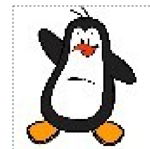


Lecture # 24-25
Synchronization between Threads and Processes

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

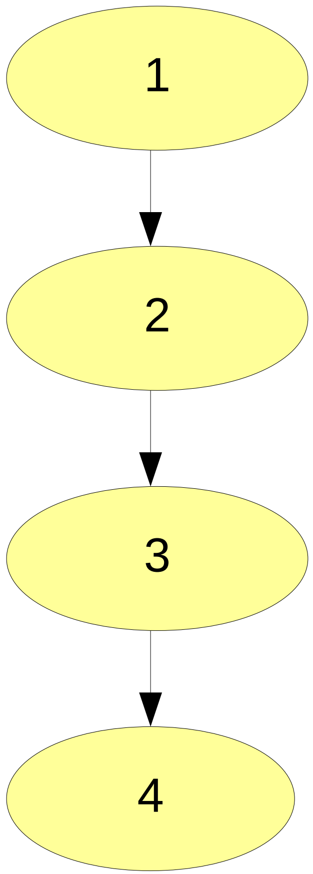
Today's Agenda

- Concurrent Programming using Threads/Processes
- Creating threads using Pthread API
- Thread Attributes
- Data Sharing among threads
- Critical Section Problem and its solution using **`pthread_mutex_t`**
- POSIX Semaphores
  - Unnamed semaphores between threads and processes
  - Named semaphores between threads and processes
- Condition Variables
- Thread Cancellation

CONCURRENT / PARALLEL PROGRAMMING

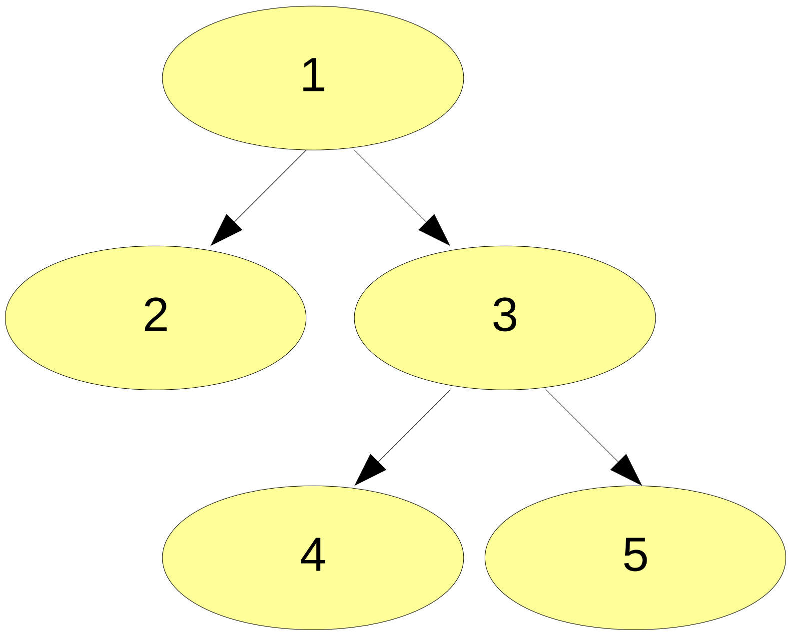
Sequential Programming

- Sequential programming is executed line by line. All computational task are executed in a sequence, one after an other.



Concurrent Programming

- Multiple computational tasks are executed simultaneously in case of multiple CPUs, OR concurrently in case of single CPU.



Concurrent Programming (cont...)

Concurrency Examples

- Web servers listen and accept a request and listen again.
- A file server listen for a file request, accept and till I/O is done listen for the other request.

Ways to achieve concurrency :

- Multiple single threaded processes
- Multiple threads within a single process
- Single process multiple events

Concurrent Programming (cont...)

- **Multiple single threaded processes**
  - Use `fork()` to create a new process for handling every new task, The child process serves the client process, while the parent listens to the new request
  - Possible only if each slave can operate in isolation
  - Need IPC between processes
  - Lot of memory and time required for process creation

Concurrent Programming (cont...)

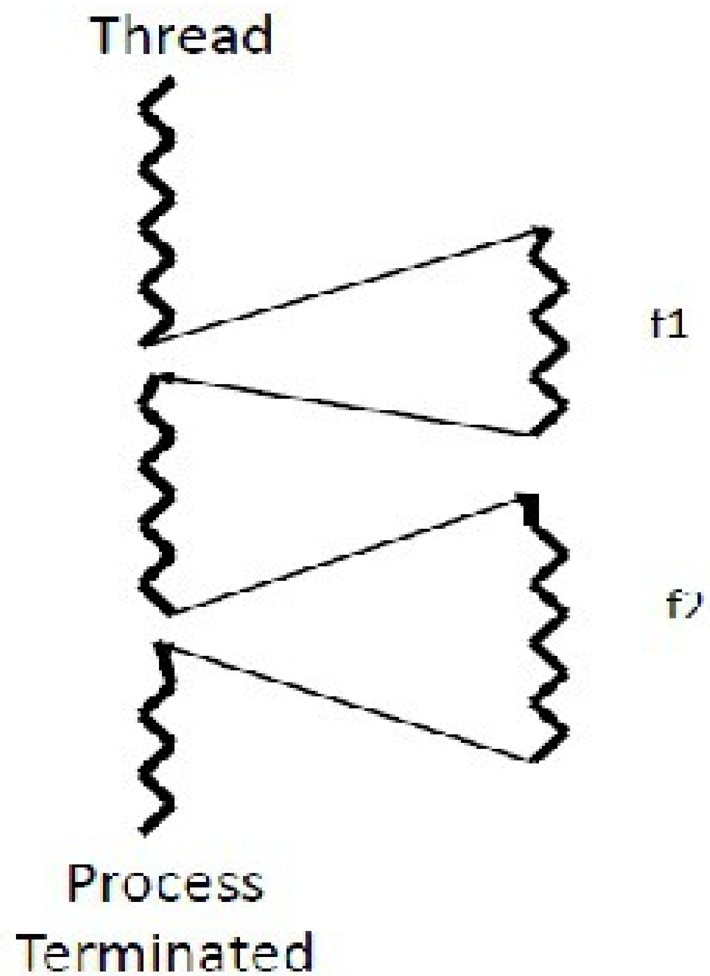
- **Multiple threads within a single process**
  - Use pthreads to create threads within a single process
  - Good if each slave need to shared data
  - Cost of creating threads is low, and no IPC required
- **Single process multiple events**
  - Use `select()` and `poll()` for asynchronous I/O. (event driven model)
  - Use non-blocking I/O, process repeatedly poll for I/O on any of the connections it has opened and handles each request

Processes and Threads

- Processes have two characteristics:
  - **Resource ownership -** process includes a virtual address space to hold the process image
  - **Scheduling/execution -** follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system.
- The **unit of resource ownership** is referred to as a **process** or task
- The **unit of dispatching** is referred to as a **thread** or lightweight process

Single Threaded Process

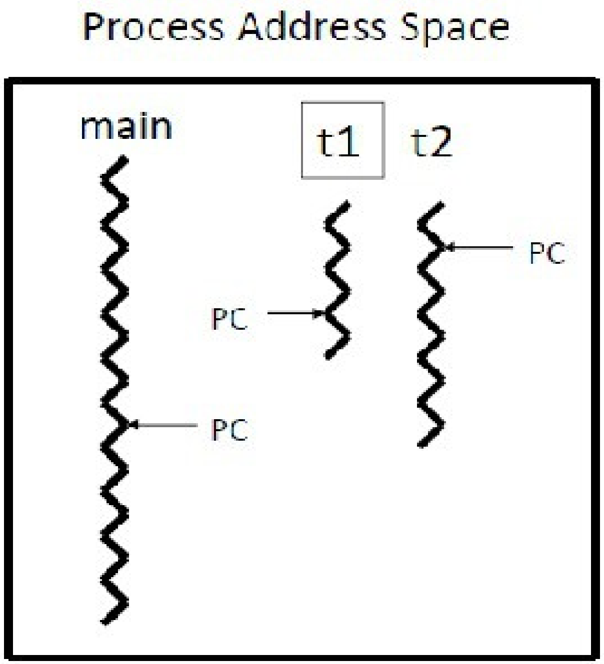
```
main()
{
        ...
        f1(...);
        ...
        f2(...);
        ...
}

f1(...)
{  ...  }

f2(...)
{  ...  }
```



Multi-Threaded Process

A thread is an execution context that is independently scheduled, but shares a single addresses space with other threads of the same process

```
main()
{
      ...
      thread(t1,f1);
      ...
      thread(t2,f2);
      ...
}

f1(...)
{ ... }

f2(...)
{ ... }
```



Temporal Multi-threading: Only one thread of instruction can execute in any given pipeline stage at a time.
Simultaneous Multi-threading (SMT/HT): More than one thread of instruction can execute in any given pipeline stage at a time. (SMT/HT is a multi-threading on a super scalar architecture.)

Example of a Multithreaded Process

Suppose we want to add eight numbers $x_1 + x_2 + x_3 \; .... + x_8$

- In case of sequential programming there are seven addition operations and if each operation take 1 CPU cycle,the entire operation will take seven cycles.
- Suppose we have 4xCPUs or a 4xCore CPU. Now we can divide the task and compute quickly:

$x_1 + x_2 \;\; + \;\; x_3 + x_4 \;\; + \;\; x_5 + x_6 \;\; + \;\; x_7 + x_8$

CPU1 + CPU2 + CPU3 + CPU4 → 1st CPU cycle

CPU 1 + CPU2 → 2nd CPU cycle

CPU1 → 3rd CPU cycle

Multi-Threaded Process

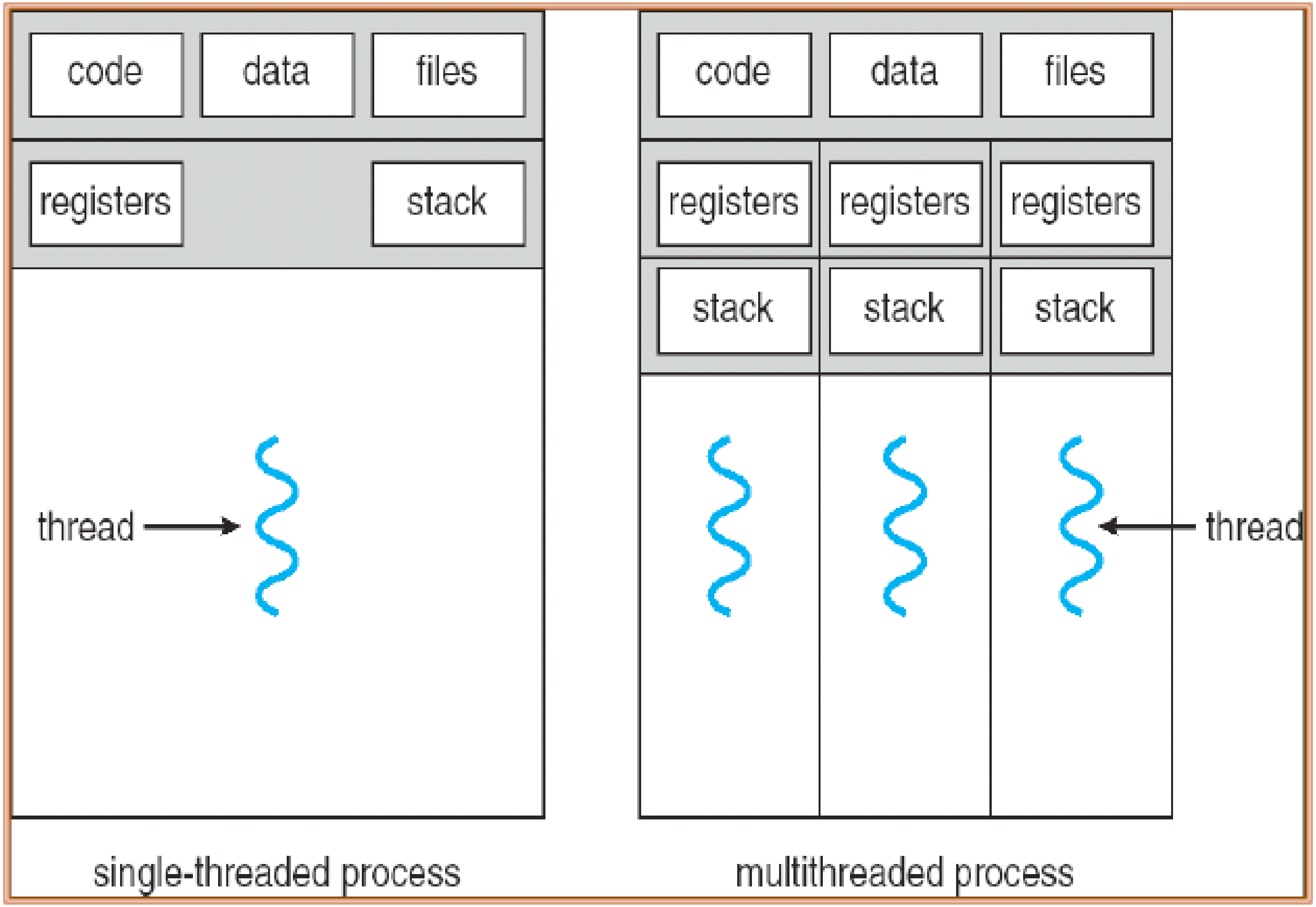
Threads within a process share :

- PID, PPID, PGID, SID, UID, GID
- Code and Data Section
- Global Variables
- errno variable
- Open files via PPFDT
- Signal Handlers
- Interval Timers
- CPU time consumed
- Resources Consumed
- Nice value

