc, char *argv[])
{
   pthread_t t1, t2;
   long loops, s;
   loops = strtol(argv[1], NULL, 10);
   s = pthread_create(&t1, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_create(&t2, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_join(t1, NULL);
   if (s != 0)
      perror("pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      perror("pthread_join");
   printf("glob = %d\n", glob);
   exit(0);
}
/*
[vishnu@mannava PP]$ ./a.out 500
glob = 1000
[vishnu@mannava PP]$ ./a.out 50000
glob = 100000
[vishnu@mannava PP]$ ./a.out 5000000
glob = 10000000
[vishnu@mannava PP]$ ./a.out 50000000
glob = 100000000
*/
```

```
/* Pthreads - Parallel Global Sum with Synchronization- Final*/
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#define no_threads 10
#define n 20
static long sum = 0;
long a[n];
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static void *slave(void *st)
{
long rank=(long*)st;
long my_n=n/no_threads;
long my_first_i = my_n *rank;
long my_last_i = my_first_i + my_n;
long i;
for (i = my_first_i; i < my_last_i; i++)
  {
    pthread_mutex_lock(&mtx);
    printf(" thread rank = %d Global sum it is fetching is %d \n", rank, sum);
    sum += *(a + i);
    printf(" hello from thread %d and its partial sum = %d\n",rank,sum);
    pthread_mutex_unlock(&mtx);
  }
return NULL; }
main()
{
long i;
pthread_t thread[10];
for (i = 0; i < n; i++)
      a[i] = i+1;
for (i = 0; i < no_threads; i++)
{
if (pthread_create(&thread[i], NULL, slave,(void*) i) != 0)
perror("Pthread create fails");
}
for (i = 0; i < no_threads; i++) {
if (pthread_join(thread[i], NULL) != 0)
 perror("Pthread join fails");  }
printf("The sum of numbers from %d to %d is %d\n",a[0],a[n-1], sum);
exit(0); } /*
[vishnu@mannava PP]$ ./a.out
 thread rank = 0 Global sum it is fetching is 0
 hello from thread 0 and its partial sum = 1
 thread rank = 0 Global sum it is fetching is 1
 hello from thread 0 and its partial sum = 3
 thread rank = 8 Global sum it is fetching is 3
 hello from thread 8 and its partial sum = 20
 thread rank = 8 Global sum it is fetching is 20
 hello from thread 8 and its partial sum = 38
 thread rank = 3 Global sum it is fetching is 38
 hello from thread 3 and its partial sum = 45
 thread rank = 3 Global sum it is fetching is 45
 hello from thread 3 and its partial sum = 53
 thread rank = 5 Global sum it is fetching is 53
 hello from thread 5 and its partial sum = 64
 thread rank = 5 Global sum it is fetching is 64
 hello from thread 5 and its partial sum = 76
 thread rank = 1 Global sum it is fetching is 76
 hello from thread 1 and its partial sum = 79
 thread rank = 1 Global sum it is fetching is 79
 hello from thread 1 and its partial sum = 83
 thread rank = 9 Global sum it is fetching is 83
 hello from thread 9 and its partial sum = 102
 thread rank = 9 Global sum it is fetching is 102
 hello from thread 9 and its partial sum = 122
 thread rank = 4 Global sum it is fetching is 122
 hello from thread 4 and its partial sum = 131
 thread rank = 4 Global sum it is fetching is 131
 hello from thread 4 and its partial sum = 141
 thread rank = 7 Global sum it is fetching is 141
 hello from thread 7 and its partial sum = 156
 thread rank = 7 Global sum it is fetching is 156
 hello from thread 7 and its partial sum = 172
 thread rank = 6 Global sum it is fetching is 172
 hello from thread 6 and its partial sum = 185
 thread rank = 6 Global sum it is fetching is 185
 hello from thread 6 and its partial sum = 199
 thread rank = 2 Global sum it is fetching is 199
 hello from thread 2 and its partial sum = 204
 thread rank = 2 Global sum it is fetching is 204
 hello from thread 2 and its partial sum = 210
 The sum of numbers from 1 to 20 is 210 */
```

```
/* Two threads update a global shared variable without synchronization*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int max;
volatile int counter = 0; // shared global variable
void * mythread(void *arg)
{   char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
       counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
int main(int argc, char *argv[])
{      if (argc != 2) {
       fprintf(stderr, "usage: main-first <loopcount>\n");
       exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,(unsigned int) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n", counter, max*2);
    return 0;
}/* [vishnu@localhost ~]$ cc t1.c -lpthread
[vishnu@localhost ~]$ ./a.out 10
main: begin [counter = 0] [60106c]
B: begin [addr of i: 0x7f7da8bf6f44]
B: done
A: begin [addr of i: 0x7f7da93f7f44]
A: done
main: done
 [counter: 20]
 [should: 20]                                    */
```

Incorrectly incrementing a global variable from two threads: Refer Increment Problem in Synchronization - I

Two threads incrementing a global variable without synchronization



The problem here results from execution sequences such as the following:

1. Thread 1 fetches the current value of glob into its local variable loc. Let's assume that the current value of glob is 2000.

2. The scheduler time slice for thread 1 expires, and thread 2 commences execution.

3. Thread 2 performs multiple loops in which it fetches the current value of glob into its local variable loc, increments loc, and assigns the result to glob. In the first of these loops, the value fetched from glob will be 2000. Let's suppose that by the time the time slice for thread 2 has expired, glob has been increased to 3000.

4. Thread 1 receives another time slice and resumes execution where it left off. Having previously (step 1) copied the value of glob (2000) into its loc, it now increments loc and assigns the result (2001) to glob. At this point, the effect of the increment operations performed by thread 2 is lost.

Mutex Deadlocks

Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex. When more than one thread is locking the same set of mutexes, deadlock situations can arise. Figure shows an example of a deadlock in which each thread successfully locks one mutex, and then tries to lock the mutex that the other thread has already locked. Both threads will remain blocked indefinitely.

Thread A	**Thread B**
1. pthread_mutex_lock(mutex1); 2. pthread_mutex_lock(mutex2);	1. pthread_mutex_lock(mutex2); 2. pthread_mutex_lock(mutex1);

blocks	blocks

Figure: A deadlock when two threads lock two mutexes

The simplest way to avoid such deadlocks is to define a mutex hierarchy. When threads can lock the same set of mutexes, they should always lock them in the same order. For example, in the scenario in Figure, the deadlock could be avoided if the two threads always lock the mutexes in the order mutex1 followed by mutex2. Sometimes, there is a logically obvious hierarchy of mutexes. However, even if there isn't, it may be possible to devise an arbitrary hierarchical order that all threads should follow.

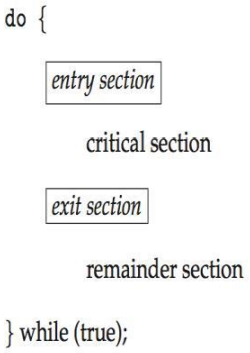
Critical section problem

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

General structure of process ***Pi***

Algorithm for Process Pi

