Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007
3rd Year CSE: 5th Semester (*Autumn 2006 - 2007*)
Lecture VI (**Process**)

Goutam Biswas *Date:* 14th - 21st August, 2005

1 Process & Thread

Process and thread are two basic units of computation.

1.1 Introduction to Process

1. What is a process?
   - A process is a program in execution.
   - A process is a virtual machine environment created by the OS and there is a stream of execution of a program on it.
   - A process is an entity to which system resources are allocated.
   - A process is identified by the process identifier (name), often a positive integer.
   - The major components of a process are: CPU state (registers etc.), data memory, stack memory, open files, signals etc.
   - One process without the help of the OS cannot affect another process.
   - More than one process may run concurrently in a multiprogramming environment.
   - More than one processes for the same program may run concurrently.
2. Why process - In a modern OS a new computation is started by creating a process. Often the first process is created by the OS kernel (**init** with PID 1 in Linux).

   A process can create a child process by requesting the OS e.g. `fork()` system call. A new program image can be loaded in the process area e.g. `execve()` system call.

   There are three main advantages of starting a computation in a child process:

   (a) The computation can speed-up if it is divided into multiple processes. When one wait for the I/O, the other may continue with computation on different data.
   But then there is communication overhead.

   (b) A parent process may need to protect itself from the computation of its child. The command interpreter e.g. bash should not crash even if a computation initiated by it does so.

   (c) A child process can be run with different priority than the parent.
3. Resource sharing among processes -

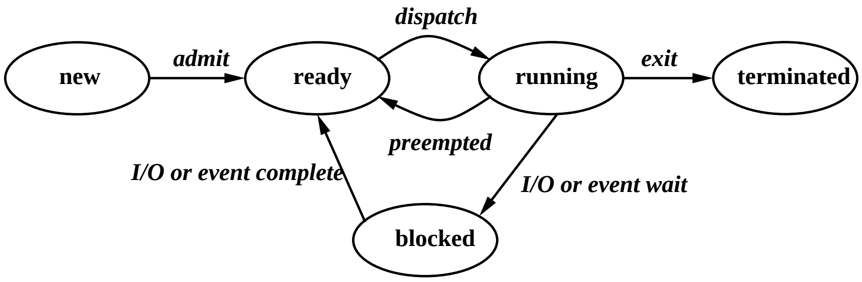
- CPU sharing - multitasking, time sharing.
- Memory sharing - more than one process images (part or whole) may reside in the memory.
- I/O sharing - different I/O devices are shared.

4. Process environment or context -

   - The process memory image i.e. code, data, stack are mapped to physical memory by the OS. This mapping is done and maintained by the OS. Protection for memory is also provided by the OS.
   - The CPU is allocated and deallocated to a process. The OS saves the state of the CPU in *process control block*.
   - A process may open a file and the OS maintains the data and metadat for the open file e.g file table, inode table, buffer cache.
   - A process may start an I/O operation that takes long time; OS suspends the process but keeps track of the I/O operation.
   - Some event might have taken place that is captured by the OS and delivered to the process.

5. States of a process -

   - A simple process state transition diagram -



5 state model

   (a) The cause of transition from the running to the ready state depends on the scheduling policy. In a time sharing system, it is often due to the timer interrupt. But if the ready queue is maintained as a priority queue, it is possible that a higher priority process has been admitted in the ready queue, and the lower priority running process is to be preempted.

   (b) The process may be blocked due to several reasons e.g there is a request from the process to the OS which cannot be serviced immediately (I/O request may need to have a disk access). The process may wait for some event in another communicating process e.g. another process is holding a semaphore.