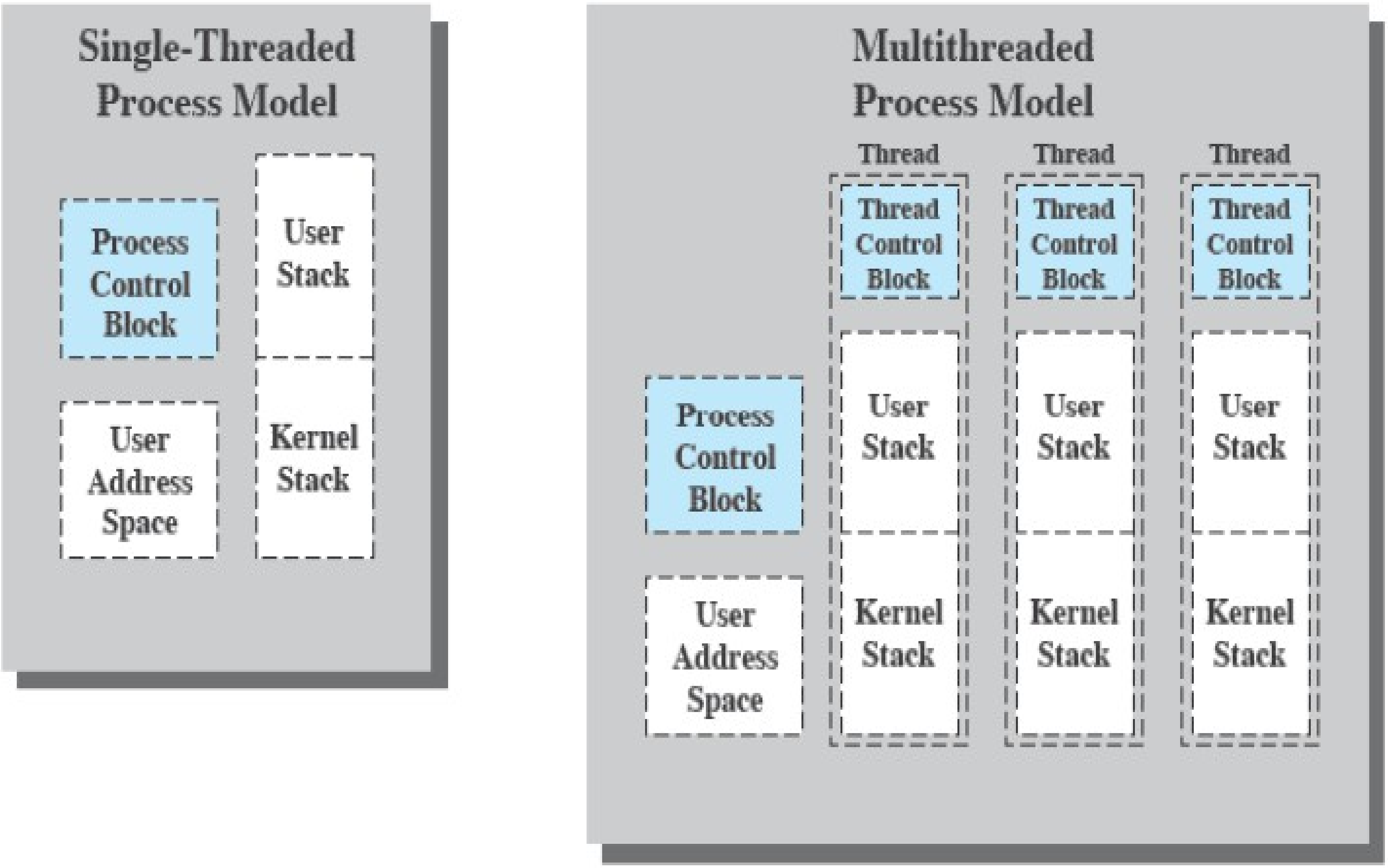
Threads have their own:

- Thread ID
- CPU Context (PC, and other registers)
- Stack
- State
- Priority
- Signal mask

Single VS Multi Threaded Processes

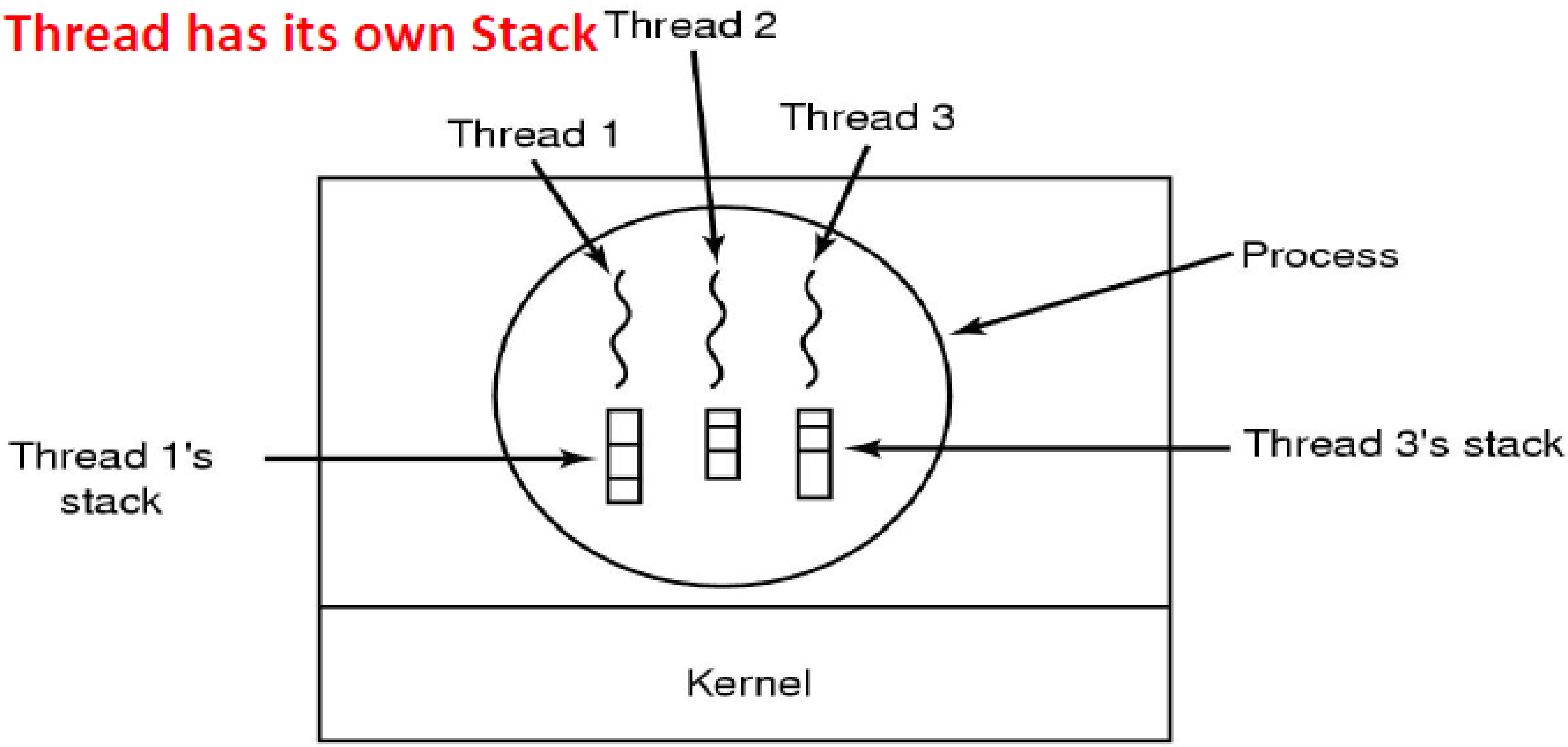


Single and Multi Threaded Processes



Single and Multi Threaded Processes

Each Thread has its own Stack



- Each thread stack contains one frame for each procedure that has been called but not yet returned from
- This frame contains the procedure's local variables and their return address to use when the procedure call has finished
- For example, if procedure X calls procedure Y and this one calls procedure Z, while Z is executing the frames for X, Y, Z will be on the stack
- Each thread will generally call different procedures and thus has a different execution history

Threads and Processes

Similarities between threads and processes:

- Like a process, a thread can also be in one of many states (new, ready, running, block, terminated)
- Only one thread can be in running state (single CPU)
- Like a process a thread can create a child thread.

Differences between threads and processes:

- No automatic protection in threads.
- Every process has its own address space, while all other threads within a process executes within the same address space.

Classification

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Win NT to XP, Solaris, HP-UX, OS X

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space

THREAD TYPES

User Level Threads

- A thread context consists of Stack Pointer, Program Counter and a set of CPU registers. On a thread switch, context of the currently running thread is saved. Thread Scheduler selects a new thread from Ready Queue. Dispatcher dispatches and restores the context of the newly selected thread from that thread's private stack. Use PC of new thread and start executing instructions.
- Above tasks can be performed by a user program written in assembly language and are called user level threads. If this code resides in OS kernel then these are called kernel level threads.
- An application can be programmed to be multi threaded by using user level thread libraries (e.g., Pthread, Win32 threads, Java threads, Solaris2 threads, Mach C Threads) and Kernel is not aware of the existence of threads.
- These libraries contain code for:
    - ➢ Thread creation and termination
    - ➢ Thread scheduling
    - ➢ Saving and restoring thread context
    - ➢ Passing messages and data between threads
    - By default an application begins with a single thread.

User level Threads (cont...)

Advantages:

- Thread switching is fast as it is done by ULT library, thus saving the overhead of two mode switches.
- Scheduling can be application specific instead of OS specific
- ULT can be used even if the underlying platform doesn't support multi-threading

Disadvantages:

- When a ULT make a blocking system call, kernel take it as if the system call has been made by the process, so all threads of that process are blocked.
- In pure ULT strategy, a multi threaded application cannot take the advantage of multiple processors/cores.

Kernel Level Threads

- Thread management is done by kernel and Kernel is aware of threads. Kernel level threads are supported in almost all modern operating systems e.g (Windows-7, Linux, Solaris,Tru64 UNIX, Mac OS X).

- **Advantages:**
  - When a KLT makes a blocking system call, only that thread within the process is blocked.
  - Can take the advantage of multiple processor/cores
- **Disadvantage:**
  - Thread switching is slow as kernel is involved.
  - An application uses KLT can't execute on non-multi-threaded Operating System.

THREAD IMPLEMENTATION MODELS

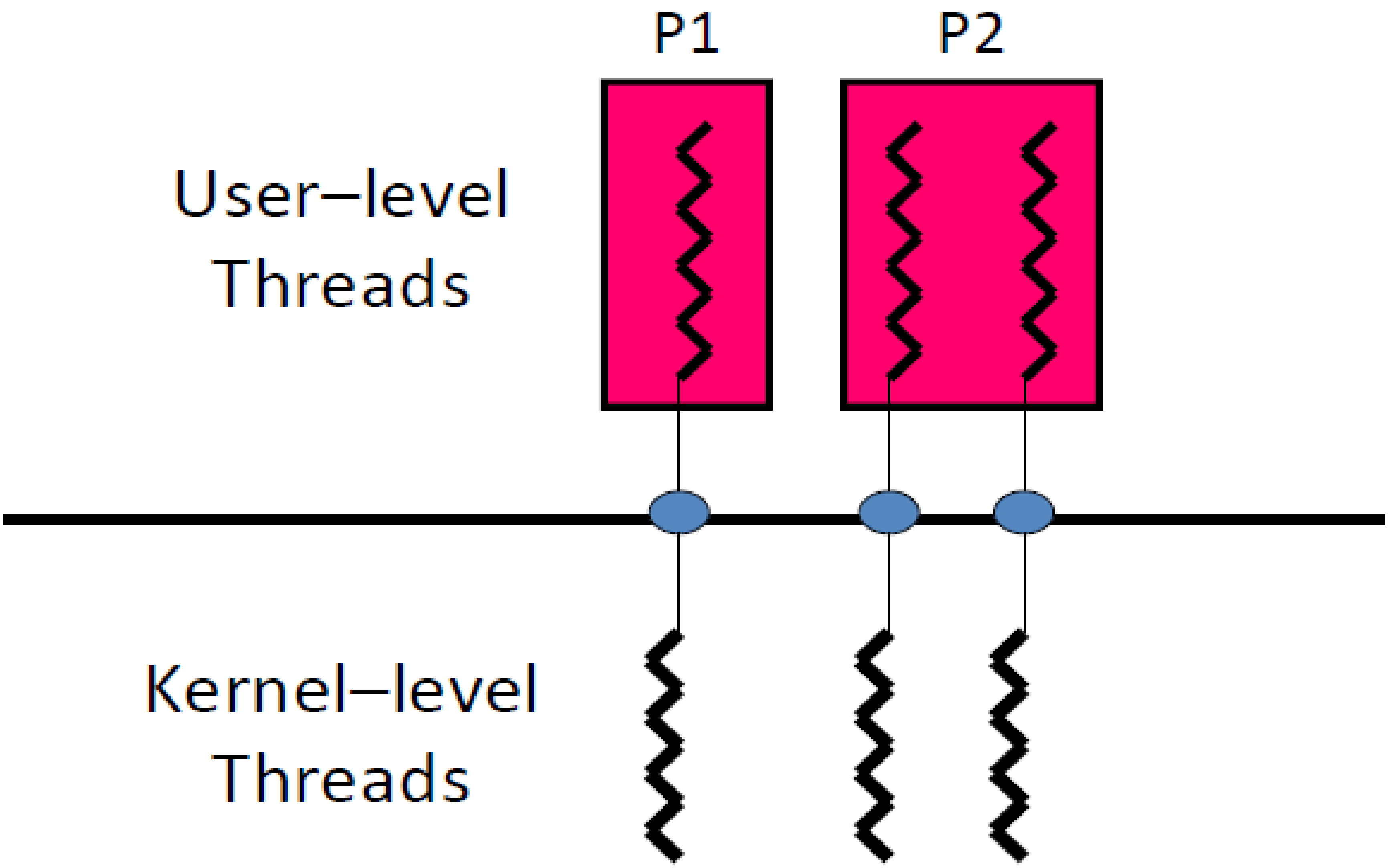
One-to-One (1:1) Model (Kernel lvl Threads)

There must exist a relationship between user thread and kernel thread. The three common implementation models are:

ONE-to-ONE (1:1)

- GNU Linux threads follow this model; each thread is actually a separate process in the kernel.
- Kernel schedules the threads just like it schedule processes.
- Threads are created with Linux **`clone()`** system call, which is a generalization of **`fork()`** allowing new process to share the memory space, file descriptors and signal handlers of the parent.