```
do {
        while (turn == j);
                critical section
                        turn = j;
                remainder section
        } while (true);
```



Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***Pi*** is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

Semaphores

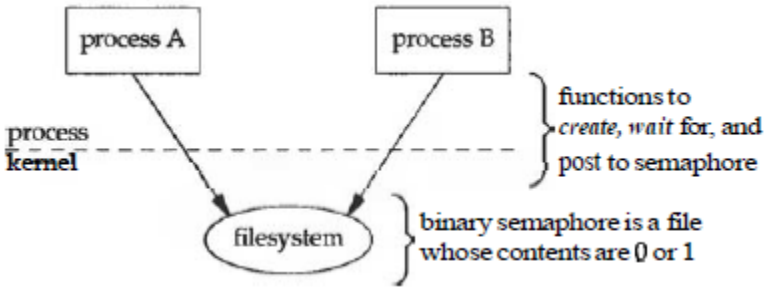
A semaphore is a primitive used to provide synchronization between various processes or between the various threads in a given process. We look at three types of semaphores in this section.

- POSIX named semaphores are identified by POSIX **IPC** names and can be used to synchronize processes or threads. By calling sem_open() with the same name, unrelated processes can access the same semaphore.
- POSIX memory-based semaphores are stored in shared memory and can be used to synchronize processes or threads. This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads. When shared between processes, the semaphore must reside in a region of (System V, POSIX, or mmap()) shared memory. When shared between threads, the semaphore may reside in an area of memory shared by the threads (e.g., on the heap or in a global variable).
- System V semaphores are maintained in the kernel and can be used to synchronize processes or threads.

For now, we concern ourselves with synchronization between different processes. We first consider a *binary semaphore:* a semaphore that can assume only the values 0 or 1.

We show that the semaphore is maintained by the kernel (which is true for System V semaphores) and that its value can be 0 or 1.

POSIX semaphores need not be maintained in the kernel. Also, POSIX semaphores are identified by names that might correspond to pathnames in the files system. Therefore, Figure is a more realistic picture of what is termed a *POSIX named semaphore.*



A Posix named binary semaphore being used by two processes

We must make one qualification with regard to Figure: although POSIX named semaphores are identified by names that might correspond to pathnames in the filesystem, nothing requires that they actually be stored in a file in the filesystem.

Three operations that a process can perform on a semaphore:

1. ***Create*** a semaphore. This also requires the caller to specify the initial value, which for a binary semaphore is often 1, but can be 0.
2. ***Wait*** for a semaphore. This tests the value of the semaphore, waits (blocks) if the value is less than or equal to 0, and then decrements the semaphore value once it is greater than 0. This can be summarized by the pseudocode

```
while (semaphore-value <= 0)
        ; /* wait; i-e., block the thread or process */
semaphore-value--;
        /* we have the semaphore */
```

The fundamental requirement here is that the test of the value in the while statement, and its subsequent decrement (if its value was greater than 0), must be done as an *atomic* operation with respect to other threads or processes accessing this semaphore.

3. ***Post*** to a semaphore. This increments the value of the semaphore and can be summarized by the pseudocode

If any processes are blocked, waiting for this semaphore's value to be greater than 0, one of those processes can now be awoken. As with the wait code just shown, this post operation must also be atomic with regard to other processes accessing the semaphore. Obviously, the actual semaphore code has more details than we show in the pseudocode for the wait and post operations: namely how to queue all the processes that are waiting for a given semaphore and then how to wake up one (of the possibly many processes) that is waiting for a given semaphore to be posted to. Fortunately, these details are handled by the implementation. Notice that the pseudocode shown does not assume a binary semaphore with the values 0 and 1. The code works with semaphores that are initialized to any nonnegative value. These are called ***counting semaphores.*** These are normally initialized to some value *N,* which indicates the number of resources (say buffers) available. We show examples of both binary semaphores and counting semaphores throughout the handout

We often differentiate between a binary semaphore and a counting semaphore, and we do so for our own edification. No difference exists between the two in the system code that implements a semaphore.

A binary semaphore can be used for mutual exclusion, just like a mutex.

initialize mutex;
`pthread_mutex_lock(&mutex);`
critical region
`pthread_mutex_unlock(&mutex);`
	initialize to 1;semaphore
	`Sem_wait(&sem);`
	critical region
	`sem_post (&sem);`

Comparison of mutex and semaphore to solve mutual exclusion problem.

We initialize the semaphore to 1. The call to **`sem_wait`** waits for the value to be greater than 0 and then decrements the value. The call to **`sem_post`** increments the value (from 0 to 1) and wakes up any threads blocked in a call to **`sem_wait`** for this semaphore.

Although semaphores can be used like a mutex, semaphores have a feature not provided by mutexes: a mutex must always be unlocked by the thread that locked the mutex, while a semaphore post need not be performed by the same thread that did the semaphore wait.

We can show an example of this feature using two binary semaphores and a **simplified version of the producer-consumer problem**: a producer that places an item into a shared buffer and a consumer that removes the item. For simplicity, assume that the buffer holds one item.

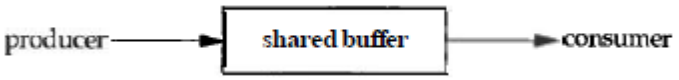


Figure: Simple producer-consumer problem with a shared buffer.

Shows the pseudocode for the producer and consumer.

Producer	**Consumer**

initialize semaphore `get` to 0;
initialize semaphore `put` to 1;
`sem_wait (&put) ;`
put data into buffer
`sem_post (&get) ;`
	`for(;;)(`
	`sem_wait(&get);`
	process data in buffer
	`sem_post (&put) ;`

Figure: Pseudocode for simple producer+mnsumer.

The semaphore **`put`** controls whether the producer can place an item into the shared buffer, and the semaphore **`get`** controls whether the consumer can remove an item from the shared buffer. The steps that occur over time are as follows:

1. The producer initializes the buffer and the two semaphores.
2. Assume that the consumer then runs. It blocks in its call to **`sem_wait`** because the value of **`get`** is 0.
3. Sometime later, the producer starts. When it calls **`sem_wait,`** the value of **`put`** is decremented from 1 to 0, and the producer places an item into the buffer. It then calls **`sem_post`** to increment the value of **`get`** from 0 to 1. Since a thread is blocked on this semaphore (the consumer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the producer continues to run. The producer then blocks in its call to **`sem_wait`** at the top of the **`for`** loop, because the value of **`put`** is 0. The producer must wait until the consumer empties the buffer.
4. The consumer returns from its call to **`sem_wait,`** which decrements the value of the **`get`** semaphore from 1 to 0. It processes the data in the buffer, and calls **`sem_post,`** which increments the value of **`put`** from 0 to 1. Since a thread is blocked on this semaphore (the producer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the consumer continues to run. The consumer then blocks in its call to **`sem_wait,`** at the top of the **`for`** loop, because the value of **`get`** is 0.
5. The producer returns from its call to sem_wait, places data into the buffer, and this scenario just continues.

We assumed that each time **`sem_post`** was called, even though a process was waiting and was then marked as ready-to-run, the caller continued. Whether the caller continues or whether the thread that just became ready runs does not matter (you should assume the other scenario and convince yourself of this fact).

Named Semaphores

To work with a named semaphore, we employ the following functions:

- The sem_open() function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode, unsigned int value */ );
Returns pointer to semaphore on success, or SEM_FAILED on error
```

