

2026

CS 332

Subject 4

NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details



Module 1

*Operating
Systems:
Internals
and Design
Principles*

Chapter 4 Threads

Ninth Edition
By William Stallings

Processes and Threads

Resource Ownership

Process includes a virtual address space to hold the process image

- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

Follows an execution path that may be interleaved with other processes

- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

Processes and Threads

- The unit of dispatching is referred to as a *thread* or *lightweight process*
- The unit of resource ownership is referred to as a *process* or *task*
- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach

- MS-DOS is an example

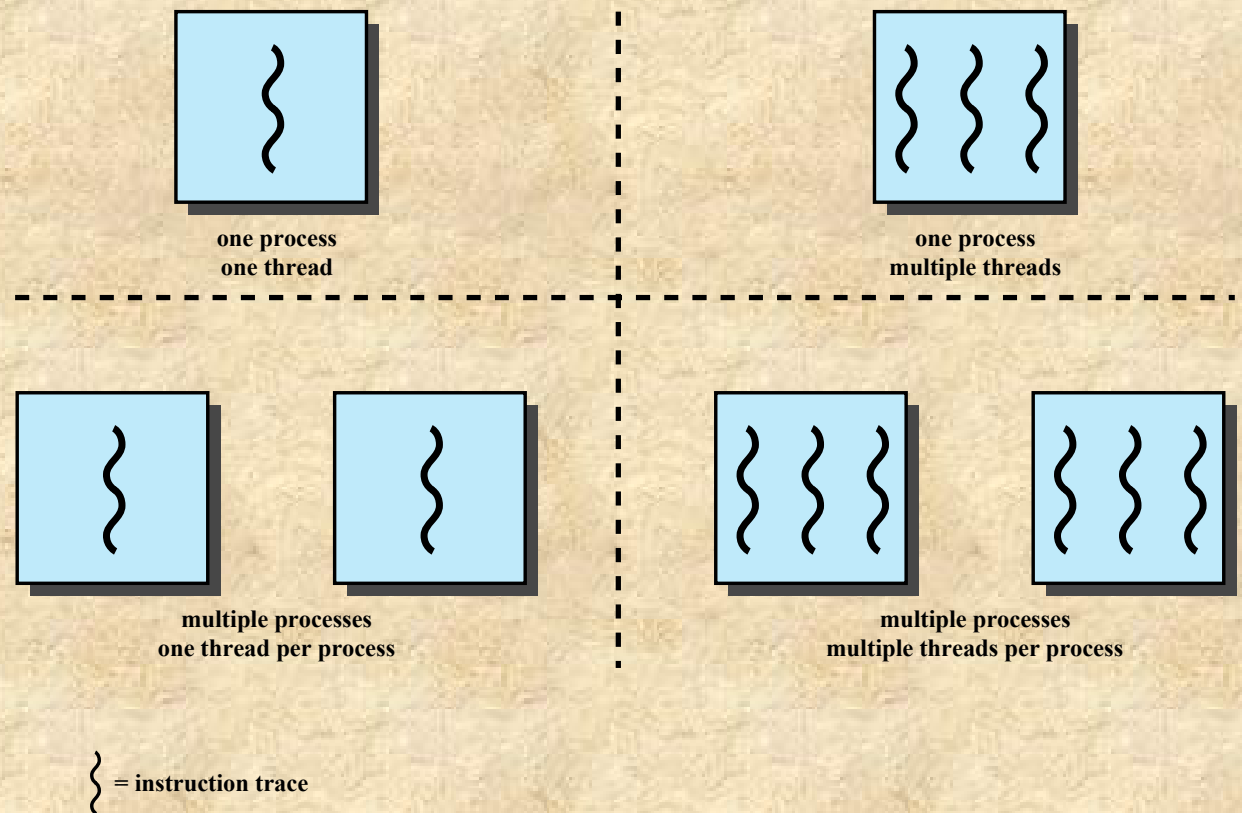
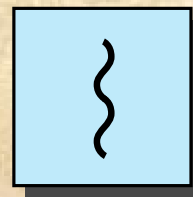