One-to-One Model (Kernel lvl Threads)



Many-to-One (M:1)Model (User lvl Threads)

- Many user threads per kernel thread, i.e., kernel sees just one thread.
- All thread management is done by user level thread libraries.

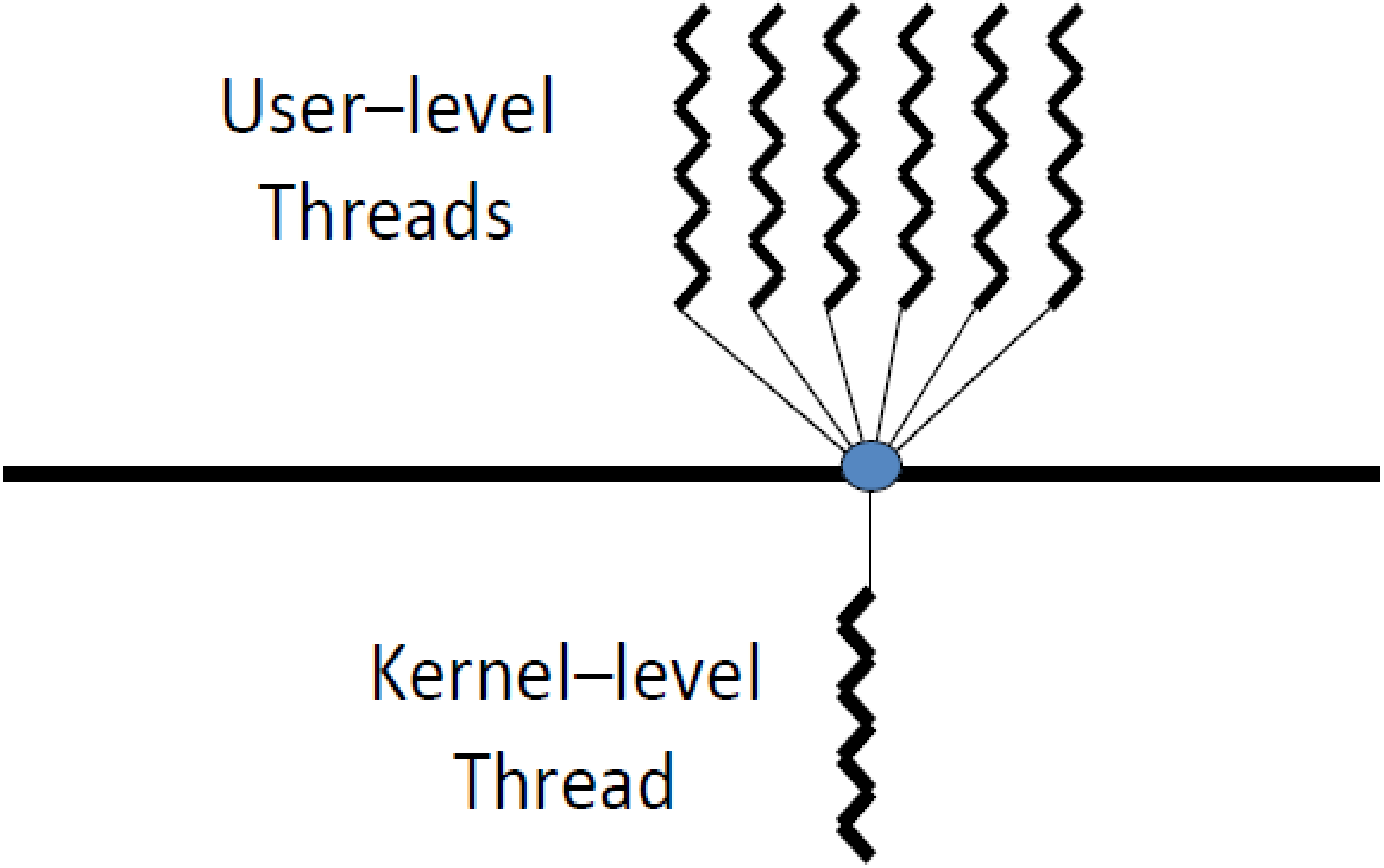
Advantages

- Thread management is done in user space, so it is efficient.

Disadvantages

- When a thread within a process makes a system call the entire process blocks.
- No advantage of multiple CPUs/cores.
- Most of the libraries are deficient in functionality, performance and robustness

Many-to-One Model (User lvl Threads)



Many-to-Many (M:N) Model

- Multiple user threads are multiplexed over a smaller or equal number of kernel threads. This model is a compromise between 1:1 and M:1
- User threads in the M:N model normally float among kernel threads – that may run on whatever kernel thread is available when they become runnable
- Pretty complex to implement, and requires kernel support which Linux does not provide at the time of this writing.

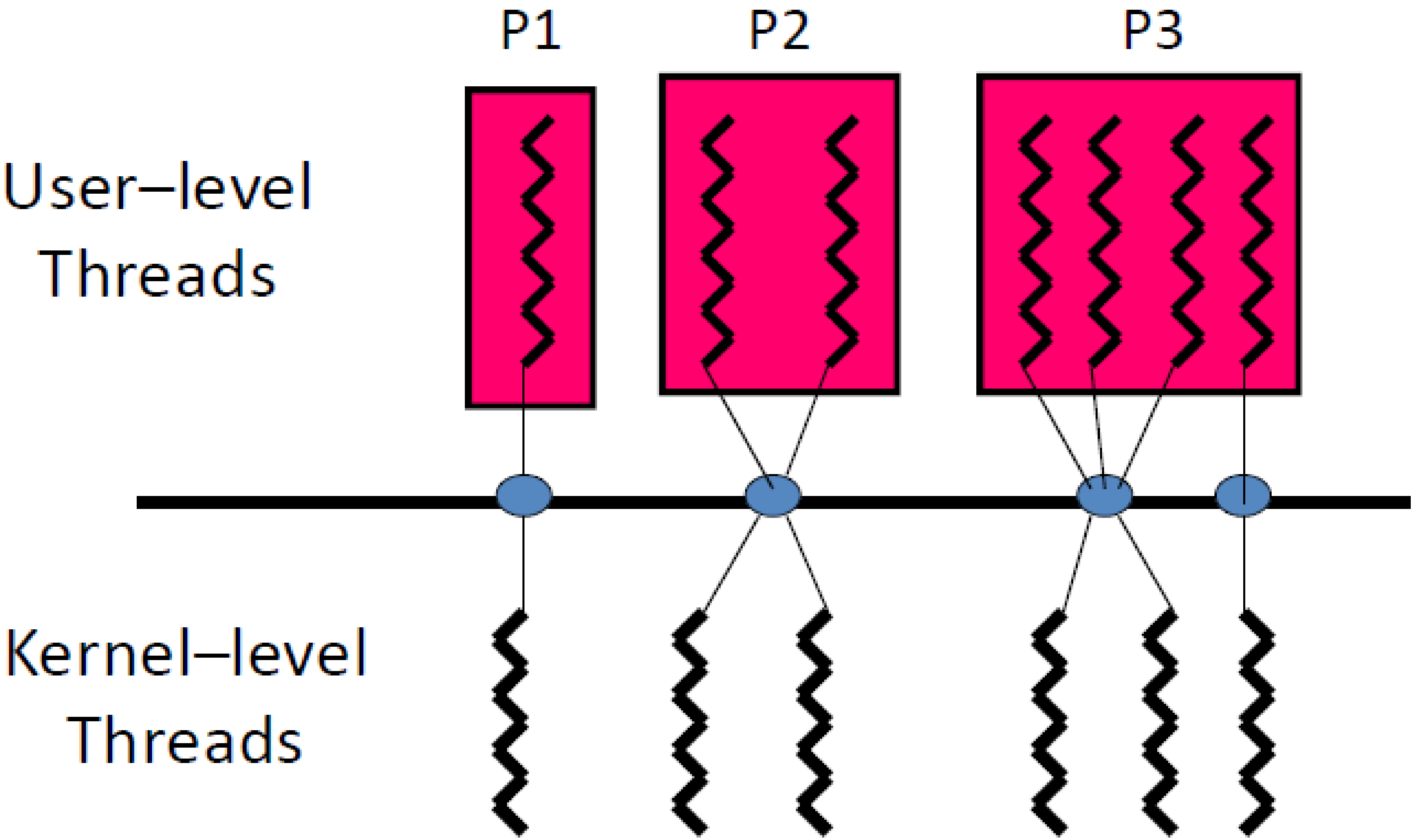
Advantages:

- Design permits the kernel to distribute the threads of an application across multiple CPUs, while eliminating the possible scaling problems associated with applications that employ large number of threads.
- If one thread makes a blocking system call, kernel can schedule another.

Disadvantage:

- Task of thread scheduling is shared between the kernel and user-space threading library, which must cooperate and communicate info with one another.
- Managing signals according to the requirements of SUSv3 is also complex