- The sem_post(sem) and sem_wait(sem) functions respectively increment and decrement a semaphore's value.

```
int sem_wait(sem_t *sem);
Returns 0 on success, or –1 on error
int sem_trywait(sem_t *sem);
```

```
Returns 0 on success, or –1 on error
int sem_post(sem_t *sem);
Returns 0 on success, or –1 on error
```

- The sem_getvalue() function retrieves a semaphore's current value.

```
int sem_getvalue(sem_t *sem, int *sval);
Returns 0 on success, or –1 on error
```

- The sem_close() function removes the calling process's association with a semaphore that it previously opened.

```
int sem_close(sem_t *sem);
Returns 0 on success, or –1 on error
```

- The sem_unlink() function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

```
int sem_unlink(const char *name);
Returns 0 on success, or –1 on error
```

Unnamed Semaphores

Unnamed semaphores (also known as memory-based semaphores) are variables of type sem_t that are stored in memory allocated by the application. The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share. Operations on unnamed semaphores use the same functions (sem_wait(), sem_post(), sem_getvalue(), and so on) that are used to operate on named semaphores.
In addition, two further functions are required:

- The sem_init() function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
Returns 0 on success, or –1 on error
```

- The sem_destroy(sem) function destroys a semaphore. These functions should not be used with named semaphores.

```
int sem_destroy(sem_t *sem);
Returns 0 on success, or –1 on error
```

The POSIX semaphore interface is simpler than the System V semaphore interface. Semaphores are allocated and operated on individually, and the wait and post operations adjust a semaphore's value by one.
POSIX semaphores have a number of advantages over System V semaphores, but they are somewhat less portable. For synchronization within multithreaded applications, mutexes are generally preferred over semaphores.

The Bounded-Buffer Problem or Producer-Consumer Problem

producer-consurner problem solutions in which multple producer threads fill an array that is processed by one consumer thread.
1. In our first solution, the consumer started only after the producers were finished, and we were able to solve this synchronization problem using a single mutex (to synchronize the producers). **See code at page number: 12**
2. In our next solution (the consumer started before the producers were finished, and this required a mutex (to synchronize the producers) along with a condition variable and its mutex (to synchronize the consumer with the producers). **See code at page number: 13**
We now extend the producer-consumer problem by using the shared buffer as a circular buffer: after the producer fills the final entry (buff [NBUFF-1]), it goes back and fills the first entry (buff [0])**,** and the consumer does the same. This adds another synchronization problem in that the producer must not get ahead of the consumer. We still assume that the producer and consumer are threads, but they could also be processes, assuming that some way existed to share the buffer between the processes (e.g., shared memory). **See code at page number: 12**
Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:
1. The consumer cannot try to remove an item from the buffer when the buffer is empty.
2. The producer cannot try to place an item into the buffer when the buffer is full.
3. Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.
Our solution using semaphores demonstrates three different types of semaphores:
1. A binary semaphore named mutex protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously we could use a real mutex for this, instead of a binary semaphore.)
2. A counting semaphore named nempty counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (NBUFF).
3. A counting semaphore named nstored counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty.

The Readers-Writers Problem

In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.

- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
    - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

- The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:
  - readcount is used by the reader processes, to count the number of readers currently accessing the data.
  - mutex is a semaphore used only by the readers for controlled access to readcount.
  - rw_mutex is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch rw_mutex.
  - Note that the first reader to come along will block on rw_mutex if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

Condition Variables

A mutex prevents multiple threads from accessing a shared variable at the same time. A condition variable allows one thread to inform other threads about changes in the state of a shared variable (or other shared resource) and allows the other threads to wait (block) for such notification.

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
```

A simple example that doesn't use condition variables serves to demonstrate why they are useful. Suppose that we have a number of threads that produce some "result units" that are consumed by the main thread, and that we use a mutex-protected variable, avail, to represent the number of produced units awaiting consumption:

```
/* prod_no_condvar.c -- A simple POSIX threads producer-consumer example that doesn't use a
condition variable. See also prod_condvar.c. */
#include <time.h>
#include <pthread.h>
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
static void *
threadFunc(void *arg)
{   int cnt = atoi((char *) arg);
    int s, j;
    for (j = 0; j < cnt; j++) {
        sleep(1);
        /* Code to produce a unit omitted */
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        avail++;        /* Let consumer know another unit is available */
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
    }
    return NULL;
}
void quit (char *message, int exit_status) {
perror(message);
exit(exit_status);
}
int main(int argc, char *argv[])
{   pthread_t tid;
    typedef enum { FALSE, TRUE } Boolean;
    int s, j;
    int totRequired;            /* Total number of units that all threads will produce */
    int numConsumed;            /* Total units so far consumed */
    Boolean done;
    time_t t;
    t = time(NULL);
    /* Create all threads */
    totRequired = 0;
    for (j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);
        s = pthread_create(&tid, NULL, threadFunc, argv[j]);
        if (s != 0)
            quit("pthread_create",s);
    } /* Use a polling loop to check for available units */
```

```
    numConsumed = 0;
    done = FALSE;
    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        while (avail > 0) {             /* Consume all available units */
            /* Do something with produced unit */
            numConsumed ++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t), numConsumed);
            done = numConsumed >= totRequired;
        }
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        if (done)
            break;
        /* Perhaps do other work here that does not require mutex lock */
    }
    exit(0);
}/* [vishnu@localhost ~]$ ./a.out 4 5 1
T=1: numConsumed=1
T=1: numConsumed=2
T=1: numConsumed=3
T=2: numConsumed=4
T=2: numConsumed=5
T=3: numConsumed=6
T=3: numConsumed=7
T=4: numConsumed=8
T=4: numConsumed=9
T=5: numConsumed=10                                 */
```

The above code works, but it wastes CPU time, because the main thread continually loops, checking the state of the variable avail. A condition variable remedies this problem. It allows a thread to sleep (wait) until another thread notifies (signals) it that it must do something (i.e., that some “condition” has arisen that the sleeper must now respond to).

A condition variable is always used in conjunction with a mutex. The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable’s state.

Statically Allocated Condition Variables

As with mutexes, condition variables can be allocated statically or dynamically.