Figure 4.1 Threads and Processes

Multithreaded Approaches

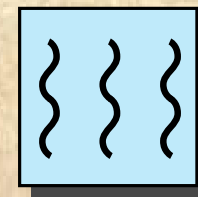
- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

one-process i.e.
single threaded process

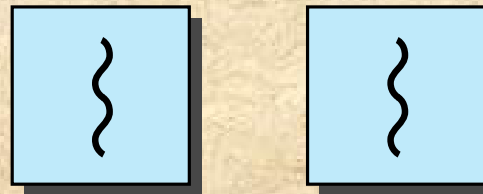


one process
one thread

multiple threads i.e.
multithreaded process

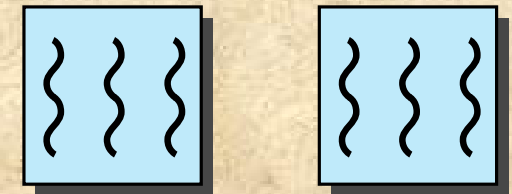


one process
multiple threads



multiple processes
one thread per process

multiprocess i.e.
multiple process
} = instruction trace



multiple processes
multiple threads per process

multithread aka
multithreaded multiprocess

Figure 4.1 Threads and Processes

Process

- Defined in a multithreaded environment as “the unit of resource allocation and a unit of protection”
- Associated with processes:
 - A virtual address space that holds the process image
 - Protected access to:
 - Processors
 - Other processes (for interprocess communication)
 - Files
 - I/O resources (devices and channels)

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its processes, shared with all other threads in that process



Process vs Thread

A Process (say with PID 4) has

1. **PCB (Process id)**
2. **PCB (Processor state i.e register file [PC,AC])**
3. **PCB (Process Control)**

7. **TEXT**
8. **Init**
9. **Uninit(aka bss)**
10. **Heap**
11. **Shmem**

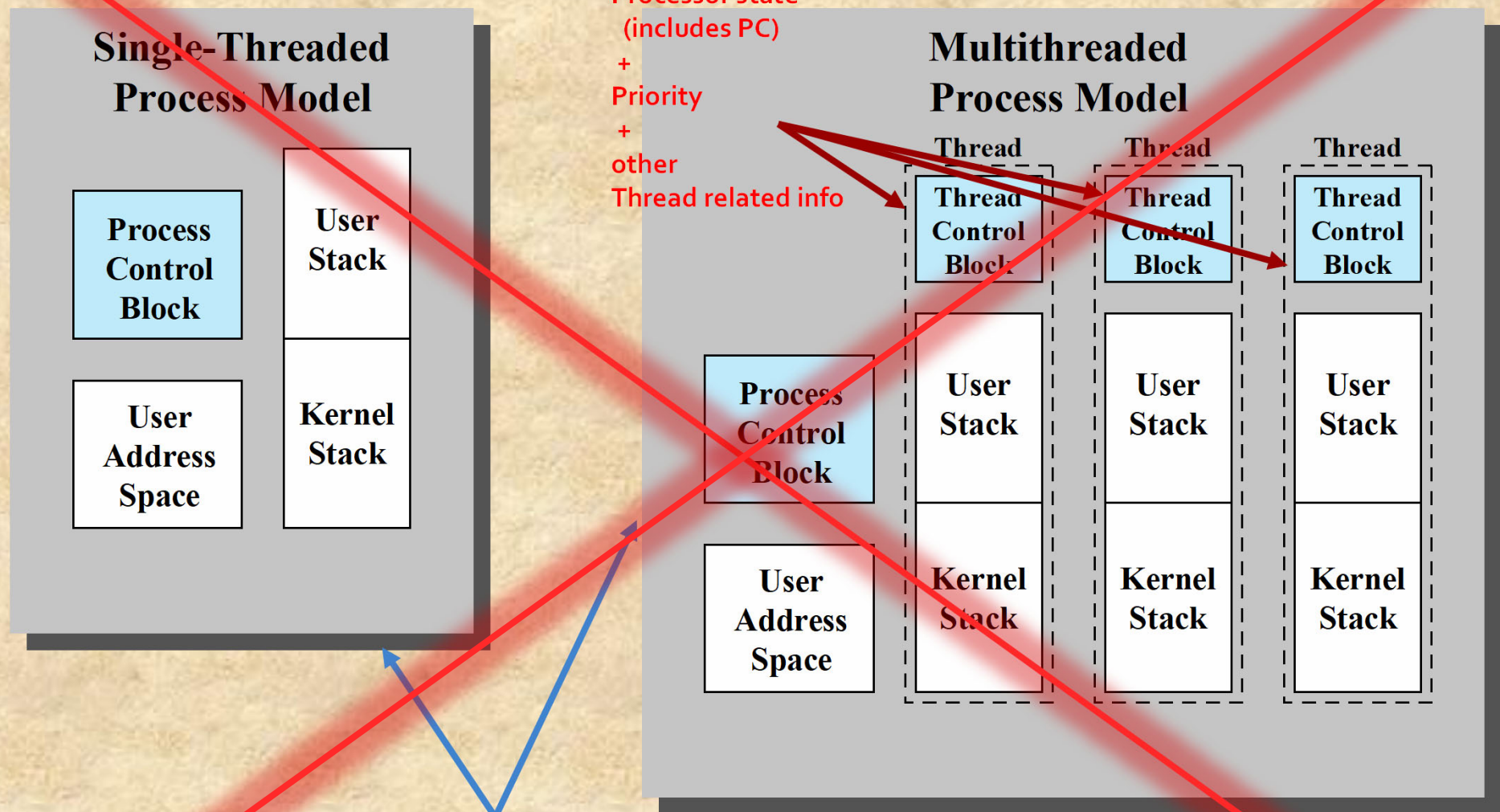
12. **User Stack (Process)**
14. **Kernel Stack (Process)**