Many-to-Many Model

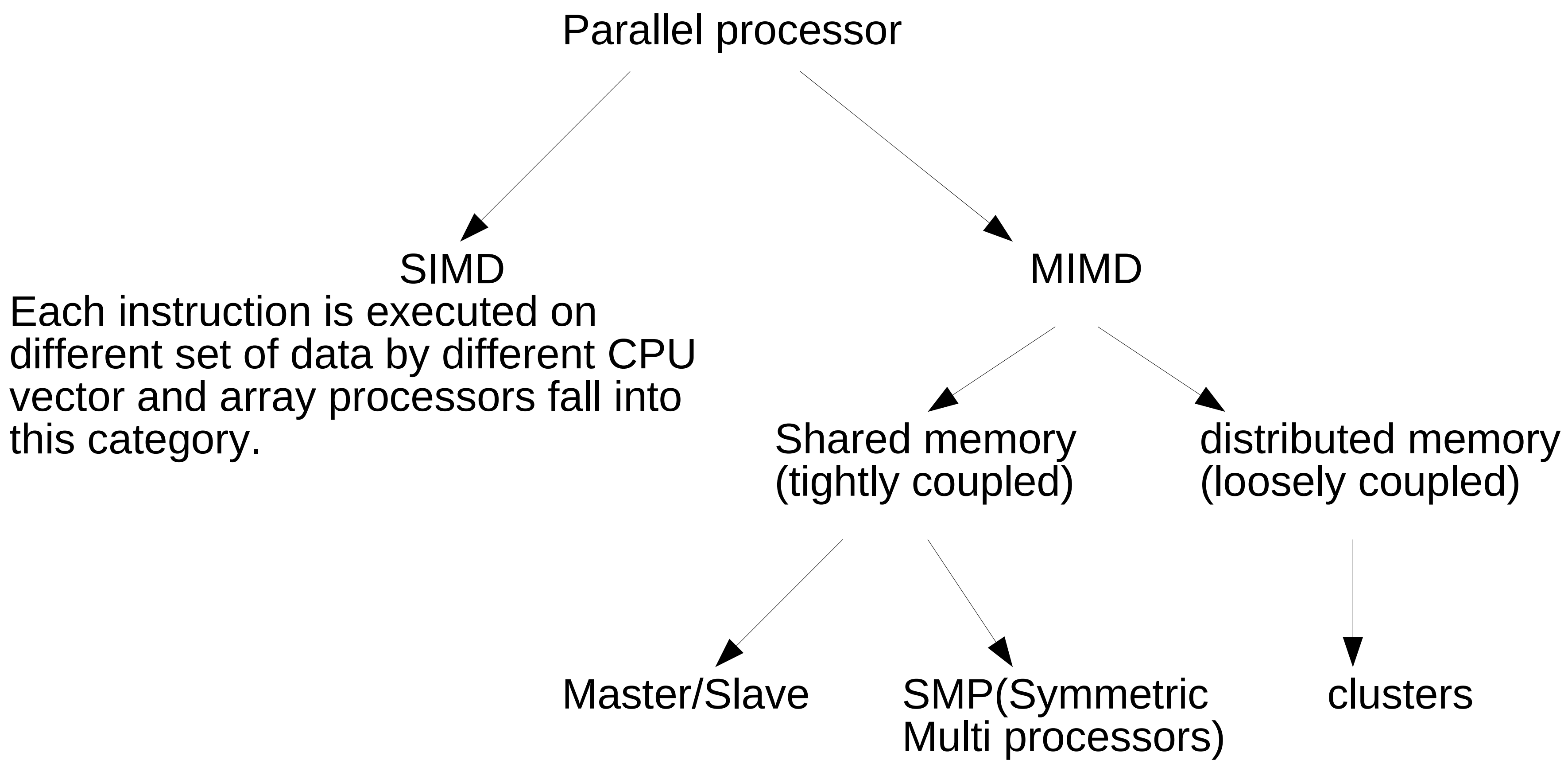


Relationship Between Threads and Process

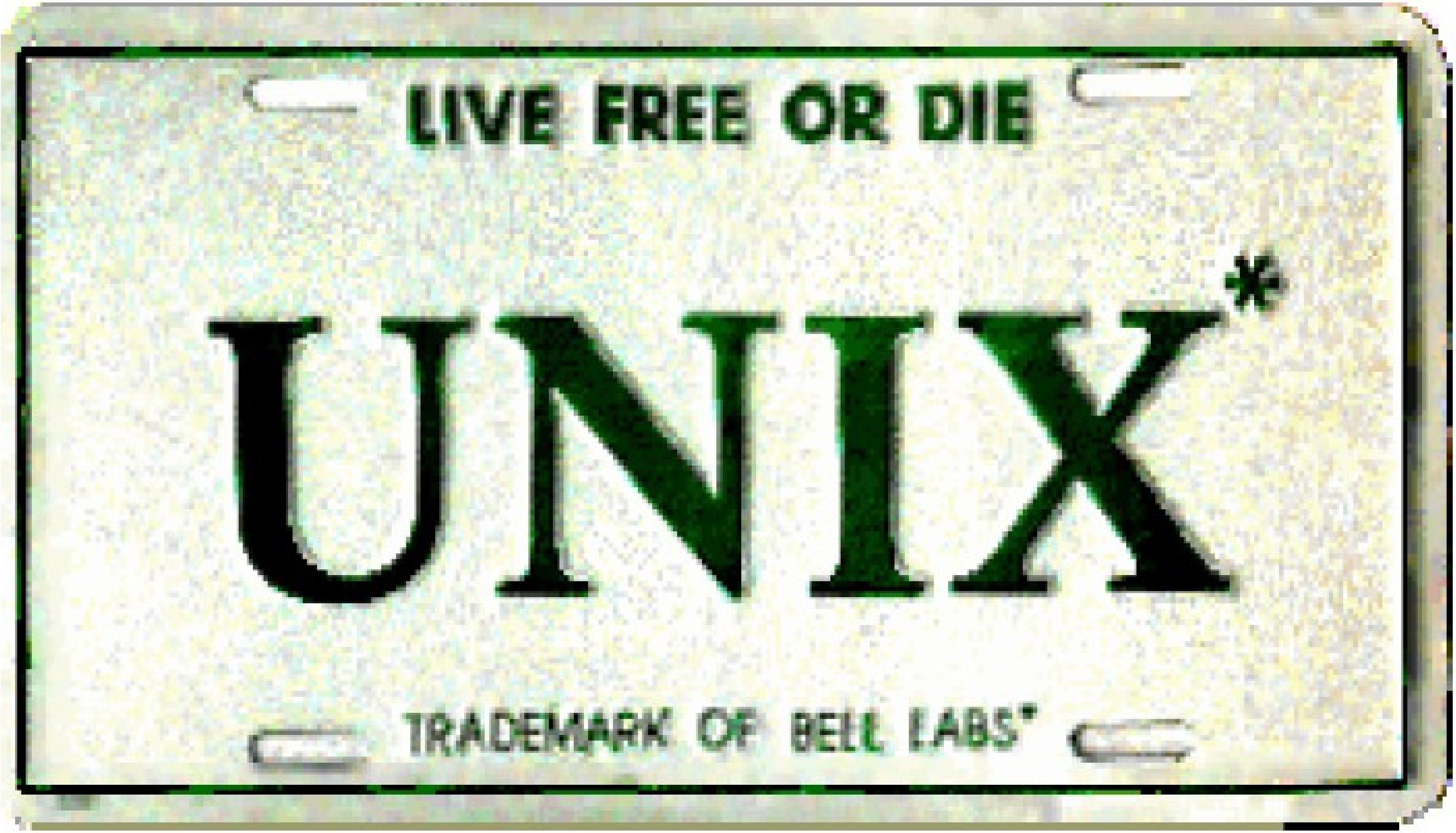
Threads:Processes	Description	Example System
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional Unix implementation.
M:1	Multiple threads may be created and executed within a process.	Windows 7, Linux, Solaris, OS/2, OS/390, MACH OS x
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra(Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases	TRIX

Process/Thread Migration. The movement of processes or threads among address spaces on different machines has become a hot topic in recent years.

Parallel Architecture



Distributed Memory: MPICH2
Shared Memory: OPENMP, openCL


LIVE FREE OR DIE
UNIX*
TRADEMARK OF BELL LABS*

Linux Implementation of Pthreads

Linux has two main implementations of the Pthreads API

- **LinuxThreads:**
  - This is the original Linux threading implementation, developed by Xavier leroy.
  - Threads are created using a **`clone()`**, using which threads share virtual memory, file descriptors, file system-related information (umask, root directory, pwd,...) and signal disposition. However, threads don't share PIDs and PPIDs.
  - In addition to the threads created by the application, LinuxThreads creates an additional “manager” thread that handles thread creation and termination.
    - **Deviations from specified behaviour**
    - **`getpid()`** returns a different value in each of the threads of a process.
    - **`getppid()`** returns the PID of the manager threads
    - If one thread creates a child using **`fork()`**, then only the thread that created the child process can **`wait()`** for it.
    - If a thread calls **`exec()`**, then SUSv3 requires that all other threads are terminated. While this is not so in LinuxThreads.
    - Threads don't share PGIDs, and SIDs
    - Threads don't share resource limits.
    - Some versions of `ps`(1) show all of the threads in a process (including the manager thread) as separate items with distinct PIDs
    - CPU time returned by **`times()`** and resource usage information returned by `getrusage()` are per thread.
    - Threads don't share nice value set by **`setpriority()`**.
    - Interval timers created using **`setitimer()`** are not shared between the threads.

Linux Implementation of Pthreads

- **NPTL (Native POSIX Threads Library):**
  - This is the modern Linux threading implementation, developed by Ulrich Drepper and Molnar as a successor to LinuxThreads.
  - It adheres more closely to SUSv3 specification for Pthreads. Supported by Linux 2.6
  - **Students are required to go through the advantages of NPTL over LinuxThreads**

- To discover thread implementation on your machine/system give following command:

```
getconf GNU_LIBPTHREAD_VERSION
NPTL 2.5
getconf GNU_LIBC_VERSION
glibc 2.5
```

- On system that provides both NPTL and LinuxThreads, one may need to find out the default implementation. Or one may want to change the current default.
- **Students are required to do it at their own (Hint: LD_ASSUME_KERNEL)**

Pthread API

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t
*attr, void *(*start)(void *), void *arg) ;
```

Returns 0 on success, or a positive Exxx value on error.

1st argument: `pthread_t *tid`

- Each thread within a process is identified by a Thread ID,whose data type is **`pthread_t`**
- On successful creation of a new thread, its ID is returned through the pointer **`tid`**

2nd argument: `const pthread_attr_t *attr`

- This arguments specifies the attributes of the newly created thread.
- Normally we pass NULL pointer for default attributes.
- We can specify these attributes by initializing a **`pthread_attr_t`** variable that overrides the default.

Pthread API (cont...)

3rd argument: `void* (*start)(void*)`

- The third argument is the thread start function, which let us pass one pointer (to anything we want) to the thread function, and lets the thread function to return one pointer (to anything we want)
- The child thread starts be calling this function and then terminates either explicitly (by calling **`pthread_exit`**) or implicitly (by letting this function return)

4th argument: `void *arg`

- Fourth argument is a pointer of type void which points to the value to be passed to thread start function. It can be Null if you do not want to pass any thing to the thread function, can also be address of a structure if you want to pass multiple arguments.

Pthread API (cont...)

```
#include <pthread.h>
void pthread_exit(void *status);
```

- This function terminate the calling thread
- If the thread calling this function is not detached, its TID and exit status are retained for a later **`pthread_join()`** call 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.

Ways for a thread to terminate:

- The thread function returns (the return value is the exit status of the thread)
- The thread function calls **`pthread_exit()`**
- The main thread returns or call **`exit()`**
- Any sibling thread calls **`exit()`**

Pthread API (cont...)

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **retval);
```

On success return 0 and on error return errno

- A thread can wait for another thread to terminate by calling **`pthread_join()`** function, similar to **`waitpid()`**
- **1st argument:**
  - It is the TID of thread for which we wish to wait. Unfortunately, we have no way to wait for any of our threads like **`wait()`**
- **2nd argument:**
  - It can be **NULL**, if the parent thread is not interested in the return value of the child thread. Otherwise, it can be a double pointer which will point to the status argument of the **`pthread_exit()`**

Pthread API (cont...)

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- Return TID of calling thread. No error.
- Since **pthread_create()** does not return the TID of the child thread (as **fork()** do). This function is used to get the TID of a thread (as **getpid()** do).

```
#include <pthread.h>
int pthread_equal(pthread_t t1,pthread_t2);
```

- This function compares the TID of t1 and t2.
- Return a nonzero value if t1 and t2 are equal otherwise zero.
- If t1 and t2 are not valid TIDs, behavior is undefined.

Pthread Data Types

$ man 5 types.h

- The data type that end in _t are called primitive data types. Usually defined in **/usr/include/sys/types.h** .Their purpose is to prevent programs from using specific data types (e.g int, short, long) to allow each implementation to choose which data type is required for a particular system. You just have to recompile the application on another system.

pthread_t	**used to identify a thread**
pthread_mutex_t	**used for mutex**
pthread_cond_t	**used for condition variables**
pthread_attr_t	**used to identify a thread attribute object**
pthread_mutexattr_t	**used to identify a mutex attribute object**
pthread_condattr_t	**used to identify a condition attribute object**
pthread_rwlock_t	**used for read write lock**
pthread_rwlockattr_t	**used for read write lock attributes**
pthread_barrier_t	**used to identify a barrier**
pathread_barrierattr_t	**used to define a barrier attribute object**
pthread_once_t	**used for dynamic package installations**
pthread_spinlock_t	**used to identify a spin lock**

BASIC
MULTI-THREADED PROGRAMS

Example 1

`/threadmgmt/threadbasics/t1.c`

Main thread creates a child thread that prints X's in an infinite loop, while the main thread prints O's in an infinite loop. Both executes concurrently. Press ctrl+c to quit.

```
void * MyThreadFunc(void *nothing){
    while(1){
        putchar('x');   } }

int main(){
   pthread_t tid;
   int rv = pthread_create(&tid, NULL, &MyThreadFunc, NULL);
   if(rv != 0){
      switch(rv){
         case EAGAIN:
            printf("EAGAIN\n");break;
         case EINVAL:
            printf("EINVAL\n");break;
         case ENOMEM:
            printf("ENOMEM\n");break;
          }exit(1);   }
   while(1){
      putc('O', stdout);
      return 0;
      }
}
```

```
int putc(int c, FILE *stream)
int putchar(int c)
```

Compiling a multi-threaded program

- Use any editor to type your program and then to compile give following command:

```
$  gcc -c t1.c
```

- Then link the resulting .o file with **/usr/lib/libpthread.so** library

```
$  gcc t1.o -o t1 -lpthread -D_REENTRANT
```

- The code will execute only on machines which have the thread library installed on it

```
$  ./t1
```

Example 2

/threadmgmt/threadbasics/t2.c

Main thread creates a thread, pass it a message and then wait for its termination. The child thread executes the function func(), displays the message and returns to main

```
void *func(void *);
int main(){
   char *msg = "Hello Students";
   pthread_t tid;
   int rv = pthread_create(&tid, NULL, &func, (void *)msg);
   if(rv != 0) {
     printf("Thread creation failed\n");
     exit(1);
   }
   pthread_join(tid, NULL);
   printf("Exiting the main function...\n");
   return 0;
}
void *func(void *args){
   char *msg = (char *)args;//must cast the parameter to what is needed
   printf("I m child thread & the message passed to me is: %s\n", msg);
   pthread_exit(NULL);   //return NULL
}
```

Example 3

/threadmgmt/threadbasics/t3.c

Main thread creates two child threads, both threads execute the same function which receives a pointer to a structure. The structure contains the xter to be displayed and the count

```
struct mystruct{
  char character;
  int count;  };
void * func(void *);
int main(){
  pthread_t t1_id, t2_id;
  struct mystruct t1_args;
  struct mystruct t2_args;
// create child thread to print 30000 * x
  t1_args.character = 'x';
  t1_args.count = 30000;
  pthread_create(&t1_id, NULL, &func, (void*)&t1_args);
//create child thread to print 20000 o
  t2_args.character = 'o';
  t2_args.count = 20000;
  pthread_create(&t2_id, NULL, &func,    (void *)&t2_args);
/*make sure that main thread wait for child threads*/
  pthread_join(t1_id, NULL);
  pthread_join(t2_id, NULL);
  printf("\n I am main thread. Bye!\n");
  return 0;}
```

```
void *MyThreadFunc(void *args){
  struct mystruct *p=(struct
  mystruct*)args;
  int i;
  for(i = 0; i < p->count; i++){
     putc(p->character, stdout);
  pthread_exit(NULL);

}
```

Example 4

/threadmgmt/threadbasics/t4.c

Main thread creates a child thread, and sends an integer value to the child thread, which sleeps for that much seconds

```
void * func(void *);
int main(){
  int stime = 10;
  pthread_t tid;
  pthread_create(&tid, NULL, &func,(void*)stime);
  sleep(5);
  printf("Concurrrent execution: inside main thread...\n");
  pthread_join(tid, NULL);
  printf("I am main thread bye... \n");
  return 0;
}
void * func(void *args){
  Int stime = int(args);
  printf(“I am child thread and I will sleep for %i seconds \n”, stime);
  printf(“Sleeping...Zeeeeeeee \n”);
  sleep(stime);
  printf(“I am child thread and I am awake now... Good Bye! \n”);
  pthread_exit(NULL);
}
```

```
I am child thread and I will sleep for 10 seconds
Sleeping....Zeeeeeee
Concurrent execution: inside main thread...
I am child thread and I am awake now... Good Bye
I am main thread bye...
```

Returning Results From Threads

We know that every thread function is declared to return a pointer of type void. We can use following ways to return data from a thread function.

```
void *thread_function(void *args)
{
    //receive arguments
    //carry out processing
    ---
    ---
    ---
    int *result = whatevercomputed;
    return (void *)result;
}
```

This will fail because the variable **result** is local to the thread function, i.e. it is created on the stack of this particular thread and might not be available on the main function's stack.

```
void *thread_function(void * args)
{
    char buffer[64];
/*carryout processing & fill buffer
with something good*/
    return buffer;
}
```

This will also fails because the internal **buffer** is automatic and it vanishes as soon as the **`thread_function()`** returns.

```
void *thread_function(void *args)
{
    static char buffer[64];
//carryout processing & fill buffer
  with something good*/
    return buffer;
}
```

Here the **buffer** is made static so that it will continue to exist even after **`thread_function()`** terminates. However, this will also fail, if multiple threads run the same **`thread_function()`**. In this case the second thread will over write the static buffer with its own data and data written by the f irst thread will be over written.