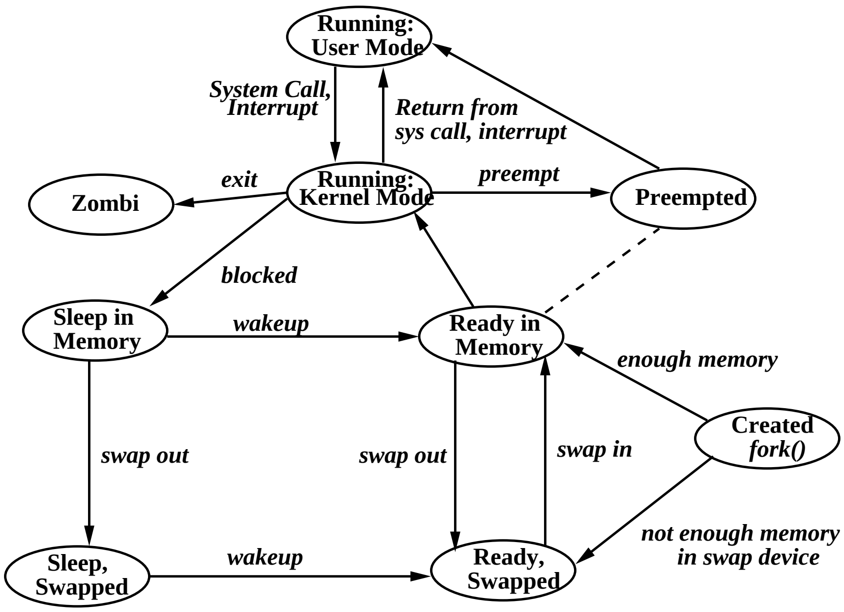
   - Swapping - The whole process or parts of it may be swapped out in the disk in the swap area when it is blocked.

     You may see the size of the swap area in Linux by the following command

`$ fdisc -l`

A process may be blocked but is in the main memory or in the swap area. In this situation it is necessary to include more states.

- Unix process state transition diagram (Maurice J Bach).



from Maurice J Bach (page 148)

(a) A new process enters the state '*created*' when the parent executes a `fork()` system call.

(b) Once it is ready to run, it may be in the memory or in the swap area.

(c) Scheduler picks up the ready to run process from memory and the process enters in the *kernel running* state where it completes the fork() system call.

(d) After completion of the system call the process moves to user mode.

(e) The timer interrupt drives the process to the kernel mode again. If the scheduler decides to schedule another process, the current process is preempted.

(f) A system call executed by a process causes a state transition to *kernel running* state.

(g) The process goes to sleep if it has to wait for some even e.g. I/O from disk.

(h) The process wakeup due to interrupt or some other event.

(i) In a modern version of Unix (or Linux), the whole process image is not swapped.

6. Context switching and process scheduling - a new process is selected from the ready queue depending on the policy. The context of the running process is saved. The context of the incoming process is is loaded and the control is transfered to the new process.

Context switching may also take place when there is transition from the user mode to the kernel mode due to interrupt, software interrupt, exception etc. In these two modes the contexts are different such as different stack, memory mapping, CPU state etc.

7. A process control block (PCB) or a process descriptor, is the main data structure (internal representation) of a process maintained by the OS. It stores complete data related to a process. The important information are:

   Process ID, parent ID, process state, user ID, priority, CPU state (ia, psw, control registers, GPRs, FPRs etc), privileged level, memory and address translation information, information about open files, pointer to the structure of the parent process, event information, signals and signal handlers etc. PCBs for processes of different states are linked.

8. A process has requested for an I/O. The I/O is initiated by the OS and the process is blocked. After completion of the I/O the OS should find out the PCB of the requesting process and remove it from the blocked state to the ready state. The event information of PCB is used for this identification.

9. Main events in a process:

   - Creation of a process
   - Termination of a process
   - I/O interrupt (including timer)
   - Process send a request to the OS (system call). These are of different kinds - I/O request, memory request, request for synchronization primitive, message send and receive, signal send and receive
   - Exception

10. Process descriptor in Linux - `struct task_struct` in (`/usr/src/linux/include/linux/sched.h` - a few important fields

    - `state` - process state - `TASK_RUNNING (0)` - running or waiting to be executed.