A condition variable has the type pthread_cond_t. As with a mutex, a condition variable must be initialized before use. For a statically allocated condition variable, this is done by assigning it the value PTHREAD_COND_INITIALIZER, as in the following example:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Signaling and Waiting on Condition Variables

The principal condition variable operations are signal and wait. The signal operation is a notification to one or more waiting threads that a shared variable’s state has changed. The wait operation is the means of blocking until such a notification is received.

The pthread_cond_signal() and pthread_cond_broadcast() functions both signal the condition variable specified by cond. The pthread_cond_wait() function blocks a thread until the condition variable cond is signaled.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

All return 0 on success, or a positive error number on error

The difference between pthread_cond_signal() and pthread_cond_broadcast() lies in what happens if multiple threads are blocked in pthread_cond_wait(). With pthread_cond_signal(), we are simply guaranteed that at least one of the blocked threads is woken up; with pthread_cond_broadcast(), all blocked threads are woken up.

Using pthread_cond_broadcast() always yields correct results (since all threads should be programmed to handle redundant and spurious wake-ups), but pthread_cond_signal() can be more efficient. However, pthread_cond_signal() should be used only if just one of the waiting threads needs to be woken up to handle the change in state of the shared variable, and it doesn’t matter which one of the waiting threads is woken up. This scenario typically applies when all of the waiting threads are designed to perform the exactly same task. Given these assumptions, pthread_cond_signal() can be more efficient than pthread_cond_broadcast(), because it avoids the following possibility:

1. All waiting threads are awoken.

2. One thread is scheduled first. This thread checks the state of the shared variable(s) (under protection of the associated mutex) and sees that there is work to be done. The thread performs the required work, changes the state of the shared variable(s) to indicate that the work has been done, and unlocks the associated mutex.

3. Each of the remaining threads in turn locks the mutex and tests the state of the shared variable. However, because of the change made by the first thread, these threads see that there is no work to be done, and so unlock the mutex and go back to sleep (i.e., call pthread_cond_wait() once more). By contrast, pthread_cond_broadcast() handles the case where the waiting threads are designed to perform different tasks (in which case they probably have different predicates associated with the condition variable).

A condition variable holds no state information. It is simply a mechanism for communicating information about the application’s state. If no thread is waiting on the condition variable at the time that it is signaled, then the signal is lost. A thread that later waits on the condition variable will unblock only when the variable is signaled once more.

pthread_cond_wait() Steps

We noted earlier that a condition variable always has an associated mutex. Both of these objects are passed as arguments to pthread_cond_wait(), which performs the following steps:

- unlock the mutex specified by mutex;
- block the calling thread until another thread signals the condition variable cond; and
- relock mutex.

Using a condition variable in the producer-consumer example

```
/* prod_condvar.c -- A simple POSIX threads producer-consumer example using a condition
variable. */
#include <time.h>
#include <pthread.h>
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;
static void *
threadFunc(void *arg)
{   int cnt = atoi((char *) arg);
    int s, j;
    for (j = 0; j < cnt; j++) {
        sleep(1);
        /* Code to produce a unit omitted */
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        avail++;        /* Let consumer know another unit is available */
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        s = pthread_cond_signal(&cond);         /* Wake sleeping consumer */
        if (s != 0)
            quit("pthread_cond_signal",s);
    }
    return NULL;
}
void quit (char *message, int exit_status) {
perror(message);
exit(exit_status);
}
int main(int argc, char *argv[])
{   pthread_t tid;
    typedef enum { FALSE, TRUE } Boolean;
    int s, j;
    int totRequired;            /* Total number of units that all threads will produce */
    int numConsumed;            /* Total units so far consumed */
    Boolean done;
    time_t t;
    t = time(NULL);
    /* Create all threads */
    totRequired = 0;
    for (j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);
        s = pthread_create(&tid, NULL, threadFunc, argv[j]);
        if (s != 0)
            quit("pthread_create",s);
    }
    /* Loop to consume available units */
    numConsumed = 0;
    done = FALSE;
    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        while (avail == 0) {            /* Wait for something to consume */
            s = pthread_cond_wait(&cond, &mtx);
            if (s != 0)
                quit("pthread_cond_wait",s);
        }
        /* At this po,int, 'mtx' is locked... */
        while (avail > 0) {             /* Consume all available units */
            /* Do something with produced unit */
            numConsumed ++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t), numConsumed);
            done = numConsumed >= totRequired;
        }
        s = pthread_mutex_unlock(&mtx);
```

```
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        if (done)
            break;
        /* Perhaps do other work here that does not require mutex lock */
    }
    exit(0);
}/* [vishnu@localhost ~]$ cc prod_condvar.c -lpthread
[vishnu@localhost ~]$ ./a.out 4 5
T=1: numConsumed=1
T=1: numConsumed=2
T=2: numConsumed=3
T=2: numConsumed=4
T=3: numConsumed=5
T=3: numConsumed=6
T=4: numConsumed=7
T=4: numConsumed=8
T=5: numConsumed=9                                          */
```

In the above code, both accesses to the shared variable must be mutex-protected for the reasons that we explained earlier. In other words, there is a natural association of a mutex with a condition variable:

1. The thread locks the mutex in preparation for checking the state of the shared variable.
2. The state of the shared variable is checked.
3. If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
4. When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.

The pthread_cond_wait() function automatically performs the mutex unlocking and locking required in the last two of these steps. In the third step, releasing the mutex and blocking on the condition variable are performed atomically. In other words, it is not possible for some other thread to acquire the mutex and signal the condition variable before the thread calling pthread_cond_wait() has blocked on the condition variable.

In the producer code shown earlier, we called pthread_mutex_unlock(), and then called pthread_cond_signal(); that is, we first unlocked the mutex associated with the shared variable, and then signaled the corresponding condition variable.

Testing a Condition Variable's Predicate

Each condition variable has an associated predicate involving one or more shared variables. For example, in the code segment in the preceding section, the predicate associated with cond is (avail == 0). This code segment demonstrates a general design principle: a pthread_cond_wait() call must be governed by a while loop rather than an if statement. This is so because, on return from pthread_cond_wait(), there are no guarantees about the state of the predicate; therefore, we should immediately recheck the predicate and resume sleeping if it is not in the desired state.