Using global variables for returning values will also suffer from the same limitation

Returning Results From Threads (cont...)

```
void * thread_function(void * args){
  //receive arguments
  //carry out processing
  int *result = (int*)malloc(sizeof(int));      //for returning single integer
  int *result = (int*)malloc(sizeof(int)*size); //for returning an integer array
  char *result = (char*)malloc(sizeof(char)*size);   //for returning a xter array
  ---
  ---
  pthread_exit((void*)result); // return (void *)result;
}
//Now let's receive the result in main
  int main()  {
    ---
    void *exit_status;
    pthread_join(tid, &exit_status);
//now cast it to appropriate type and dereference it before using
    int *answer = (int*)exit_status;
    printf("Answer from thread \n",*answer);
    ---
    ---  }
```

Example 5

threadmgmt/threadbasics/t5.c

Program receives an integer via cmd line. The main thread creates a child thread & pass **n** to it. The child thread computes the sum of first **n** integers and return the value to the main thread.

```
void *MyThreadFunc(void *);
int main(int argc, char *argv[]){
  if(argc != 2){
    printf("invalid arguments...\n");
    exit(1);    }
  pthread_t tid;
  int args = atoi(argv[1]);
  void *exit_status;
  pthread_create(&tid, NULL, &MyThreadFunc, (void *)&args);
  pthread_join(tid, &exit_status);
  printf("\n Sum returned by child thread: %d\n",*(int*)exit_status);
  return 0;}
void *MyThreadFunc(void *args){
  int n = *((int *)args);
  int * result = (int*)malloc(sizeof(int));
  int i;
  for(i = 1; i <= n; i++)
    *result = *result + i;
  pthread_exit((void*)result);
}
```

Has to be a double ptr of void type

Example 6

threadmgmt/threadbasics/t6.c

Program creates an array of ten threads, each thread prints the argument passed to it

```
#define NUM_THREADS 10
void * MyThreadFunc(void * arg);
int main(){
   pthread_t tid[NUM_THREADS];
   int i;

   for(i=0; i < NUM_THREADS; i++){
      pthread_create(&tid[i], NULL, MyThreadFunc, (void *)&i);
      pthread_join(tid[i], NULL);
}
   printf("main(): Reporting that all child threads have termineted\n");
   exit(0);
}

void * MyThreadFunc(void * arg)
{
   int i = *((int*)arg);
   printf("I am child thread number %d \n", i);
   pthread_exit(NULL);
}
```

Example 7

threadmgmt/threadbasics/t7.c

“The child thread computes nth prime number and return to main.”

```
void *MyThreadFunc(void*);
int main(int argc, char *argv[]){
     if(argc != 2){printf("Error ...\n"); exit(1);}
     pthread_t tid;
     int args = atoi(argv[1]);
     pthread_create(&tid, NULL, &MyThreadFunc, (void*)&args);
     void *exit_status;
     pthread_join(tid, &exit_status);
     unsigned long * thread_result = (unsigned long*)exit_status;
     printf("\n Main(): the %dth prime number is %d\n", args, thread_result);
     return 0;
}

void *MyThreadFunc(void *args){
     int n = *((int*)args);
     unsigned long *candidate = (unsigned long *)malloc(sizeof(unsigned long));
     *candidate = 2;
     while(1)  {
          unsigned long factor ;
          unsigned long is_prime = 1;
          for(factor = 2; factor <= sqrt((*candidate)) ; ++factor){
               if((*candidate) %2 == 0){is_prime = 0; break;}
          }   //end of for loop
          if(is_prime)  {
               if(--n == 0)
                    pthread_exit((void*)(candidate));
          }
          ++(*candidate);
      }  //end of while loop
}
```

```
100       → 541
1000      → 7919
10000     → 104729
100000    → 1299709
1000000   → 15485863
10000000    179424673
```

Example 8

threadmgmt/threadbasics/t8.c

Take **size** of arrays via cmd line, create three arrays dynamically . Get I/P in arrays from user (you can take I/p from files). Also create **size** number of threads. Now each thread should add two locations of arrays and place result in corresponding third array. All arrays being global. Finally thread should display the result. ”

```
void * MyThreadFunc (void*);
int *arr1;
int *arr2;
int *result;
int mian(int argc, char *argv[]){
     if(argc != 2){printf("invalid arguments...\n"); exit(1);}
     int ctr = atoi(argv[1]);
     arr1   = (int*)malloc(sizeof(int)*ctr);
     arr2   = (int*)malloc(sizeof(int)*ctr);
     result = (int*)malloc(sizeof(int)*ctr);
     int i;
      for(i = 0 ; i < ctr; i++)
          scanf("%d",&arr1[i]);
     for(i = 0; i < ctr; i++)
          scanf("%d", &arr2[i]);
     pthread_t *tid = (pthread_t*)malloc(sizeof(pthread_t)*ctr);
     for(i = 0 ; i < ctr; i++)
          pthread_create(&tid[i], NULL, &MyThreadFunc,(void*)i);
     for(i = 0 ; i < ctr; i++)
          pthread_join(tid[i],NULL);
     printf("\n Main thread: The results are \n");
      sleep(5);
     for(i = 0 ; i < ctr; i++)
          printf("sum[%d] = %d", i, result[i]);
     return 0;
}
```

```
void *MyThreadFunc(void *args)
{
     int n = (int)args;
     result[n] = arr1[n]+arr2[n];
     pthread_exit(NULL);
}
```

THREAD ATTRIBUTES

Thread Attributes

- Every thread has a set of attributes which can be set before creating it and passed to the `pthread_create()` function as its second argument.
- If we pass a null pointer, the default thread attributes are used to configure the new thread

Attribute	Default value	Description
detachstate	PTHREAD_CREATE_JOINABLE	joinable by other threads
stackaddr	NULLL	stack allocated by system
stacksize	NULL	1 BM
priority	NULL	priority of calling thread
inheritsched	PTHREAD_EXPLICIT_SCHED	

Thread Attributes

- Steps to specify customized thread attributes:
  - Create a `pthread_attr_t` object
  - Call `pthread_attr_init()`, passing a pointer of above object
  - Modify the attribute object to contain the desired attribute value using the appropriate setters.
  - Pass a pointer to the attribute object when calling `pthread_create()`
  - Destroy pthread attribute object by calling `pthread_attr_destroy()`

Detach State (Avoiding Zombie Threads)

Most important attribute is the thread detach state having two values:

- **Joinable Thread:**
  - A joinable thread (like a process) is not automatically cleaned up by GNU/LINUX when it terminates
  - The thread's exit status hangs around in system until another thread calls **`pthread_join()`** to obtain its return value. Only then its resources are released.
  - For example whenever we want to return data from child thread to its parent thread the child thread must be a joinable thread.
- **Detached Thread:**
  - A detachable thread is cleaned up automatically when it terminates
  - Since a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using **`pthread_join()`** to obtain its return value
  - For example suppose the main thread crates a child thread to do back up of a file; while the main thread continues to service the user. When the backup is finished , the second thread can just terminate. There is no need for it to rejoin the main thread.

Detach State

```
int pthread_attr_getdetachstate(const
   pthread_attr_t *attr, int *detachstate);

int pthread_attr_setdetachstate
     (pthread_attr_t*attr, int detachstate);
```

- The above two functions are used to get and set the state attribute of a thread in the attribute object.
- The two possible detach states are:
  - PTHREAD_CREATE_JOINABLE
  - PTHREAD_CREATE_DETACHED

Example 9

Creating a detached thread: threadmgmt/threadbasics/tattr1.c

```
void *MyThreadFunc(void*);
char message[] = “Hello Students.”;
int thread_finished = 0;
int main(){
   pthread_t tid;
   //create an attribute object
   pthread_attr_t thread_attr;
   //initialize the attribute object to default values
   pthread_attr_init(&thread_attr);
   //modify attribute to detachstate
   pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
   //create thread with modified attributes
   pthread_create(&tid,&thread_attr,&MyThreadFunc,(void*)message);
   //destroy attribute object
   pthread_attribute_destroy(&thread_attr);
   while(!thread_finished){
      printf(“Waiting for thread to say, its finished...\n”);
      sleep(1);
   }
   printf(“Main thread exiting, Bye!\n”);
   exit(EXIT_SUCCESS);
}
```

Example 9 (cont...)

Creating a detached thread

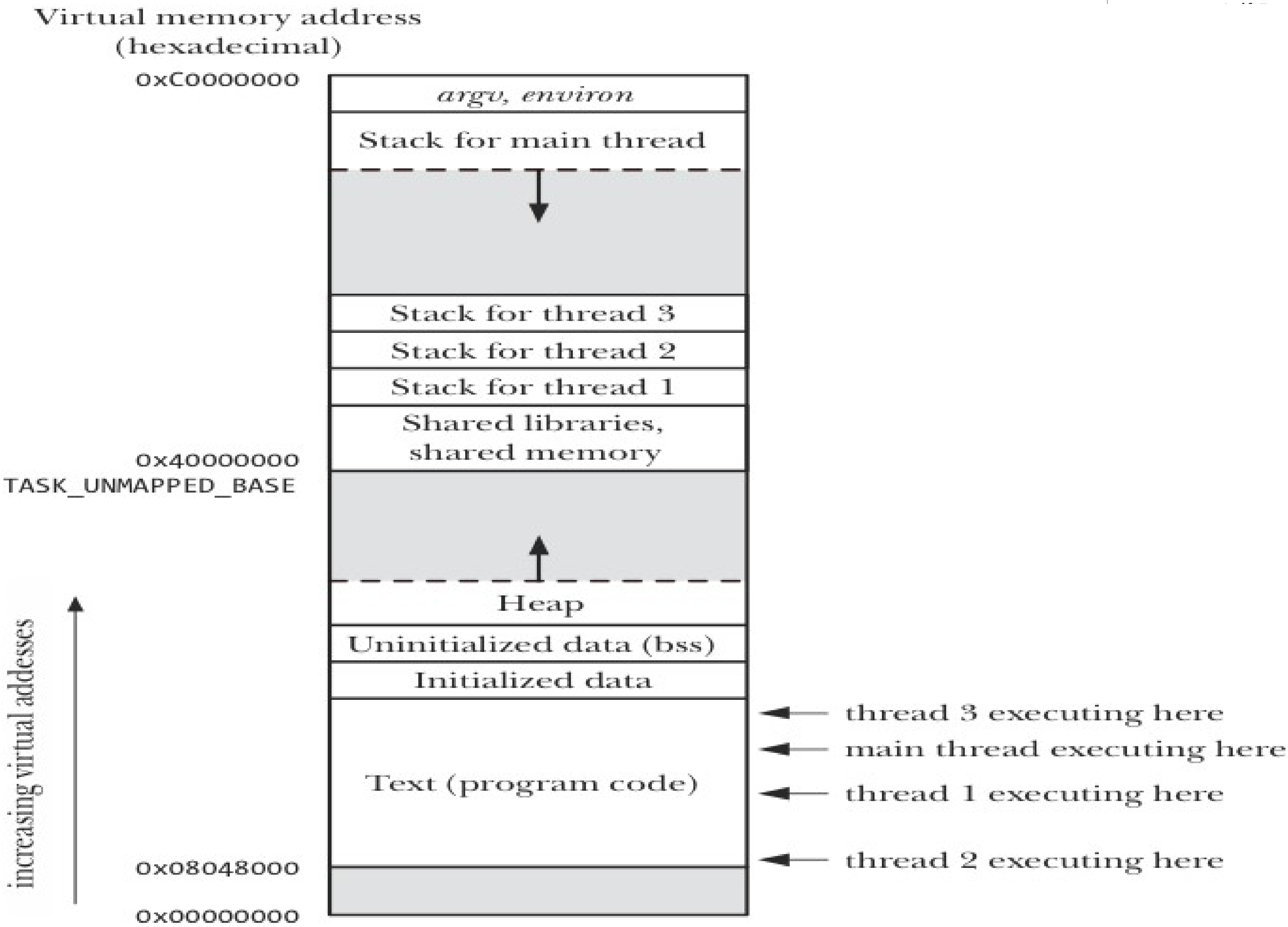
```
//thread function of previous example
void *MyThreadFunc(void *arg){
    printf("Child thread is running. Received message%s\n",(char*)arg);
    sleep(4);
    printf("Child thread setting the finished flag, and exiting now\n");
    thread_finished=1;
    pthread_exit(NULL);
}
```

Setters and Getters of pthread_attr_t object

```
Int pthread_attr_getschedpolicy(const pthread_attr_t
            *attr, int *policy);

int pthread_attr_setschedpolicy(pthread_attr_t
           *attr, int policy);
```

```
int pthread_attr_getinheritsched(const pthread_attr_t
         *attr, int *inheritsched);

int pthread_attr_setinheritsched(pthread_attr_t
            *attr, int inheritsched);
```

```
int pthread_attr_getstacksize(const pthread_attr_t*
           attr, size_t *stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr,
          size_t stacksize);
```



Virtual memory address (hexadecimal)
0xC0000000
argv, environ
Stack for main thread
Stack for thread 3
Stack for thread 2
Stack for thread 1
Shared libraries, shared memory
0x40000000
TASK_UNMAPPED_BASE
Heap
Uninitialized data (bss)
Initialized data
Text (program code)
thread 3 executing here
main thread executing here
thread 1 executing here
thread 2 executing here
increasing virtual addesses
0x08048000
0x00000000

Data Sharing Among Threads

Data Sharing Among Threads

Normally modifying an object requires several steps. While these steps are being carried out the object is typically not in a well formed state. If another thread tries to access the object during that time, it will likely get a corrupt information. The entire program might have undefined behavior after wards.

What data is shared?

- Global data and static local data. The case of static local data is only significant if two (or more) threads execute the function containing static local variable at the same time.
- Dynamically allocated data (in heap) that has had its address put into a global/static variable.
- Data members of a class object that has two (or more) of its member functions called by different threads at the same time.

Data Sharing among Threads (cont...)

What Data is not Shared ???

- Local variables are not shared. Even if two threads call the same function they will have different copies of the local variable in that function. This is because the local variables are kept on stack and every thread has its own stack.
- Function parameters are not shared. In Languages like C, the parameters of function are also put on the stack & thus every thread will have its own copy of those as well.

Threads share many data structures:Answer following Questions:

- What happens if one thread closes a f ile while another is still reading from it?
- What happens when one thread feels that there is too little memory & starts allocating more memory, soon after wards another thread of same process executes and do the same. Does the allocation happens once or twice?