      `TASK_INTERRUPTIBLE (1)` - sleeping for some hardware interrupt or system resource or signal.

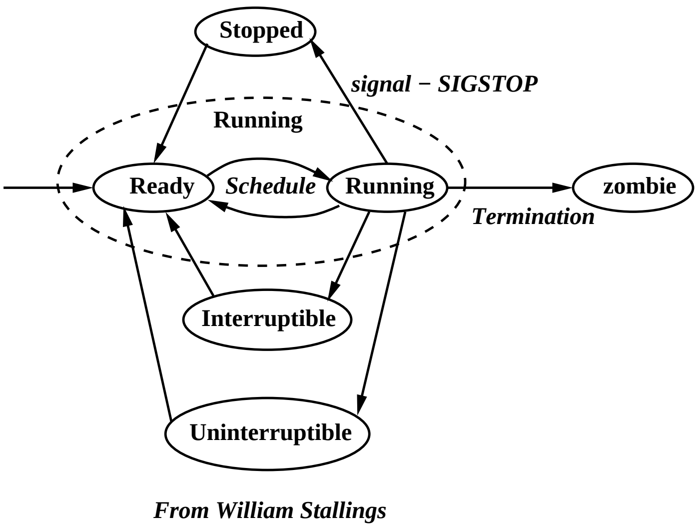
      `TASK_ZOMBIE (4)` - execution of the process is terminated, but the parent has not issued system call like `waitpid()`. The OS keeps the data structure of the process (child) with the expectation that the parent may need it.

```
/***********************************
 * What is a zombi (defunct) process
 * $ ./a.out &
 * $ ps -A
 ***********************************/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main(){
          int pid, status ;

          if((pid = fork()) != 0) {
                while(1) ;
                waitpid(pid, &status, 0) ;
          }
          else printf("\t\tGoing to die\n") ;
          return 0 ;
}
```

`TASK_STOPPED (8)` - process is stopped. Enters the state after receiving the signal SIGSTOP, SIGTSTP etc (ptrace() call).



11. Process related main operations - creation, termination, loading a new image, interprocess communication e.g. signals, shared memory, message passing, semaphore.

1.2 Introduction to Threads

1. What is a thread?

   - A thread is a stream of execution of a program within a process.
   - There may be more than one threads of computation within a process.
   - Two processes normally do not share states. But there may be two threads within a process that share part of the process state e.g. data memory, code, resources like open file.
   - A thread has its thread ID, CPU (ia/pc, psw etc) and stack. Different threads within a process have different program counters and CPU states. As the sequence of function calls may be different, stacks for different threads are also different.
   - A kernel thread is also called a lightweight process (LWP) as the thread switching is faster than process switching. Sun reports that creation of an unbounded thread (not bound to a kernel LWP) is 30 to 40 times faster than creation of a process. Creation of a bounded thread (bound to a kernel LWP) is 5 to 10 times faster than a process creation.
   - Two threads within a process share the data area of the process and can communicate through global variables. So the interthread communication is faster than interprocess communication.

     But the synchronization problem due to sharing is similar.

2. POSIX thread - the standard defines an API for creating and manipulating threads. Implementation of this standard is called the `pthread` library.