Semaphores	*Condition Variables*
Can be used anywhere in a program, but should not be used in a monitor	**Can only be used in monitors**
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	**Wait() always blocks the caller.**
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	**Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.**
If Signal() releases a blocked thread, the caller and the released thread *both* continue.	**If Signal() releases a blocked thread, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.**

```
/* prodcons-mutex.c - Producer Consumer problem using mutex and pthreads */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <sys/types.h>
#define     MAXNITEMS         1000000
#define     MAXNTHREADS       100
int         nitems;                 /* read-only by producer and consumer */
struct {
  pthread_mutex_t mutex;
  int buff[MAXNITEMS];
  int nput;
  int nval;
} shared ={PTHREAD_MUTEX_INITIALIZER};
void  *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
      shared.nput=0;
      shared.nval=0;
      int               i, nthreads, count[MAXNTHREADS];
      pthread_t   tid_produce[MAXNTHREADS], tid_consume;
      if (argc != 3)
      {
            printf("usage: prodcons1 <#items> <#threads>");
            exit(1);
      }
      nitems = atoi(argv[1]);
      nthreads = atoi(argv[2]);
      pthread_setconcurrency(nthreads);
            /* 4start all the producer threads */
      for (i = 0; i < nthreads; i++) {
            count[i] = 0;
            pthread_create(&tid_produce[i], NULL, produce, &count[i]);
      }
            /* 4wait for all the producer threads */
      for (i = 0; i < nthreads; i++) {
            pthread_join(tid_produce[i], NULL);
            printf("count[%d] = %d\n", i, count[i]);
      }
            /* 4start, then wait for the consumer thread */
      pthread_create(&tid_consume, NULL, consume, NULL);
      pthread_join(tid_consume, NULL);
      exit(0);
}
/* end main */
/* include producer */
void * produce(void *arg)
{
pthread_t tid;
int i=*((int *) arg);
      for ( ; ; ) {
            pthread_mutex_lock(&shared.mutex);
            tid=pthread_self();
            printf("threadid=%u\n", (unsigned int) tid);
            if (shared.nput >= nitems) {
                  pthread_mutex_unlock(&shared.mutex);
                  return(NULL);           /* array is full, we're done */
            }
            shared.buff[shared.nput] = shared.nval;
            printf("buff[%d] = %d\n", shared.nput, shared.buff[shared.nput]);
            shared.nput++;
            shared.nval++;
            *((int *) arg) += 1;
            pthread_mutex_unlock(&shared.mutex);
            printf("shared.nput=%d, shared.nval=%d,count[%u] =
```

```
            %d\n",shared.nput,shared.nval, i, *((int *) arg));
      }
}
void * consume(void *arg)
{
      int   i;
      for (i = 0; i <nitems; i++) {
            if (shared.buff[i]! = i)
                  printf("buff[%d] = %d\n", i, shared.buff[i]);
      }
      return(NULL);
}
/* end producer */
/*
[vishnu@mannava mutex]$ ./a.out 4 5
threadid=3078474608
buff[0] = 0
shared.nput=1, shared.nval=1,count[0] = 1
threadid=3078474608
buff[1] = 1
shared.nput=2, shared.nval=2,count[0] = 2
threadid=3078474608
buff[2] = 2
shared.nput=3, shared.nval=3,count[0] = 3
threadid=3078474608
buff[3] = 3
shared.nput=4, shared.nval=4,count[0] = 4
threadid=3078474608
count[0] = 4
threadid=3067984752
threadid=3057494896
threadid=3047005040
count[1] = 0
count[2] = 0
count[3] = 0
threadid=3036515184
count[4] = 0
*/
```

/* prodcons-semaphores.c Producer Consumer problem using semaphores and pthreads */

```
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <semaphore.h>
#include    <sys/types.h>
#define     NBUFF       10
#define     SEM_MUTEX   "mutex"           /* these are args to px_ipc_name() */
#define     SEM_NEMPTY  "nempty"
#define     SEM_NSTORED "nstored"
int nitems; /* read-only by producer and consumer */
struct {    /* data shared by producer and consumer */
  int buff[NBUFF];
  sem_t     *mutex;
  sem_t     *nempty;
  sem_t     *nstored;
} shared;
void *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
      pthread_t   tid_produce, tid_consume;
      if (argc != 2)
      {
            printf("usage: prodcons1 <#items>");
            exit(1);
      }
      nitems = atoi(argv[1]);
```

```
        /* 4create three semaphores */
    shared.mutex = sem_open(SEM_MUTEX, O_CREAT | O_EXCL, 0644, 1);
    shared.nempty = sem_open(SEM_NEMPTY, O_CREAT | O_EXCL, 0644, NBUFF);
    shared.nstored = sem_open(SEM_NSTORED, O_CREAT | O_EXCL, 0644, 0);
        /* 4create one producer thread and one consumer thread */
    pthread_setconcurrency(2);
    pthread_create(&tid_produce, NULL, produce, NULL);
    pthread_create(&tid_consume, NULL, consume, NULL);
        /* 4wait for the two threads */
    pthread_join(tid_produce, NULL);
    pthread_join(tid_consume, NULL);
        /* 4remove the semaphores */
    sem_unlink(SEM_MUTEX);
    sem_unlink(SEM_NEMPTY);
    sem_unlink(SEM_NSTORED);
    exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
    int         i;
    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nempty);      /* wait for at least 1 empty slot */
        sem_wait(shared.mutex);
        shared.buff[i % NBUFF] = i;   /* store i into circular buffer */
        sem_post(shared.mutex);
        sem_post(shared.nstored);     /* 1 more stored item */
    }
    return(NULL);
}
void *consume(void *arg)
{
    int         i;
    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nstored);           /* wait for at least 1 stored item */
        sem_wait(shared.mutex);
        if (shared.buff[i % NBUFF] == i)
            printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
        sem_post(shared.mutex);
        sem_post(shared.nempty);            /* 1 more empty slot */
    }
    return(NULL);
}
/* end prodcons */
/*[vishnu@mannava pxsem]$ ./a.out 5
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
*/
```

```
/* Implementation of Producer consumer problem using condition
variables */
/* include globals */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <sys/types.h>
#define     MAXNITEMS           1000000
#define     MAXNTHREADS         100
            /* globals shared by threads */
int         nitems;                         /* read-only by producer and consumer
*/
int         buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  int                   nput;  /* next index to store */
  int                   nval;  /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };
struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int                   nready;       /* number ready for consumer */
} nready = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER };
/* end globals */
```

```
void   *produce(void *), *consume(void *);
/* include main */
int main(int argc, char **argv)
{
	int			i, nthreads, count[MAXNTHREADS];
	pthread_t	tid_produce[MAXNTHREADS], tid_consume;
	if (argc != 3)
	{
		printf("usage: prodcons6 <#items> <#threads>");
	}
	nitems = atoi(argv[1]);
	nthreads = atoi(argv[2]);
	pthread_setconcurrency(nthreads + 1);
		/* 4create all producers and one consumer */
	for (i = 0; i < nthreads; i++) {
		count[i] = 0;
		pthread_create(&tid_produce[i], NULL, produce, &count[i]);
	}
	pthread_create(&tid_consume, NULL, consume, NULL);
		/* wait for all producers and the consumer */
	for (i = 0; i < nthreads; i++) {
		pthread_join(tid_produce[i], NULL);
		printf("count[%d] = %d\n", i, count[i]);
	}
	pthread_join(tid_consume, NULL);
	exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
	for ( ; ; ) {
		pthread_mutex_lock(&put.mutex);
		if (put.nput >= nitems) {
			pthread_mutex_unlock(&put.mutex);
			return(NULL);		/* array is full, we're done */
		}
		buff[put.nput] = put.nval;
		put.nput++;
		put.nval++;
		pthread_mutex_unlock(&put.mutex);
		pthread_mutex_lock(&nready.mutex);
		if (nready.nready == 0)
			pthread_cond_signal(&nready.cond);
		nready.nready++;
		pthread_mutex_unlock(&nready.mutex);
		*((int *) arg) += 1;
	}
}
void *consume(void *arg)
{
	int			i;
	for (i = 0; i < nitems; i++) {
		pthread_mutex_lock(&nready.mutex);
		while (nready.nready == 0)
			pthread_cond_wait(&nready.cond, &nready.mutex);
		nready.nready--;
		pthread_mutex_unlock(&nready.mutex);

		if (buff[i] == i)
			printf("buff[%d] = %d\n", i, buff[i]);
	}
	return(NULL);
}/* end prodcons */
/*[vishnu@mannava mutex]$ ./a.out 5 4
count[0] = 4
count[1] = 1
count[2] = 0
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
count[3] = 0
*/
```

```
/* prodcons-sem.c - Producer Consumer problem using POSIX Semaphores */
/* include main */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <semaphore.h>
#include    <sys/types.h>
#define     NBUFF       10
#define     SEM_MUTEX   "mutex"           /* these are args to px_ipc_name() */
#define     SEM_NEMPTY  "nempty"
#define     SEM_NSTORED "nstored"
int nitems; /* read-only by producer and consumer */
struct {    /* data shared by producer and consumer */
  int buff[NBUFF];
  sem_t     *mutex;
  sem_t     *nempty;
  sem_t     *nstored;
} shared;
void *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
      pthread_t   tid_produce, tid_consume;
      if (argc != 2)
      {
            printf("usage: prodcons1 <#items>");
            exit(1);
      }
      nitems = atoi(argv[1]);
            /* 4create three semaphores */
      shared.mutex = sem_open(SEM_MUTEX, O_CREAT | O_EXCL, 0644, 1);
      shared.nempty = sem_open(SEM_NEMPTY, O_CREAT | O_EXCL, 0644, NBUFF);
      shared.nstored = sem_open(SEM_NSTORED, O_CREAT | O_EXCL, 0644, 0);
            /* 4create one producer thread and one consumer thread */
      pthread_setconcurrency(2);
      pthread_create(&tid_produce, NULL, produce, NULL);
      pthread_create(&tid_consume, NULL, consume, NULL);
            /* 4wait for the two threads */
      pthread_join(tid_produce, NULL);
      pthread_join(tid_consume, NULL);
            /* 4remove the semaphores */
      sem_unlink(SEM_MUTEX);
      sem_unlink(SEM_NEMPTY);
      sem_unlink(SEM_NSTORED);
      exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
      int         i;
      for (i = 0; i < nitems; i++) {
            sem_wait(shared.nempty);      /* wait for at least 1 empty slot */
            sem_wait(shared.mutex);
            shared.buff[i % NBUFF] = i;   /* store i into circular buffer */
            sem_post(shared.mutex);
            sem_post(shared.nstored);     /* 1 more stored item */
      }
      return(NULL);
}
void *consume(void *arg)
{
      int         i;
      for (i = 0; i < nitems; i++) {
            sem_wait(shared.nstored);           /* wait for at least 1 stored item */
            sem_wait(shared.mutex);
            if (shared.buff[i % NBUFF] == i)
```

```
                printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
            sem_post(shared.mutex);
            sem_post(shared.nempty);            /* 1 more empty slot */
        }
        return(NULL);
}
/* end prodcons */
/*
[vishnu@mannava pxsem]$ ./a.out 5
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
*/
```