A thread that belongs to process 4 has

4. **TCB (thread ID)**
5. **TCB (Processor state i.e. register file)**
6. **TCB (thread control)**

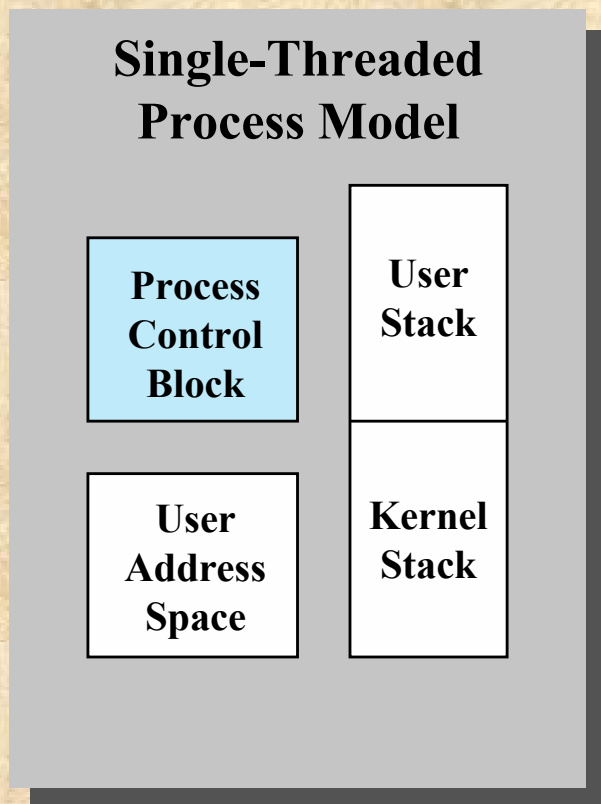
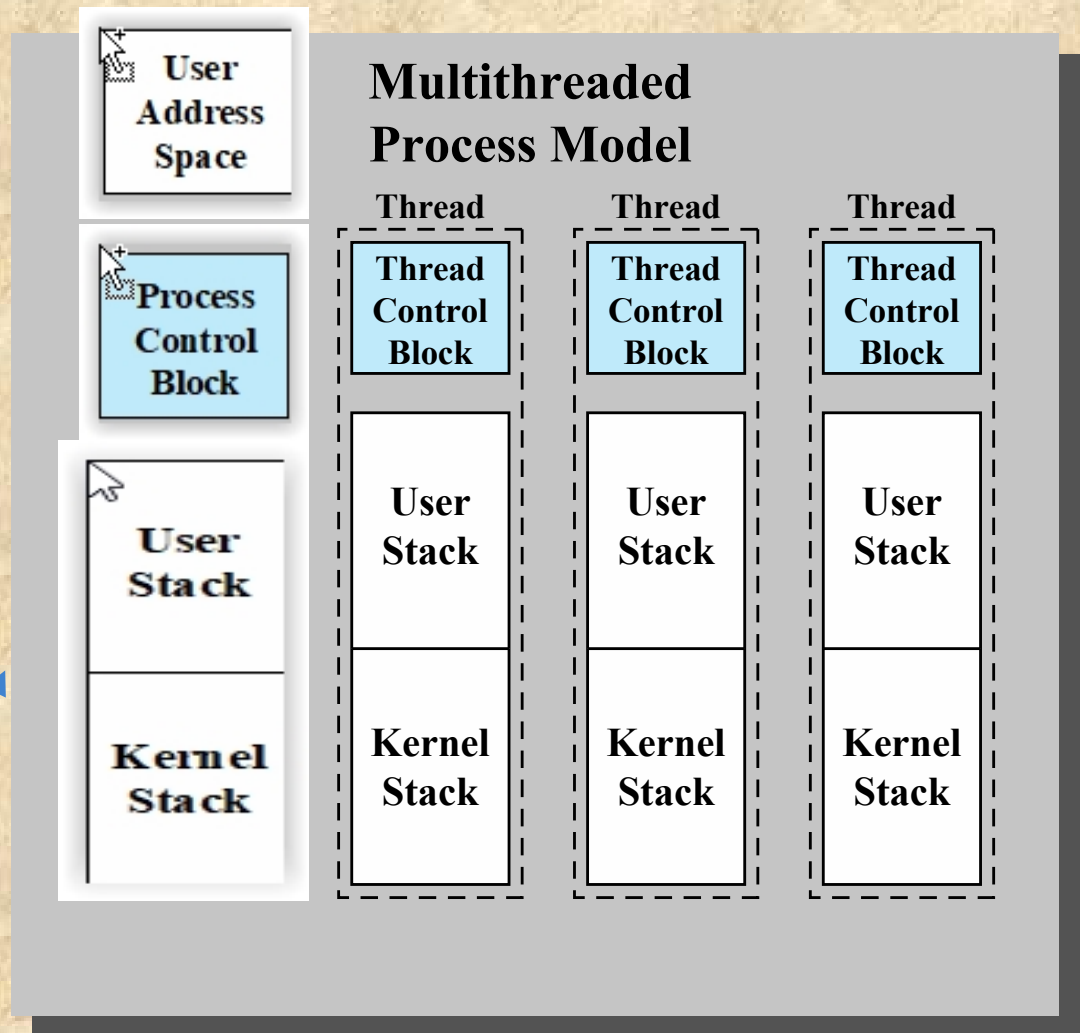
13. **User Stack (Thread)**
15. **Kernel Stack (Thread)**

Thread has (some, controlled by kernel)
access to 1,3 and FULL access to 7-11.



Question (Tricky): How many program counters (PC)?

Figure 4.2 Single Threaded and Multithreaded Process Models



Maybe

Figure 4.2 Single Threaded and Multithreaded Process Models



PCB: Kernel knows about process (PCB, PL, FT, MT, DT)

TCB: Kernel might or might not know about the thread (KLT vs ILT)

Process (Fig. 4.2 of textbook not Fig 4.1) **creates 3 (three) threads.**
Each thread has a TCB, process has a PCB.
In Linux the TCB or PCB uses part or all of task_struct