   Following are a few APIs of Pthread -

   - `int pthread_create(arg1, arg2, arg3, arg4)` creates a pthread.
     - *arg1*: `pthread_t *` is an integer pointer. The thread ID is obtained through this.
     - *arg2*: `pthread_attr_t *` - pointer to the thead attribute structure. If it is NULL, then default attributes are taken.
     - *arg3*: `void *(*)(void *)` - function name that the thread executes. The function takes a pointer and returns a pointer.
     - *arg4*: `void *` - first argument to the function specified earlier.
     - The return value is 0 (zero) on success and non-zero on error.
   - `int pthread_join(arg1, arg2)` suspends the execution of the calling thread until the specified pthread finishes.
     - *arg1*: `pthread_t` - pthread ID to finish
     - *arg2*: `void **` - if not NULL, the return value is stored in the location pointed by arg2.

```
/*********************************************
 * Programming with pthread: pthreadProg3.c
 * Compile: cc -Wall -lpthread pthreadProg3.c
 * Execute it as $ ./pthreadProg3 5
 * *******************************************/

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

void * thread1(void *) ;
void * thread2(void *) ;

int fact(int n){
     if(n == 0) return 1 ;
     return n*fact(n-1) ;
}

int fib(int n) {
     int f0 = 0, f1 = 1, i ;

     if(n == 0) return f0 ;
     if(n == 1) return f1 ;
     for(i=2; i<=n; ++i) {
          int temp = f0 ;

          f0 = f1 ;
          f1 = f0 + temp ;
     }
     return f1 ;
}


int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2; // thread ID
     int n ;

     if(count < 2) {
          printf("No argument for times\n") ;
          exit(0) ;
     }

     n = atoi(vect[1]) ;
     printf("In the main thread: &n = %p\n", &n) ;

     pthread_create(&thID1, NULL, thread1, &n) ; // 1st child thread1
```

```
    pthread_create(&thID2, NULL, thread2, &n) ; // 2nd child thread2

    pthread_join(thID2, NULL) ; // 2nd thread joins
    pthread_join(thID1, NULL) ; // 1st thread joins

    return 0 ;
}

void *thread1(void *vp) { // Address of n is passed
    int i, *p ;

    p = (int *) vp ;
    printf("\t In thread 1: &i = %p\n", &i) ;
    for(i=0; i<=*p; ++i) {
        printf("\t%d! = %d\n", i, fact(i)) ;
        sleep(1) ;
    }
    return NULL ;
}

void *thread2(void *vp) { // Address of n is passed
    int i, *p ;

    p = (int *) vp ;
    printf("\t\t\tIn thread 2: &i = %p\n", &i) ;
    for(i=0; i<=*p; ++i) {
        printf("\t\t\tfib(%d) = %d\n", i, fib(i)) ;
        sleep(2) ;
    }
    return NULL ;
}
```

Use the command `$ ps -mA`. Why are there four (4) threads[1]?

[1] According to Prof. I Sengupta, one is for the thread scheduler.

3. The following program shows that there is *race condition* in the execution of thread.

```
/**********************************************************
 * Programming with pthread: pthreadProg1.c
 * Race condition
 * Compile: $ cc -Wall -lpthread pthreadProg1.c
 *          $ ./a.out 500000
 * outpu are 0, +ve and -ve
 * *******************************************************/

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

int times, n = 0 ;
void * thread1(void *) ;
void * thread2(void *) ;
void inc() {int a; a = n; a = a+1; n = a;}
void dec() {int a; a = n; a = a-1; n = a;}

int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2;

     if(count < 2) {
          printf("No argument for times\n") ;
          exit(0) ;
     }
     times = atoi(vect[1]) ;
     pthread_create(&thID1, NULL, thread1, NULL) ;
     pthread_create(&thID2, NULL, thread2, NULL) ;
     pthread_join(thID1, NULL) ;
     pthread_join(thID2, NULL) ;

     printf("n: %d\n", n) ;
     return 0 ;
}

void *thread1(void *vp) {
     int i ;

     for(i=1; i<times; ++i) inc() ;
     return NULL ;
}

void *thread2(void *vp) {
     int i ;
```

```
    for(i=1; i<times; ++i) dec() ;
    return NULL ;
}
```