Example 10

Checking shared data: `threadmgmt/threadsynch/mutex/shareddata.c`

```
char** ptr;  //only one instance of global variable ptr
void * thread_function(void * localarg);
int main(){
   int i;//local auto variable
   pthread_t tid;
//msg is a local variable on main thread's stack
   char* msg[2] = {"Hello from Arif", "Hello from PUCIT"};
   ptr = msg;
   for(i=0;i<2;i++){
      pthread_create(&tid, NULL, thread_function, (void*)i);
      pthread_join(tid,NULL);
   }
   return 0;
}

void * thread_function(void * localarg){//localarg is local for each thread
   int myid = (int)localarg;//myid is local for each thread
   static int svar = 0;//static variable svar is shared among threads
//myid is local to all threads as it is created on their respective stacks
   printf("[%d]: %s (svar = %d)\n", myid, ptr[myid], ++svar);
   pthread_exit(NULL);
}
```

Example 10 (cont...)

Analysis: A variable x is shared iff multiple threads reference at least one instance of x either directly or indirectly

Variable Instance	Referenced by main	Referenced by t0	Referenced by t1	
`ptr`	yes	yes	yes	shared
`i`	yes	no	no	
`msg`	yes	yes	yes	shared
`myid.t0`	no	yes	no	
`myid.t1`	no	no	yes	
`svar`	no	yes	yes	shared

Problem with Threads

Synchronization means making two things / events happen at the same time. It has two constraints:

a) **Serialization:** Event A must happen before event B.

b) **Mutual Exclusion:** Event A and B must not happen at the same time.

i. **Concurrent Programs** are non-deterministic in nature, which means it is not possible to tell by looking at the program, what will be the output when it executes (e.g. two threads within a program, one prints **"yes"** & other **"no"**. What will be printed first).

Problem with Threads (cont...)

ii.Important Concepts

a) **Race Condition:** The situation where several threads are reading or writing some shared data concurrently & the final value of the data depends on which thread finishes last.

b) **Critical Section:** A piece of code in cooperating threads/ processes in which the threads may update some shared data.

c) **Critical section Problem:** If multiple threads try to execute their CS section concurrently we need to execute them one by one completely.

d) **Atomic Operation:** An operation which can not be preempted in between. e.g. `LOAD`, `STORE`, `SWAP`, `TSL`, are atomic operations. (An operation that always runs to completion or not at all.)

Problem with Threads (cont...)

e) CSP Solution

f) Characteristics of good CSP Solution:

- **Mutual Exclusion:** If a process is executing in its CS, no other cooperating process or thread can execute their CS.
- **Progress:** If no process is executing in its CS, and some processes wish to enter in their critical section; two things need to happen:
  i. No process in <RS> should participate in the decision.
  ii. The decision has to be taken in finite time.
- **Bounded wait:** If a processes has requested to enter in its CS a bound must exist on the number of times that other processes are allowed to enter in their CS before the request of 1st process is granted.

Example 11

Showing race condition: threadmgmt/threadsynch/mutex/race1.c

```
#include<pthread.h>
int balance = 0;
void *inc(void *arg);
void *dec(void *arg);
int main()
{
   pthread_t t1,t2;
   pthread_create(&t1,NULL,inc,NULL);
    pthread_create(&t2,NULL,dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("value of balance is: %i\n", balance);
   return 0;
}
```

```
void * inc(void *arg)
{
   long i;
   for(i = 0; i< 1000000; i++)
      balance++;
   pthread_exit(NULL);
}
```

```
void * inc(void *arg)
{
   long i;
   for(i = 0; i< 1000000; i++)
      balance--;
   pthread_exit(NULL);
}
```

Synchronization among Threads
Mutexes

Mutexes

- To achieve both mutual exclusion as well as serialization, GNU/Linux provides mutexes (Mutual Exclusion) or locks.
- A mutex is a special type of lock that only one thread may lock at a time.
- If a thread locks a mutex & later a second thread also tries to lock the same mutex, the second thread is blocked. When the first thread unlocks the mutex, the second thread is allowed to resume execution.
- Linux guarantees that race condition do not occur among threads attempting to lock a mutex.

Typical way to use a mutex

i. Create and initialize a mutex variable

ii. Several threads attempt to lock the mutex

iii. Only one thread succeed and that thread owns the mutex

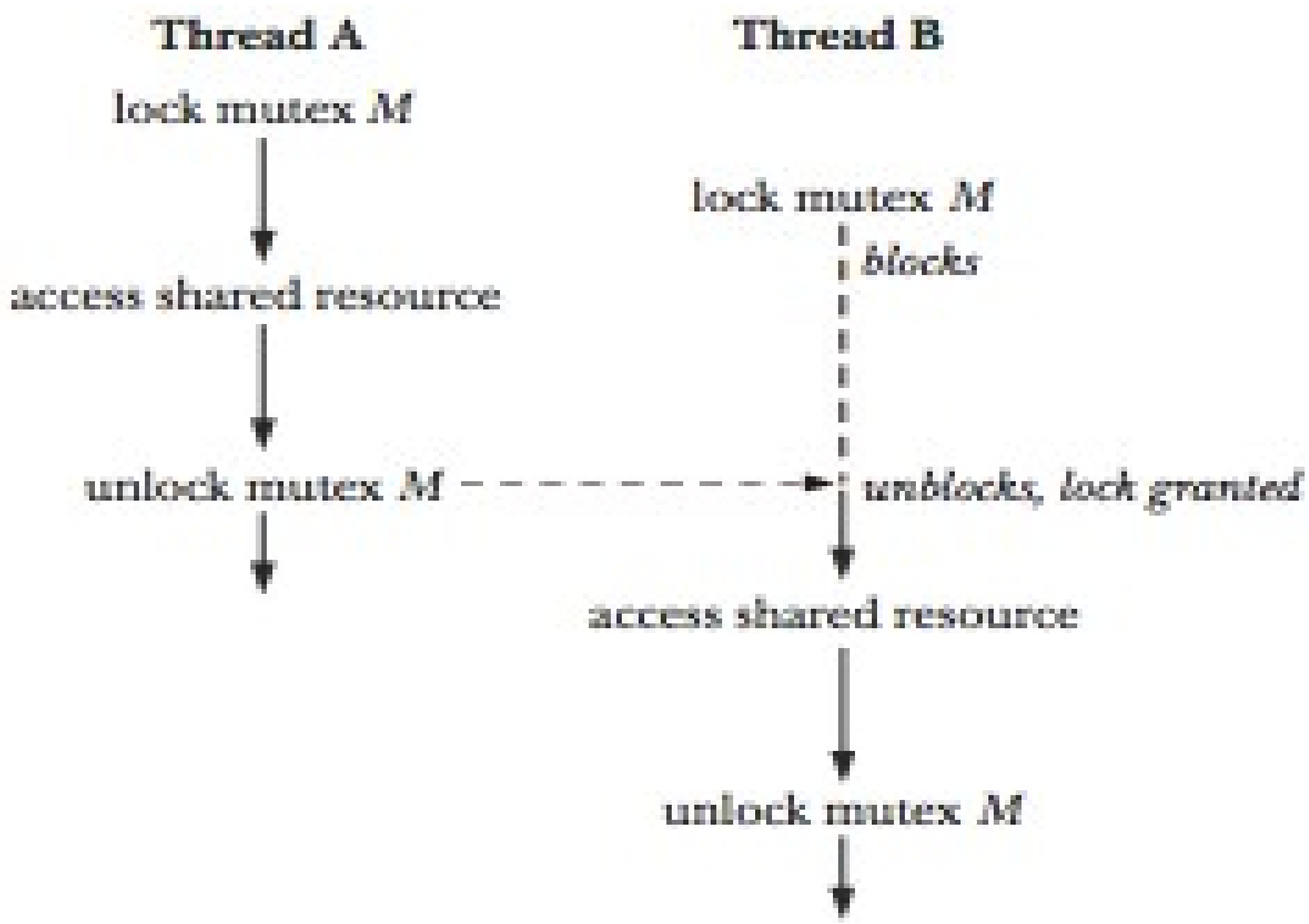
iv. The owner thread carry out operations on shared data

v. The owner threads unlock the mutex

vi. Another thread acquires the mutex and repeats the process

vii. Finally the mutex is destroyed

Typical way to use a mutex



Mutex Initialization

Static Initialization: In case where default mutex attributes are appropriate, the following macro can be used to initialize a mutex that is statically allocated.

```
static pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Run time initialization: In all other cases, we must dynamically initialize the mutex using `pthread_mutex_init()`

```
int pthread_mutex_init (pthread_mutex_t* mptr,
                        const pthread_mutexattr_t * attr);
```

Mutex Initialization

We must use `pthread_mutex_init()` rather than a static initializer in the following scenarios:

- The mutex was dynamically allocated on the heap. For example, suppose that we create a dynamically allocated linked list of structures, and each structure in the list includes a `pthread_mutex_t` field that holds a mutex that is used to protect access to that structure
- The mutex is an automatic variable allocated on the stack
- We want to initialize a statically allocated mutex with attributes other than the defaults

Locking, unlocking and destroying mutexes

```
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
int pthread_mutex_trylock(pthread_mutex_t *mptr);
int pthread_mutex_destroy(pthread_mutex_t *mptr);
// return 0 on success, else positive Exxx value on error
```

- **Lock()** will lock the mutex object referenced by **mptr**. If mutex is already locked, the calling thread shall block until the mutex become available.
- **Trylock()** is similar to lock except that if the mutex object is currently locked, the call shall return immediately with the error code **EBUSY**.
- **Unlock()** release the mutex object referenced by mptr. The manner in which a mutex is released is dependent on the mutex's attribute type. If there are threads blocked on the mutex object referenced by **mptr** when the **unlock()** is called, the scheduling policy shall determine which thread shall acquire the mutex.
- **Destroy()** shall destroy the mutex object referenced by **mptr**. The mutex object becomes uninitialized. A destroyed mutex can be reinitialized using **pthread_mutex_init()**.

Mutex Dead Locks

Be sure to observe following points to avoid dead locks while using mutexes:

i. No thread should attempt to lock or unlock a mutex that has not been initialized.

ii. Only the owner thread of the mutex (i.e the one which has locked the mutex) should unlock it

iii. Do not lock a mutex that is already locked

iv. Do not unlock a mutex that is not locked.

v. Do not destroy a locked mutex.

Example 12

Handling race condition: threadmgmt/threadsynch/mutex/race1mutex.c

```
#include<pthread.h>
int balance = 0;
void *inc(void *arg);
void *dec(void *arg);
pthread_mutex_t mut;
int main(){
    pthread_t t1,t2;
    pthread_mutex_init(&mut, NULL);
    pthread_create(&t1,NULL,inc,NULL);
    pthread_create(&t2,NULL,dec,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_mutex_destroy(&mut);
    printf("value of balance is: %i\n", balance);
    return 0;
}
```

```
void * inc(void *arg){
    long i;
    for(i = 0; i< 1000000; i++){
      pthread_mutex_lock(&mut);
      balance++;
      pthread_mutex_unlock(&mut);
    }
    pthread_exit(NULL);
}
```

```
void * inc(void *arg){
    long i;
    for(i = 0; i< 1000000; i++){
       pthread_mutex_lock(&mut);
       balance--;
       pthread_mutex_unlock(&mut);
    }
    pthread_exit(NULL);
}
```

Example 13

Program receives two file names via command line. Create two threads and pass them one file name each. Both threads execute the function count_words() and update the global variable word count. Finally main thread displays the final value of word count

```
//threadmgmt/threadsynch/mutex/race2.c
int wordcount = 0;//global variable
void * count_words(void * arg);
int main(int argc, char* argv[]){
   if(argc != 3){
      printf("Must pass two file names...\n");
      exit(1);
   }
   pthread_t t1, t2;
   pthread_create(&t1, NULL, &count_words, (void*)argv[1]);
   pthread_create(&t2, NULL, &count_words,(void*)argv[2]);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Total Words: %d\n", wordcount);
   return 0;
}
```

Example 13 (cont...)


```
void* count_words(void* args){
   char* filename = (char*)args;
   int fd = open(filename, O_RDONLY);
   char ch;
   char prevch = '\0';
   while((read(fd, &ch, 1)) != 0){
      if(!isalnum(ch) && isalnum(prevch))
          wordcount++;
      prevch = ch;
   }
  close(fd);
}
```

Mutex Attributes

- A mutex has a set of attributes which can be set before creating it and passed to the `pthread_mutex_init()` function as its second argument. (which we have kept NULL in previous examples)
- Various Pthreads functions can be used to initialize and retrieve the attributes in a pthread_mutexattr_t object. We won't describe the prototypes of the various functions that can be used to initialize the attributes in a `pthread_mutexattr_t` object.
- However, on next slide we'll briefly describe one of the attributes that can be set for a mutex: **its type**.

Mutex Attributes (cont...)

In case of a deadlock, behavior depends on type of mutex:

PTHREAD_MUTEX-DEFAULT (standard mutex)

- Locking an already locked mutex results in undefined behavior.
- Unlocking an already unlocked mutex results in undefined behavior.
- Unlocking a mutex that is not locked by calling thread results in undefined behavior.

PTHREAD_MUTEX_NORMAL (fast mutex)

- Locking an already locked mutex results in deadlock.
- Unlocking an already unlocked mutex results in undefined behavior.
- Unlocking a mutex that is not locked by calling thread results in undefined behavior.

Mutex Attributes (cont...)

PTHREAD_MUTEX_ERRORCHECK (error checking mutex)

- Locking an already locked mutex returns an error.
- Unlocking an already unlocked mutex returns an error.
- Unlocking an mutex that is not locked by calling thread returns an error.

PTHREAD_MUTEX_RECURSIVE (recursive mutex)

- Mutex maintains a concept of lock count. When a thread successfully acquires a mutex for the first time, the lock count is incremented by one. Each time the threads unlock the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex become available for other thread to acquire.
- Unlocking an unlocked mutex returns an error.
- Unlocking a mutex that is not locked by the calling thread results in undefined behavior.

Producer Consumer Problem

Producer-Consumer Problem

- Producer produces information that is consumed by a consumer process.To allow producer and consumer run concurrently we must have a buffer that can be filled by the producer and emptied by the consumer.The buffer can be bounded or unbounded.
- **Unbounded Buffer:** Places no practical limit on the size of the buffer. The consumer may have to wait for new items if the buffer is empty, but the producer can always produce new items.
- **Bounded Buffer:** Assumes a fixed size buffer. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer. If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Producer-Consumer Problem

- **Implicit Synchronization:**