OSD 19CS2106S
Session – 39
Deadlocks
Lecture Notes

Mutex Deadlocks

Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex. When more than one thread is locking the same set of mutexes, deadlock situations can arise. Figure shows an example of a deadlock in which each thread successfully locks one mutex, and then tries to lock the mutex that the other thread has already locked. Both threads will remain blocked indefinitely.

Thread A	**Thread B**
1. pthread_mutex_lock(mutex1);	1. pthread_mutex_lock(mutex2);
2. pthread_mutex_lock(mutex2);	2. pthread_mutex_lock(mutex1);
blocks	blocks

Figure: A deadlock when two threads lock two mutexes

The simplest way to avoid such deadlocks is to define a mutex hierarchy. When threads can lock the same set of mutexes, they should always lock them in the same order. For example, in the scenario in Figure, the deadlock could be avoided if the two threads always lock the mutexes in the order mutex1 followed by mutex2. Sometimes, there is a logically obvious hierarchy of mutexes. However, even if there isn't, it may be possible to devise an arbitrary hierarchical order that all threads should follow.

Program to demonstrate deadlock.

```
/* deadlock.c */
/* On running this, nothing will be printed as both the threads are waiting to get the lock
acquired by the other process. Hence, never enter their critical sections.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
/* Create and initialize the mutex locks */
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
/* function both threads will execute in */
void* worker(void* arg) {
   if ((long long) arg == 1) { /* true for first thread  */
    /* Get the first mutex lock if it's available */
        pthread_mutex_lock(&m1);
    /* Get the second mutex lock if it's available */
        pthread_mutex_lock(&m2);
    /* Critical section start */
    printf("Inside Thread 1\n");
    /* Critical section ends */
  } else {  /* true for second thread  */
    /* Get the second mutex lock if it's available */
    pthread_mutex_lock(&m2);
    /* Get the first mutex lock if it's available */
    pthread_mutex_lock(&m1);
    /* Critical section start */
    printf("Inside Thread 2\n");
    /* Critical section ends */
  }
  /* Release the locks */
 pthread_mutex_unlock(&m1);
 pthread_mutex_unlock(&m2);
 /* Exit the thread */
 pthread_exit(0);
}
/* Note the order of waiting on the locks.
The second thread gets the second mutex lock and waits for the first mutex lock, which is
held by the first thread, which is waiting for the second mutex lock to be released. This
leads to DEADLOCK. */
int main(int argc, char *argv[]) {
 pthread_t p1, p2; /* the thread identifiers */
/* Create thread one */
 pthread_create(&p1, NULL, worker, (void *) (long long) 1);
/* Create thread two */
 pthread_create(&p2, NULL, worker, (void *) (long long) 2);
/* Wait for the two threads to complete */
 pthread_join(p1, NULL);
 pthread_join(p2, NULL);
 return 0;
}
/*
vishnu@mannava:~/threads$ cc deadlock.c -lpthread
vishnu@mannava:~/threads$ ./a.out
```