4. The race condition can be avoided by making the `inc()` and `dec()` operations **atomic** (indivisible).

```
/************************************************************
 * making inc() and dec() atomic: pthreadProg2.c
 * Compile: $ cc -Wall -lpthread pthreadProg2.c
 *          $ ./a.out 500000
 * ***********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int times, n = 0 ;
pthread_mutex_t makeAtomic = PTHREAD_MUTEX_INITIALIZER ;

void * thread1(void *) ;
void * thread2(void *) ;
void inc() {
     int a;

     pthread_mutex_lock(&makeAtomic) ;
     a = n; a = a+1; n = a;
     pthread_mutex_unlock(&makeAtomic) ;
}
void dec() {
     int a;

     pthread_mutex_lock(&makeAtomic) ;
     a = n; a = a-1; n = a;
     pthread_mutex_unlock(&makeAtomic) ;
}

int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2;

     if(count < 2) {
          printf("No argument for times\n") ;
          exit(0) ;
     }
     times = atoi(vect[1]) ;
     pthread_create(&thID1, NULL, thread1, NULL) ;
     pthread_create(&thID2, NULL, thread2, NULL) ;
     pthread_join(thID1, NULL) ;
```

```
    pthread_join(thID2, NULL) ;

    printf("n: %d\n", n) ;
    return 0 ;
}

void *thread1(void *vp) {
    int i ;

    for(i=1; i<times; ++i) inc() ;
    return NULL ;
}

void *thread2(void *vp) {
    int i ;

    for(i=1; i<times; ++i) dec() ;
    return NULL ;
}
```

- The variable `makeAtomic` of type `pthread_mutex_t` is used to make the operations (`inc()` and `dec()`) *atomic*. The variable is initialized by the constant `PTHREAD_MUTEX_INITIALIZER`.

  The mutual exclusion variable can be initialized to different types: fast, recursive and error checking. We consider simplest one, fast.
- The function `pthread_mutex_lock()` is used to lock the mutual exclusion variable `makeAtomic` and the function `pthread_mutex_unlock()` is used to unlock it.
- A mutual exclusion variable can be locked (owned) by only one thread. No other thread can own it when it is already locked. The thread is suspended if it tries to lock a mutual exclusion variable that is already locked by another thread.

5. Use of static and global may make a code *thread unsafe*:

```
/*********************************************
 * Thread unsafe code: static int in fib()
 * Compile: cc -Wall -lpthread pthreadProg5.c
 * Execute it as $ ./a.out 5
 * ******************************************/

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

void * thread1(void *) ;
void * thread2(void *) ;
```

```
int fib(int n) {
     static int f0 = 0, f1 = 1 ;
     int temp ;

     if(n == 0) return f0 ;
     if(n == 1) {
              temp = f1 ;
              f0 = 0 ;
              f1 = 1 ;
              return temp ;
     }
     temp = f0 ;
     f0 = f1 ;
     sleep(1) ; // Delay
     f1 = f0 + temp ;
     return fib(n-1) ;
}

int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2; // thread ID
     int n ;

     if(count < 2) {
          printf("No argument for times\n") ;
          exit(0) ;
     }

     n = atoi(vect[1]) ;
     pthread_create(&thID1, NULL, thread1, &n) ; // 1st child thread1
     pthread_create(&thID2, NULL, thread2, &n) ; // 2nd child thread2

     pthread_join(thID2, NULL) ; // 2nd thread joins
     pthread_join(thID1, NULL) ; // 1st thread joins

     return 0 ;
}

void *thread1(void *vp) { // Address of n is passed
     int i, *p ;

     p = (int *) vp ;
     for(i=0; i<=*p; ++i) {
          printf("\tfib(%d) = %d\n", i, fib(i)) ;
     }
     return NULL ;
}
```

```
void *thread2(void *vp) { // Address of n is passed
     int i, *p ;

     p = (int *) vp ;
     for(i=0; i<=*p; ++i) {
          printf("\t\t\tfib(%d) = %d\n", i, fib(i)) ;
     }
     return NULL ;
}
```