```
$ grep prog1.c | wc -l
```

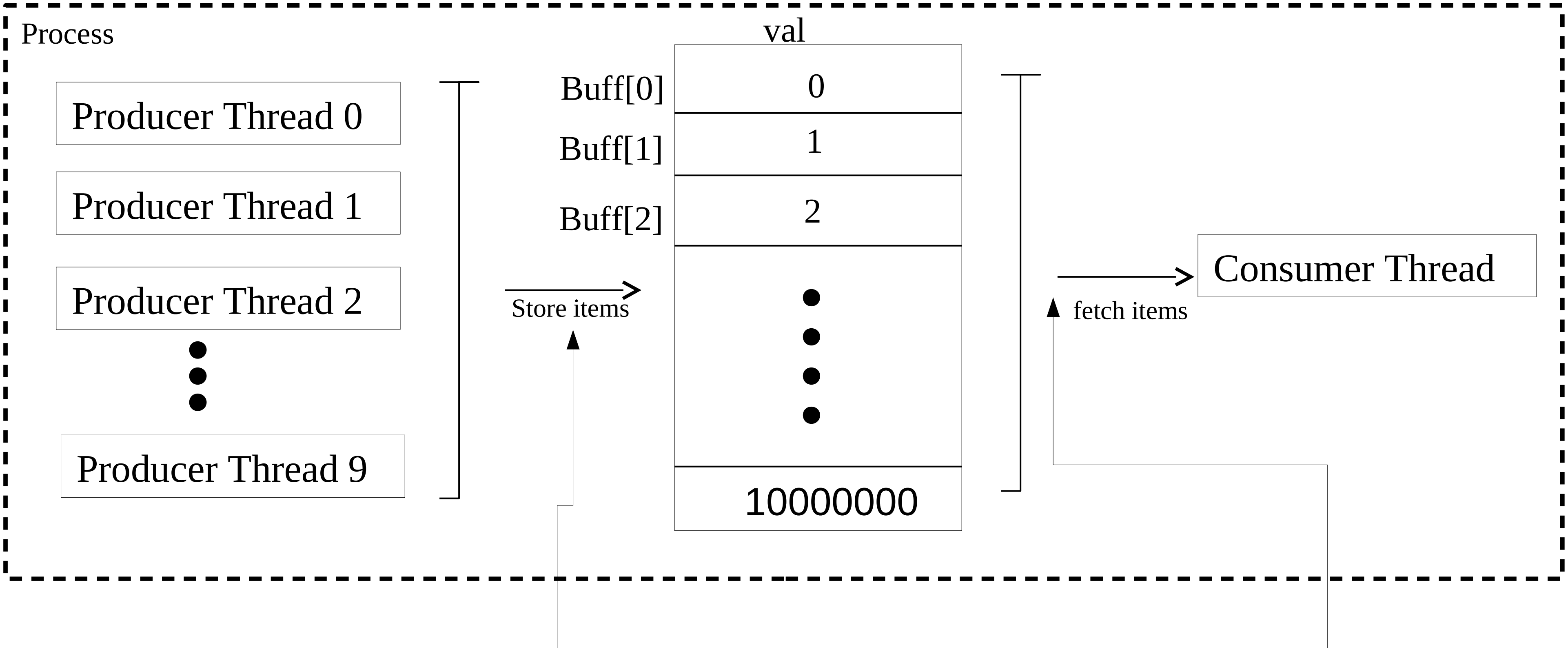
  - **`grep`** is the single producer and **`wc`** is the single consumer. The required synchronization is handled implicitly by the kernel. **`grep`** writes into the pipe and **`wc`** reads from the pipe. If producer get ahead of the consumer (i.e. the pipe fills up), the kernel puts the producer to sleep when it calls **`write(),`** until more room is available in the pipe. If consumer gets ahead of the producer (i.e. the pipe is empty), the kernel puts the consumer to sleep when it calls **`read(),`** until some data is there in the pipe.
- **Explicit Synchronization:** When we as programmers are using some shared memory/data structure, we use some form of IPC between the procedure and the consumer for data transfer. We also need to ensure that some type of explicit synchronization must be performed between the producer and consumer.

Example: Producer-Consumer Problem

Description of Example:

- We have multiple producer threads and a single consumer thread within a single process.
- A memory buffer **buff** is shared between producer threads and consumer thread.
- Producer threads just set buff[0] to 0, buff[1] to 1 and so on.
- We do not start the consumer thread until all the producers are done.
- Once the buffer is full, the only consumer thread goes through this array and verifies that each entry is correct.
- So we just need to synchronize between multiple producer threads.

Producer-Consumer Example



Each producer thread accesses the buffer at the location pointed to by **in** and places the value **val** at that location. Afterwards the thread increments both the variables **in** and **val**. Finally the thread also increment its own count

Consumer simply traverses the entire buffer and checks whether index i contains value i or not.

Producer-Consumer Example

```
//threadmgmt/threadsynch/mutex/producerconsemer/prodcons1.c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define SIZE 10000000 //buffer size
#define MAXTHREADS 10 // total number of producer threads
struct myobject{
    pthread_mutex_t mutex;
    int buff[SIZE];
    int in; //next index where to store item in the buffer 0 to size
    int val; //next value to be stored in the buffer 0 to size
};
struct myobject shared;
void* produce(void*);
void* consume(void*);
```

```
int main(){
  pthread_mutex_init(&shared.mutex, NULL);
  shared.in = 0; shared.val = 0; int i = 0;
  pthread_t tid_producers[MAXTHREADS];
  pthread_t tid_consumer;
  int count[MAXTHREADS]; /*Will contain the number of items produced
//initialize the entire count array to zero and then pass the respective count
variable to produce function. /*
  for(i =0; i< MAXTHREADS; i++){
     count[i] = 0;
     pthread_create(&tid_produces[i],NULL,&produce,&count[i]);
  }
/*wait for all the producer threads and as each thread returns
print the number of items that particular thread has produced./*
  for(i =0; i< MAXTHREADS; i++){
      pthread_join(tid_producers[i], NULL);
      printf("count [%d] = %d\n", I, count[i]);
}
//start the only consumer thread and wait for its completion
   pthread_create(&tid_consumers, NULL, &consume, NULL);
   pthread_join(tid_consumer, NULL);
   return 0;
}
```

```
void * produce(void * args){
   while(1){
     pthread_mutex_lock(&shared.mutex);
      if(shared.in >= SIZE){
         pthread_mutex_unlock(&shared.mutex);
          return(NULL);
      } //if buffer is full, job is done and this producer thread
returns w/o creating any more item
     shared.buff[shared.in] = shared.val;
      shared.in ++;
     shared.val++;
     pthread_mutex_unlock(&shared.mutex);
      *((int*) args) += 1; // this actually is incrementing the count of
items this thread has produced. Since each thread has its own counter
in the count array, so we have not included it in the CS
   }
}
/* no need of synchronization because only one consumer thread executes
this function and that too after all the producer threads have finished
  void* consume(void * args){
     int i;
    for(i = 0; i < SIZE; i++){
        if(shared.buff[i] != i)
           printf("buf[%d] = %d\n", i.shared.buff[i]);
    }
     return(NULL);
}
```

Producer-Consumer Problem

Output:

$ ./prodcons1

Count[0] = 2466731
Count[1] = 1
Count[2] = 303104
Count[3] = 3264511
Count[4] = 2593791
Count[5] = 1371861
Count[6] = 0
Count[7] = 0
Count[8] = 0
Count[9] = 0

Total count of items produced is 10000000

If you comment the lock and unlock statements in the `produce()` func, the program will execute fast but the count of items produced will not be 10000000. This has been shown in prodconsrace.c

Producer-Consumer Example 2

```
//prodcons2.c
```

Let us modify the previous example and start the consumer thread immediately after all the producer threads have started.

- Just cut the LOC where you are creating the consumer thread and place it immediately after the LOC where you have created the producer threads.
- Once you execute the above program, that the consumer will try to consume the items that have yet not been produced.
- We must now synchronize the consumer with the producer to make sure that the consumer must wait if buffer is empty. Or should process only those data items that have already been produced by the producer threads.

Producer-Consumer Example 2

```
//prodcons3.c
void consume_wait(int i){
   while(i){
   pthread_mutex_lock(&shared.mutex);
   if(i < shared.in){
        pthread_mutex_unlock(&shared.mutex);
        return;
   }  //if index variable i sent by consumer thread doesnot contain data yet consumer spins
   pthread_mutex_unlock(&shared.mutex);
   }
}
void * consume(void* args){
   int i;
   for(i = 0; i < SIZE; i++){
     consume_wait(i);//consumer calls wait() before fetching next item from the buffer
      if(shared.buff[i] ! = i)
           printf("buf[%d] = %d\n", i. shared.buff[i]);
   }
    return(NULL);
}
```

Shared.in points to the location where next item is to be placed, so if (i < in) that means the thread can consume data at index i, so it retunrs, otherwise the consumer thread spins

Producer-Consumer Problem

Points to ponder:

- The `consume_wait()` function must wait until the producers have generated the i^{th} item.
- What should consumer do when the desired item is not ready?
- Loop around locking and unlocking mutex each time. This is called spinning or polling and is a waste of CPU time.
- Consumer could sleep for a short amount of time, but we do not know how long to sleep.
- So we need another type of synchronization that lets a thread or process sleep until some event occurs.
- Logically we want to sleep inside the CS, but if the buffer is empty and the consumer go to sleep inside the CS, the producer will not be able to produce the item and the consumer will sleep forever.

Synchronization among Threads
Condition Variables

Condition Variables/ Monitor

- **Condition variable** enable a thread to sleep inside a CS. Any lock held by the thread is automatically released when the thread is put to sleep.
- A **mutex** is for **locking** and a **condition variable** is for **waiting**.
- We know that mutexes are used to protect critical regions of code, so that only one thread is executing with Critical Section at a particular instance of time.
- Sometimes a thread obtains a mutex lock and then discovers that it need to wait for some condition to be true. For example, a consumer thread wants to consume an item from an empty buffer and blocks there i.e. inside the CS. Now the producer cannot place item in buffer.
- Solution is condition variable, i.e. let the consumer sleep and release the lock so that producer can produce.
- Condition variables are given another name in some books that is **monitors**. It's a programming language construct available in Java and not in C++.

Condition Variables/ Monitor

- Condition variables support three operations:

  `wait()` → release lock, goto sleep,reacquire lock after you are awoken up

  `signal()` → wake up a sleeping thread on this condition variable.

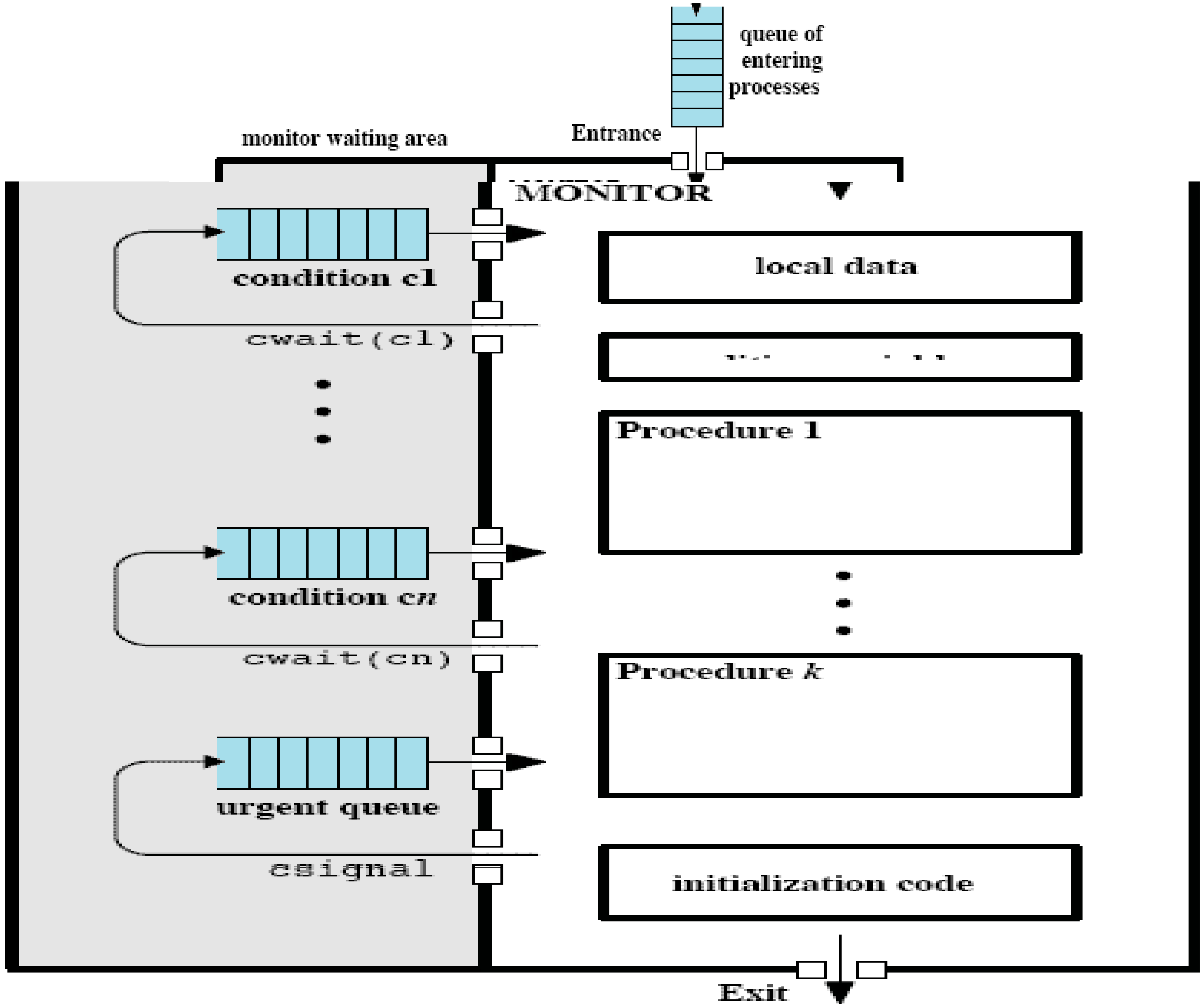
  `broadcast()` → wake up all waiters.

Note: With every condition variable there is an associated lock/mutex. Whenever a thread wants to invoke any of the above operations, it must hold the lock associated with that condition variable.

Monitor

- Monitor is a programming language construct that has been implemented in number of programming languages like Concurrent Pascal, Pascal-Plus, Module-2, Module-3 and Java.The concept was first defined by Hoare, C in 1970.
- A monitor is **similar** to a class that ties data and operation together. It can contain procedures, initialization code and shared data.
- A monitor is **similar** to a class in sense that its private data can only be accessed by its methods.
- A monitor is **different** from a class in same sense that it allows only a single process at a time to execute its procedure.
- In order to turn a Java class into a monitor:
  - Make all data private.
  - Make all methods synchronized.

Structure of Monitor



Pthread Condition Variables

- Condition variable is a variable of type **`pthread_cond_t`** on which a thread can call **`wait()`, `signal()`, `broadcast()`**.