Semaphores	*Condition Variables*
Can be used anywhere in a program, but should not be used in a monitor	**Can only be used in monitors**
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	**Wait() always blocks the caller.**
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	**Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.**
If Signal() releases a blocked thread, the caller and the released thread *both* continue.	**If Signal() releases a blocked thread, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.**

Deadlocks:

There are four conditions that are necessary to achieve deadlock:

1. **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.
2. **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.
3. **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.
4. **Circular Wait** - A set of processes { P_0, P_1, P_2, . . ., P_N } must exist such that every $P_{[i]}$ is waiting for $P_{[(i+1)\%(N+1)]}$. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

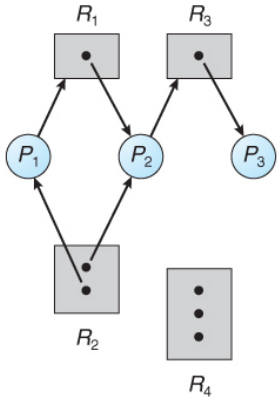
Resource-Allocation Graph

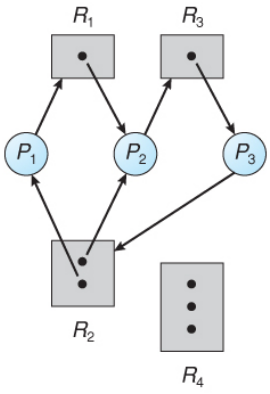
• **Request Edges -** A set of directed arcs from P_i to R_j, indicating that process P_i has requested R_j, and is currently waiting for that resource t**o** become available.

• **Assignment Edges -** A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i, and that P_i is currently holding resource R_j.

• Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.) For example:

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs.) See the example in Figure 7.2 above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- 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. Consider, for example, Figures 7.3 and 7.4 below:





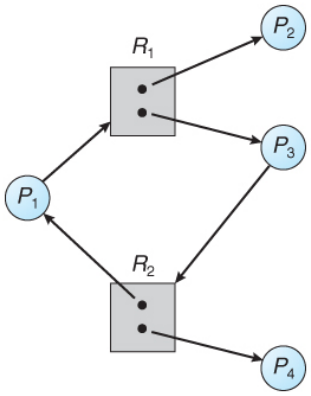


Figure 7.1 - Resource allocation graph,
Figure 7.2 - graph with a deadlock,
Figure 7.3 – graph with a cycle but no deadlock

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
  3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

Deadlock Avoidance

- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation **state** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes $\{P_0, P_1, P_2, ..., P_N\}$ such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.

Banker's Algorithm for One Resource

This is modelled on the way a small town banker might deal with customers' lines of credit. In the course of conducting business, our banker would naturally observe that customers rarely draw their credit lines to their limits. This, of course, suggests the idea of extending more credit than the amount the banker actually has in her coffers.

Suppose we start with the following situation

Customer	Credit Used	Credit Line
Andy	**0**	**6**
Barb	**0**	**5**
Marv	**0**	**4**
Sue	**0**	**7**
Funds Available	**10**	
Max Commitment		**22**

Our banker has 10 credits to lend, but a possible liability of 22. Her job is to keep enough in reserve so that ultimately each customer can be satisfied over time: That is, that each customer will be able to access his full credit line, just not all at the same time. Suppose, after a while, the bank's credit line book shows

Customer	Credit Used	Credit Line
Andy	1	6
Barb	1	5
Marv	2	4
Sue	4	7
Funds Available	2	
Max Commitment		22

Eight credits have been allocated to the various customers; two remain. The question then is: Does a way exist such that each customer can be satisfied? Can each be allowed their maximum credit line in some sequence? We presume that, once a customer has been allocated up to his limit, the banker can delay the others until that customer repays his loan, at which point the credits become available to the remaining customers. If we arrive at a state where **no customer** can get his maximum because not enough credits remain, then a deadlock *could* occur, because the first customer to ask to draw his credit to its maximum would be denied, and all would have to wait.

To determine whether such a sequence exists, the banker finds the customer closest to his limit: If the remaining credits will get him to that limit, The banker then assumes that that loan is repaid, and proceeds to the customer next closest to his limit, and so on. If all can be granted a full credit, the condition is *safe*.

In this case, Marv is closest to his limit: assume his loan is repaid. This frees up 4 credits. After Marv, Barb is closest to her limit (actually, she's tied with Sue, but it makes no difference) and 3 of the 4 freed from Marv could be used to award her maximum. Assume her loan is repaid; we have now freed 6 credits. Sue is next, and her situation is identical to Barb's, so assume her loan is repaid. We have freed enough credits (6) to grant Andy his limit; thus this state safe.

Suppose, however, that the banker proceeded to award Barb one more credit after the credit book arrived at the state immediately above:

Customer	Credit Used	Credit Line
Andy	**1**	**6**
Barb	**2**	**5**
Marv	**2**	**4**
Sue	**4**	**7**
Funds Available	**1**	
Max Commitment		**22**

Now it's easy to see that the remaining credit could do no good toward getting anyone to their maximum.

So, to recap, the banker's algorithm looks at each request as it occurs, and tests if granting it will lead to a safe state. If not, the request is delayed. To test for a safe state, the banker checks to see if enough resources will remain after granting the request to

satisfy the customer closest to his maximum. If so, that loan is assumed repaid, and the next customer checked, and so on. If all loans can be repaid, then the request leads to a safe state, and can be granted. In this case, we see that if Barb is awarded another credit, Marv, who is closest to his maximum, cannot be awarded enough credits, hence Barb's request can't be granted —it will lead to an unsafe state[3].

Banker's Algorithm for Multiple Resources

Suppose, for example, we have the following situation, where the first table represents resources assigned, and the second resources still required by five processes, A, B, C, D, and E.

Resources Assigned				
Processes	**Tapes**	**Plotters**	**Printers**	**Toasters**
A	**3**	**0**	**1**	**1**
B	**0**	**1**	**0**	**0**
C	**1**	**1**	**1**	**0**
D	**1**	**1**	**0**	**1**
E	**0**	**0**	**0**	**0**
Total Existing	**6**	**3**	**4**	**2**
Total Claimed by Processes	5	3	2	2
Remaining Unclaimed	**1**	**0**	**2**	**0**

Resources Still Needed				
Processes	**Tapes**	**Plotters**	**Printers**	**Toasters**
A	**1**	**1**	**0**	**0**
B	**0**	**1**	**1**	**2**
C	**3**	**1**	**0**	**0**
D	**0**	**0**	**1**	**0**
E	**2**	**1**	**1**	**0**

The vectors ***E, P*** and ***A*** represent Existing, Possessed and Available resources respectively:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2)
A = (1, 0, 2, 0)
```

Notice that