Rewrite the function `fib()` without using `static` variable.

```
/*********************************************
 * Thread safe code: static int in fib()
 * Compile: cc -Wall -lpthread pthreadProg5.c
 * Execute it as $ ./pthreadProg3 5
 * ******************************************/

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

void * thread1(void *) ;
void * thread2(void *) ;

int fib(int n) {
     int f0 = 0, f1 = 1, i ;

     if(n == 0) return f0 ;
     if(n == 1) return f1 ;
     for(i=2; i<=n; ++i) {
          int temp = f0 ;

          f0 = f1 ;
          sleep(1) ; // Delay
          f1 = f0 + temp ;
     }
     return f1 ;
}

int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2; // thread ID
     int n ;

     if(count < 2) {
```

```
        printf("No argument for times\n") ;
        exit(0) ;
    }

    n = atoi(vect[1]) ;
    pthread_create(&thID1, NULL, thread1, &n) ; // 1st child thread1
    pthread_create(&thID2, NULL, thread2, &n) ; // 2nd child thread2

    pthread_join(thID2, NULL) ; // 2nd thread joins
    pthread_join(thID1, NULL) ; // 1st thread joins

    return 0 ;
}

void *thread1(void *vp) { // Address of n is passed
    int i, *p ;

    p = (int *) vp ;
    for(i=0; i<=*p; ++i) {
        printf("\tfib(%d) = %d\n", i, fib(i)) ;
    }
    return NULL ;
}

void *thread2(void *vp) { // Address of n is passed
    int i, *p ;

    p = (int *) vp ;
    for(i=0; i<=*p; ++i) {
        printf("\t\t\tfib(%d) = %d\n", i, fib(i)) ;
    }
    return NULL ;
}
```

6. If thread calls a `fork()`, by POSIX standard only that thread is duplicated in the new process.

```
/**********************************************
 * fork() in a thread?
 * Programming with pthread: pthreadProg7.c
 * Compile: cc -Wall -lpthread pthreadProg3.c
 * Execute it as $ ./a.out
 * *******************************************/

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

```
#include <pthread.h>

void * thread1(void *) ;
void * thread2(void *) ;

int fact(int n){
     if(n == 0) return 1 ;
     return n*fact(n-1) ;
}

int fib(int n) {
     int f0 = 0, f1 = 1, i ;

     if(n == 0) return f0 ;
     if(n == 1) return f1 ;
     for(i=2; i<=n; ++i) {
          int temp = f0 ;

          f0 = f1 ;
          f1 = f0 + temp ;
     }
     return f1 ;
}


int main(int count, char *vect[]) { // argument is data for times
     pthread_t thID1, thID2; // thread ID
     int n ;

     if(count < 2) {
          printf("No argument for times\n") ;
          exit(0) ;
     }

     n = atoi(vect[1]) ;
     printf("In the main thread: &n = %p\n", &n) ;

     pthread_create(&thID1, NULL, thread1, &n) ; // 1st child thread1
     pthread_create(&thID2, NULL, thread2, &n) ; // 2nd child thread2

     pthread_join(thID2, NULL) ; // 2nd thread joins
     pthread_join(thID1, NULL) ; // 1st thread joins

     return 0 ;
}

void *thread1(void *vp) { // Address of n is passed
```

```
     int i, *p, chPID ;

     p = (int *) vp ;
     if((chPID = fork()) != 0){ // Parent ;
       printf("In thread 1:parent: &i = %p\n", &i) ;
       for(i=0; i<=*p; ++i) {
          printf("%d! = %d\n", i, fact(i)) ;
          sleep(1) ;
       }
     }
     else { // Child
       printf("\t\tIn thread 1:child: &i = %p\n", &i) ;
       for(i=0; i<=*p; ++i) {
          printf("\t\t%d! = %d\n", i, fact(i)) ;
          sleep(1) ;
       }
     }
     while(1) ;
     return NULL ;
}

void *thread2(void *vp) { // Address of n is passed
     int i, *p ;