```
int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
int pthread_cond_broadcast(pthread_cond_t *cptr);
```

- Every call to **`pthread_cond_wait()`** should be done as part of a conditional statement. e.g.

```
  if(flag == 0)
      pthread_cond_wait(…);
```

- en the producers and the consumer threads. so,

```
pthread_mutex_lock(&mutex);
if(flag == 0)
    pthread_cond_wait(&condition, &mutex);
pthread_mutex_unlock(&mutex);
```

Pthread Condition Variables

- The thread that signals this condition will use the same mutex to gain exclusive access to the flag. Thus there is no way that the signaling could occur between the test of the flag and waiting on the condition.
- For above code to work, **`pthread_cond_wait()`** needs to wait on the condition and unlock the mutex as an **atomic action**. It does this, but it needs to know which mutex to unlock. Hence the need of the 2nd parameter of **`pthread_cond_wait()`**.
- When the condition is signaled, **`pthread_cond_wait()`** will lock the mutex again before returning so that the **`pthread_mutex_unlock()`** in above example is appropriate.

Pthread Condition Variables

- Here is how the signaling thread might look like:

```
pthread_mutex_lock(&mutex);
flag = 1;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&condition);
```

- **`pthread_cond_signal()`** releases only one thread at a time. In some cases it is desirable to release all threads waiting on a condition. This can be done using **`pthread_cond_broadcast()`**

```
pthread_mutex_lock(&mutex);
flag = 1;
pthread_mutex_unlock(&mutex);
pthread_cond_broadcast(&condition);
```

Pthread Condition Variables

Problem:

- Under certain conditions the **`wait()`** function might return even though the condition variables has not actually been signaled.
- For example, if a Linux process receives a signal, the thread blocked in **`pthread_cond_wait()`** might be elected to process the signal handling function. Thus the thread might come out of **`wait()`** (which should not happen).

Solution:

- A solution to this problem is simply retest the condition after **`pthread_cond_wait()`** returns. This is most easily done using a while loop, e.g.

```
pthread_mutex_lock(&mutex);
while(flag == 0) pthread_cond_wait(&condition, &mutex);
pthread_mutex_unlock(&mutex);
```

Threads Safety

Threads Safety

- Functions called from a thread must be thread safe. There are four classes of thread unsafe functions:

Class I: Failing to protect sheared variables.

(Solution: Use Locks to protect shared variable)

Class II: Relying on persistent state across invocations.

Class III: Returning a pointer to a static variable.

Class IV: Calling a thread unsafe function.

(Solution: Do Not Call Thread unsafe Functions.)

Threads Safety(cont...)

Solution to Class II:

- e.g. random number generator functions relies on static state.
- Solution is rewrite functions so that call passes all necessary state as argument. i.e. caller keep with itself the seed and passes it as argument to rand.

Limitation: you need to change interface of function if it already exists in library. Moreover if previous function is used by programs, you need to change them and recompile.

```
// rand- return pseudo random integer
int rand(){
   static unsigned int next = 1;
   next = next *1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}
// srand - set seed for rand()
void srand (unsigned int seed){
   next = seed;
}
```

Threads Safety(cont...)

Solution to Class III: (Function that returns a pointer to static variable)e.g.

- DNS name resolution functions in LINUX.

```
struct hostent* gethostbyname(const char* name);
```

- It uses a static variable where it stores the name of the host. So if this function is called by various threads then problem may arise i.e one thread requesting IP of "condor" may get IP of "linuxclient" which another thread has requested for.
- **Fix 1: Rewrite the library function**

```
int gethostbyname_r(const char *name,struct
hostent *ret, char *buf,size_t buflen,struct
hostent **result, int *h_errnop);
```

Threads Safety(cont...)

Solution to Class III: (Function that returns a pointer to static variable)

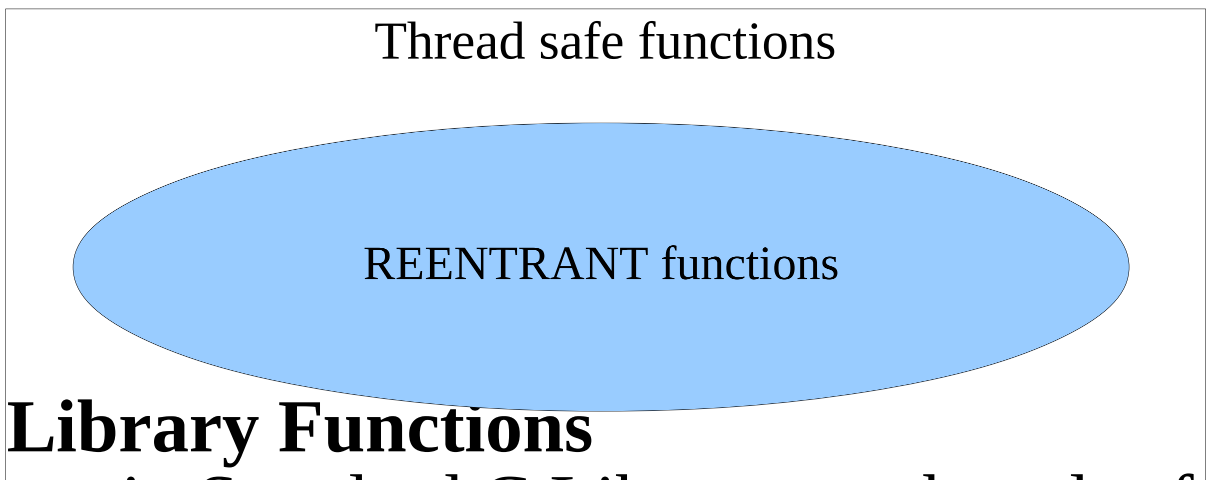
- **Fix 2: Lock and Copy**
- This requires only simple changes in caller and none in callee.
- Write a wrapper function around `gethostbyname()`

```
struct hostent* gethostbyname_ts(const char* name){
   struct hostent* q = malloc(---);
   wait(&mutex);  //lock
   p = gethostbyname(name);
   *q = *p;          //copy
   signal(&mutex); //unlock
   return q;
}
```

`gethostbyname()`

REENTRANT Functions/Code

- A function is REENTRANT if and only if it accesses NO shared variable when called from multiple threads.
- REENTRANT functions are proper subset of the set of thread safe functions.
- Not all threads safe functions are REENTRANT.



Thread Safe Library Functions

a) All functions in Standard C Library are thread safe; e.g. **`malloc(), free(), printf(), scanf(),...`**

b) Most UNIX calls are thread safe with a few exceptions.

REENTRANT Functions/Code (cont...)

Thread Unsafe Functions	Thread Safe Functions (REENTRAMT versions)
`asctime()`	`asctime_r()`
`ctime()`	`ctime_r()`
`gethostbyname()`	`gethostbyname_r()`
`gethostbyaddr()`	`gethostbyaddr_r()`
`inet_ntoa()`	----
`localtime()`	`localtime_r()`
`rand()`	`rand_r`

So always compile your multi threaded code with _REENTRANT defined.

```
$gcc -c thread1.c -D_REENTRANT
$gcc thread1.o -o thread1 -lpthread
```

OR

```
$gcc thread1.c -o thread1 -lpthread -D_REENTRANT
```

QUESTION 1

Errno is a global variable used in UNIX systems. What problem can occur due to this shared variable in mutli-threaded program?

ANSWER 1

- If all threads were to store error codes in the same global `errno` variable, then the value of `errno` after system call or library function would be unpredictable.
- It may be possible that between the time a system call stores its `errno` and your code inspect this global variable to see which error has occurred, another thread might have stored another code in the same `errno` variable.

$100 QUESTION 2

Why all multi-threaded code must be compiled with `-D_REENTRANT` def ined? What difference does it make?

ANSWER 2

- It affects include file in three ways:
  1. If `_REENTRANT` is defined, the include files define prototypes for the `_REENTRANT` variant of some of the standard library functions_ _e.g. `gethostbyname_r()` as a `_REENTRANT` equivalent to `gethostbyname()`
  2. If `_REENTRANT` is defined, some `<stdio.h>` library functions are no longer defined as macro; e.g. `getc()` and `putc()`. In multi-threaded programs, `<stdio.h>` library functions require additional locking which macros' don't perform, so we must call function instead.
  3. If `_REENTRANT` is defined, `<errno.h>` library redefines `errno`, so that errno refers to thread specific error location rather than global variable. This is achieved by the following:

```
#define errno(*(_errno_location()))
```

Which causes e… …taining location where error coc…

QUESTION 3

Consider a multi-threaded code containing `read()` system call.What happens if it is not compiled with `-D_REENTRANT`?

ANSWER 3

```
do{
   r = read(fd, buf, n);
   if (r == -1){
      if (errno == EINTR)
         continue;
      else{
         perror("read fail");
         exit(100);
      }
   }
}while(...);
```

Remember, C Library itself is compiled with -D_REENTRANT, `read()` stores its error code in location pointed to by `_errno_location()`, which is the thread local errno variable.

Now, consider above code and lets assume that when a thread is executing the function `read()` it is interrupted. `read()` returns -1 and sets errno to EINTR. Since _REENTRANT is not defined in your application, the reference to errno accesses global errno variable, which is most likely 0. Hence the code prints error message and exits.

QUESTION 4

If a child process closes a file descriptor inherited from the parent, that file descriptor is still open for the parent.

In a multithreaded program if the same file descriptor is passed to two threads, if one of the thread closes the descriptor, what happens?

QUESTION 5

If one of the threads executes the `exec()` system call, what happens?

If one of the threads calls `exit()`, what happens?

If a thread causes a segmentation violation, the thread crashes. What about the process?

If a signal is sent to a multi-threaded process. Which thread will receive that signal?

QUESTION 6

If one thread executes the `fork()` system call, does the new process duplicate only the calling thread or all threads? Is it single threaded or multi-threaded?

$100 ANSWER !!!

- Some UNIX systems implement it both ways by having two versions of **`fork()`**:
  a) One that duplicates all threads.
  b) Other that duplicates only the thread that invoked the **`fork()`**.
- If **`exec()`** after **`fork()`**, then replace only the calling thread, as the new process will replace the whole calling process anyway.
- If no **`exec()`** after **`fork()`**, then duplicates the whole process with all the threads, not just the calling thread.
- Write a program to check what is the default behavior of your Linux implementation

Thread Cancellation

Thread Cancellation

- A process can terminate:
  1. If main function execute **`return`** statement.
  2. If any of the thread call **`exit()`**
  3. A process receives a term **`signal`**

- It is possible for a thread to request that another thread terminate. This is called canceling a thread. e.g., Suppose multiple threads are searching through a database, if one thread returns data, remaining threads might be cancelled.
- Thread to be cancelled is called the target thread.

Thread Cancellation(cont...)

Problems:

- Often a thread may be in some code that must be executed in all-or nothing fashion. e.g., a thread may allocate some resources, use them and then deallocate them. If the thread is cancelled in middle of this code, resources don't get deallocated. To counter this possibility, it is possible for a thread to control whether and when it can be cancelled. A thread may be in one of three states with regard to thread cancellation.
  - A thread may be **asynchronously cancelable** i.e., thread may be cancelled at any point in its execution.
  - A thread may be **synchronously cancelable**. A thread may be cancelled but NOT at any point in its execution. The particular places in a thread's execution where that thread can be cancelled are called cancellation points. The thread will queue a cancellation request until it reaches next cancellation point.
  - A thread may be **uncancellable** i.e., attempts to cancel thread are quietly ignored.

Thread Cancellation(cont...)

- A thread calls **`pthread_cancel()`** to request that another thread be cancelled.
- The target thread's type and cancel-ability state determine the result.

```
int pthread_cancel(pthread_t tid);
```

Returns: 0 on success and nonzero on error

- The above function doesn't cause the caller to block while the cancellation completes, rather, **`pthread_cancel()`** returns after making the cancellation.
- Other related functions are:

`pthread_setcancelstate()`
`pthread_setcanceltype()`
`pthread_testcancel()`

Thread Cancellation(cont...)

```
int pthread_setcanelstate(int newState,
                          int *oldState);
```

- **`pthread_setcancelstate()`** system call changes the cancel ability **state** of the calling thread
- Possible values of states are
  - **`PTHREAD_CANCEL_ENABLE`**
  - **`PTHREAD_CANCEL_DISABLE`**
- On success, it returns 0,
- On failure it returns non-zero value.

Thread Cancellation(cont...)

```
int pthread_setcanceltype(int newType, int *oldType);
```

- **`pthread_setcanceltype()`** system call changes the cancel ability **type** of the calling thread
- Possible values of states are
  - **`PTHREAD_CANCEL_DEFERRED`**
  - **`PTHREAD_CANCEL_ASYNCHRONOUS`**
- On success, it returns 0,
- On failure it returns non-zero value.

Thread Cancellation(cont...)

```
void pthread_testcancel(void);
```

- **`pthread_testcancel()`** creates a cancellation point in the calling thread.
- It has no effect if cancel ability is disabled. (it is used when a thread is synchronously cancel able.)

Things To Do

If you have problems visit me in counseling hours. . . .