```
A = E - P
```

Now, to state the algorithm more formally, but in essentially the same way as the example with Andy, Barb, Marv and Sue:

1. Look for a row whose unmet needs don't exceed what's available, that is, a row where ***P*** <= ***A***; if no such row exists, we are deadlocked because no process can acquire the resources it needs to run to completion. If there's more than one such row, just pick one.
2. Assume that the process chosen in 1 acquires all the resources it needs and runs to completion, thereby releasing its resources. Mark that process as virtually terminated and add its resources to ***A***.
3. Repeat 1 and 2 until all processes are either virtually terminated (safe state), or a deadlock is detected (unsafe state).

Going thru this algorithm with the foregoing data, we see that process D's requirements are smaller than ***A***, so we virtually terminate D and add its resources back into the available pool:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2) - (1, 1, 0, 1) = (4, 2, 2, 1)
A = (1, 0, 2, 0) + (1, 1, 0, 1) = (2, 1, 2, 1)
```

Now, A's requirements are less than ***A***, so do the same thing with A:

```
P = (4, 2, 2, 1) - (3, 0, 1, 1) = (1, 2, 1, 0)
A = (2, 1, 2, 1) + (3, 0, 1, 1) = (5, 1, 3, 2)
```

At this point, we see that there are no remaining processes that can't be satisfied from available resources, so the illustrated state is *safe*.

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.

- This algorithm determines if the current state of a system is safe, according to the following steps:
  1. Let Work and Finish be vectors of length m and n respectively.
     a. Work is a working copy of the available resources, which will be modified during the analysis.
     b. Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
     c. Initialize Work to Available, and Finish to false for all elements.
  2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
  3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
  4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Need - Request

An Illustrative Example

- Consider the following situation:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))
- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

/* Dining philosophers Problem solution */

HOLD AND WAIT STRATEGY: THE LR ALGORITHM

***The LR algorithm*:** The philosophers are assigned fork acquisition strategies as follows: the philosophers whose numbers are even are R-type, and the philosophers whose numbers are odd are L-type. Concurrency in the context of the dining philosophers problem is a measure of how many philosophers can eat simultaneously (in the worst case) when all the philosophers are hungry. Having high concurrency is a most desirable property. The LR algorithm does not achieve high concurrency: it is possible to get into a situation in which only a *quarter* of the philosophers will be able to eat simultaneously when all of them are hungry.

```
/* Implementation in C  phil-lr.c:
Odd numbered philosophers pick up the right chopstick first and then the left, while even
numbered philosophers pick up the left chopstick first and then the right. */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#define NUMBER_OF_PHILOSOPHERS 5
void *philosopher(void *);
void think(int);
void pickUp(int);
void eat(int);
void putDown(int);
pthread_mutex_t chopsticks[NUMBER_OF_PHILOSOPHERS];
pthread_t philosophers[NUMBER_OF_PHILOSOPHERS];
pthread_attr_t attributes[NUMBER_OF_PHILOSOPHERS];
int main() {
	int i;
	srand(time(NULL));
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_mutex_init(&chopsticks[i], NULL);
	}
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_attr_init(&attributes[i]);
	}

	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_create(&philosophers[i], &attributes[i], philosopher, (void *)(i));
	}
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_join(philosophers[i], NULL);
	}
	return 0;
```

```
}
void *philosopher(void *philosopherNumber) {
	while (1) {
		think(philosopherNumber);
		pickUp(philosopherNumber);
		eat(philosopherNumber);
		putDown(philosopherNumber);
	}
}
void think(int philosopherNumber) {
	int sleepTime = rand() % 3 + 1;
	printf("Philosopher %d will think for %d seconds\n", philosopherNumber, sleepTime);
	sleep(sleepTime);
}
void pickUp(int philosopherNumber) {
	int right = (philosopherNumber + 1) % NUMBER_OF_PHILOSOPHERS;
	int left = (philosopherNumber + NUMBER_OF_PHILOSOPHERS) % NUMBER_OF_PHILOSOPHERS;
	if (philosopherNumber & 1) {
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, right);
		pthread_mutex_lock(&chopsticks[right]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, right);
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, left);
		pthread_mutex_lock(&chopsticks[left]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, left);
	}
	else {
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, left);
		pthread_mutex_lock(&chopsticks[left]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, left);
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, right);
		pthread_mutex_lock(&chopsticks[right]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, right);
	}
}
void eat(int philosopherNumber) {
	int eatTime = rand() % 3 + 1;
	printf("Philosopher %d will eat for %d seconds\n", philosopherNumber, eatTime);
	sleep(eatTime);
}
void putDown(int philosopherNumber) {
	printf("Philosopher %d will will put down her chopsticks\n", philosopherNumber);
	pthread_mutex_unlock(&chopsticks[(philosopherNumber + 1) % NUMBER_OF_PHILOSOPHERS]);
	pthread_mutex_unlock(&chopsticks[(philosopherNumber + NUMBER_OF_PHILOSOPHERS) %
NUMBER_OF_PHILOSOPHERS]);
}
/*
[vishnu@mannava ~]$ cc phil-lr.c -lpthread
[vishnu@mannava ~]$ ./a.out
Philosopher 1 will think for 2 seconds
Philosopher 0 will think for 3 seconds
Philosopher 2 will think for 2 seconds
Philosopher 4 will think for 2 seconds
Philosopher 3 will think for 2 seconds
Philosopher 1 is waiting to pick up chopstick 2
Philosopher 1 picked up chopstick 2
Philosopher 1 is waiting to pick up chopstick 1
Philosopher 1 picked up chopstick 1
Philosopher 1 will eat for 3 seconds
Philosopher 2 is waiting to pick up chopstick 2
Philosopher 4 is waiting to pick up chopstick 4
Philosopher 4 picked up chopstick 4
Philosopher 4 is waiting to pick up chopstick 0
Philosopher 4 picked up chopstick 0
Philosopher 4 will eat for 3 seconds
Philosopher 3 is waiting to pick up chopstick 4
Philosopher 0 is waiting to pick up chopstick 0
Philosopher 1 will will put down her chopsticks
Philosopher 1 will think for 2 seconds
Philosopher 2 picked up chopstick 2
Philosopher 2 is waiting to pick up chopstick 3
Philosopher 2 picked up chopstick 3
Philosopher 2 will eat for 3 seconds
Philosopher 4 will will put down her chopsticks
Philosopher 4 will think for 1 seconds
Philosopher 0 picked up chopstick 0
Philosopher 0 is waiting to pick up chopstick 1
Philosopher 0 picked up chopstick 1
Philosopher 3 picked up chopstick 4
Philosopher 3 is waiting to pick up chopstick 3
Philosopher 0 will eat for 1 seconds
^C
```

/* Program to demonstrate Semaphores - Reader Writer Problem */

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
sem_t mutex,writeblock;
int data = 0,rcount = 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);
}/*
[vishnu@mannava ~]$ ./a.out
Data read by the reader0 is 0
Data read by the reader1 is 0
Data read by the reader2 is 0
Data writen by the writer1 is 1
Data writen by the writer0 is 2
Data writen by the writer2 is 3 */
```

```
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);
}
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;
}
```