     p = (int *) vp ;
     printf("\t\t\t\tIn thread 2: &i = %p\n", &i) ;
     for(i=0; i<=*p; ++i) {
          printf("\t\t\t\tfib(%d) = %d\n", i, fib(i)) ;
          sleep(2) ;
     }
     while(1) ;
     return NULL ;
}
```

7. `clone()` - Linux uses this system call to implement kernel level thread. In fact the call can be used to implement process as well.

   It is possible to create kernel threads that shares the VM of a process and runs concurrently.

   ```
   int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
   ```

   The clone system call starts executing the function `fn(arg)`, where the function name (pointer) is the first parameter and the argument `arg` is the fourth parameter. The return value of the function is the exit code for the child thread.

   The child process shares the same virtual memory of the parent, so they cannot use the stack of the parent and the parent have to supply the *bottom* address of the stack which normally grows towards the lower address of the memory.

   ```
   /*************************************
    *  Creation of new thread by clone()
    *  clone.c
    *************************************/

   #include <stdio.h>
   #include <sched.h>
   #include <sys/types.h>
   #include <unistd.h>
   #include <stdlib.h>
   #include <sys/types.h>
   #include <sys/wait.h>

   #define MAXSTACK 4096

   int n ; // global data
   int what(void *p) ;

   int main() {
      int chPID, status ;
      char *chStack ;

      chStack = (char *) malloc(MAXSTACK) ;  // New stack
      chStack = chStack  + MAXSTACK ;        // Stack grows towards lower
                                             // address (not in HP PA)

      chPID = clone(what, chStack, CLONE_VM, 0) ; // Cloned process will
                                             // execute 'what'
                // chStack - base of new stack
                // CLONE_VM - same virtual memory
                // 0 - No parameter to 'what'
                // chPID - cloned process id
   ```

```
    n = 25 ;
    printf("\tInside process: pid = %d\n", getpid()) ;
    printf("\tParent = %d\n", getppid()) ;
    printf("\tChild = %d\n", chPID) ;
    waitpid(chPID, &status, __WCLONE) ;  // __WCLONE - wait for
    printf("\tn = %d\n", n) ;            // cloned process
    return 0 ;
}

int what(void *p) {
     printf("\t\t\tInside clone: pid = %d\n", getpid()) ;
     printf("\t\t\tParent pid = %d\n", getppid()) ;
     n = 100 ;

     return 0 ;
}
```

8. There are different implementations of thread.

   - *Kernel level thread*: The thread is created by the
   - User thread can be managed above the kernel by the thread library. The library provides support for creation, scheduling and termination of the threads in the user space. It is faster.

     An application starts as a process with a single thread. New threads are created by calling thread creation procedure to the thread library. The library maintains data structure to manage threads within the process. But the kernel treats the process as a single execution unit.

     Problem of blocking all threads when one thread gives a blocking system call. Kernel should provide non-blocking system calls
   - Kernel thread are managed by the kernel.
   - many-to-one, one-to-one and many-to-many models.

1.3 Communicating Processes

Two or more processes may interact to do a certain job. There are different ways of interaction. Two processes are said to be *interacting processes* or *communicating processes* if the intersection of the read set of one process and the write set of the other process is not null. The word read set and write set are to be understood in a general set up - receiving a message, reading some data from a shared memory location, catching a signal etc. are all 'read' operations. Similarly sending a message, writing data in a shared location, locking some resource are write operations.

Consider a shared location where two processes P_1 and P_2 may update the data. Let the initial data be d, and the update operations are f_1 and f_2 respectively. If the update operation is not properly serialized, the data after update may be inconsistent in the sense that its value will be other than $f_1(f_2(d))$ or $f_2(f_1(d))$. This is called *race condition*.

Race can be prevented by making the data access mutually exclusive or non-concurrent i.e. by making the operations f_1 and f_2 *atomic*. The operations can be made atomic by making the code corresponding to these operations atomic. In other word the execution of the code for f_1 in process P_1 and the code for f_2 in process P_2 should be mutually exclusive i.e. at no point the thread of controls should point to both the codes. These two portions of codes are called *critical sections*.

Synchronization of control between two processes P_1 and P_2 means that the thread of execution in one process (say P_2) will not cross a certain point (instruction) before the completion of execution upto a certain point (instruction) of the other process (P_1 in this case).