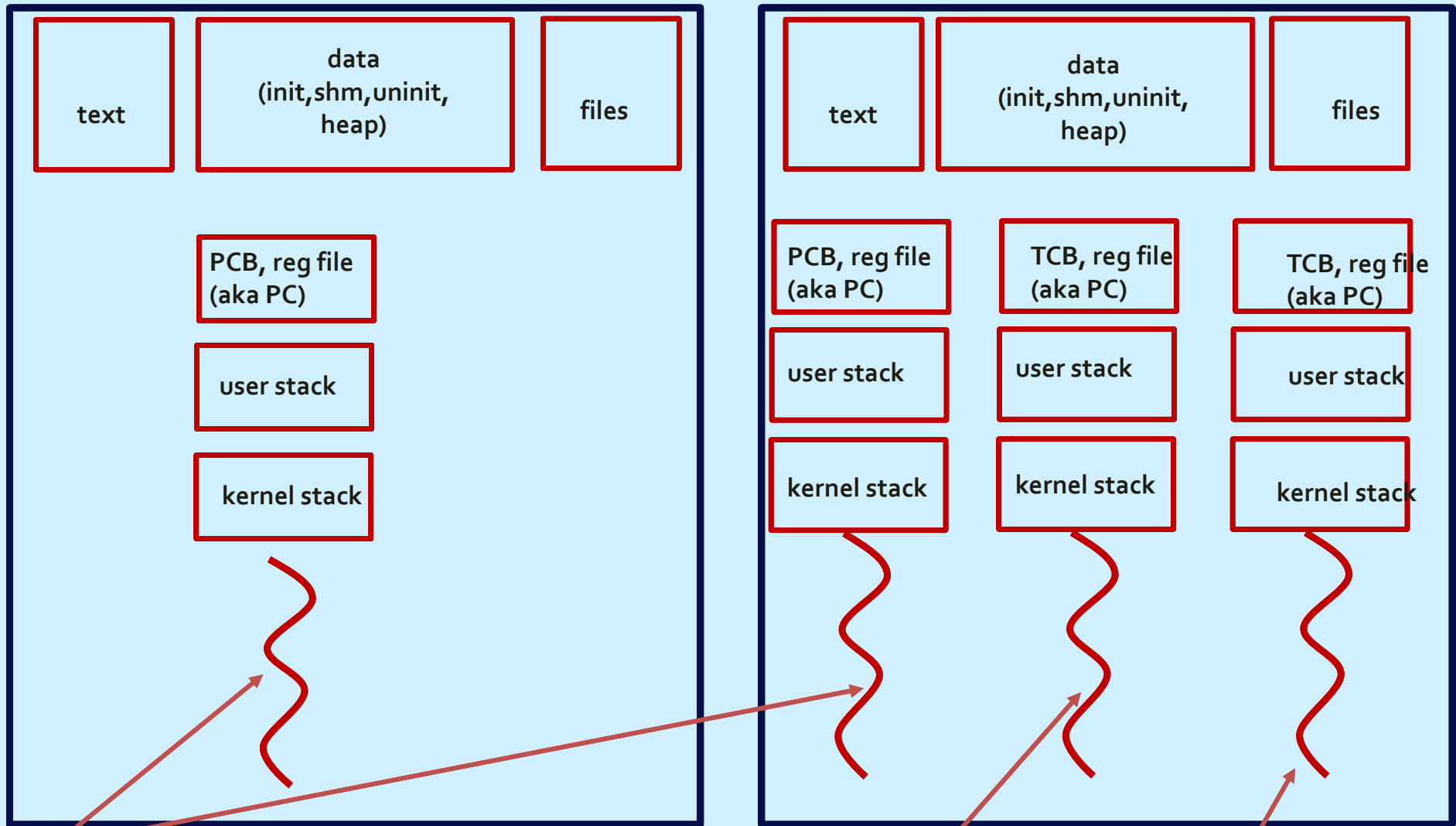
After successful creation of threads there are
 $3+1=4$ threads of execution i.e.

4 separate processOR states ,
4 separate stacks

but one text, init, uninit(aka bss), heap, shmem

THERE ARE 4 THREADS of EXECUTION

Example

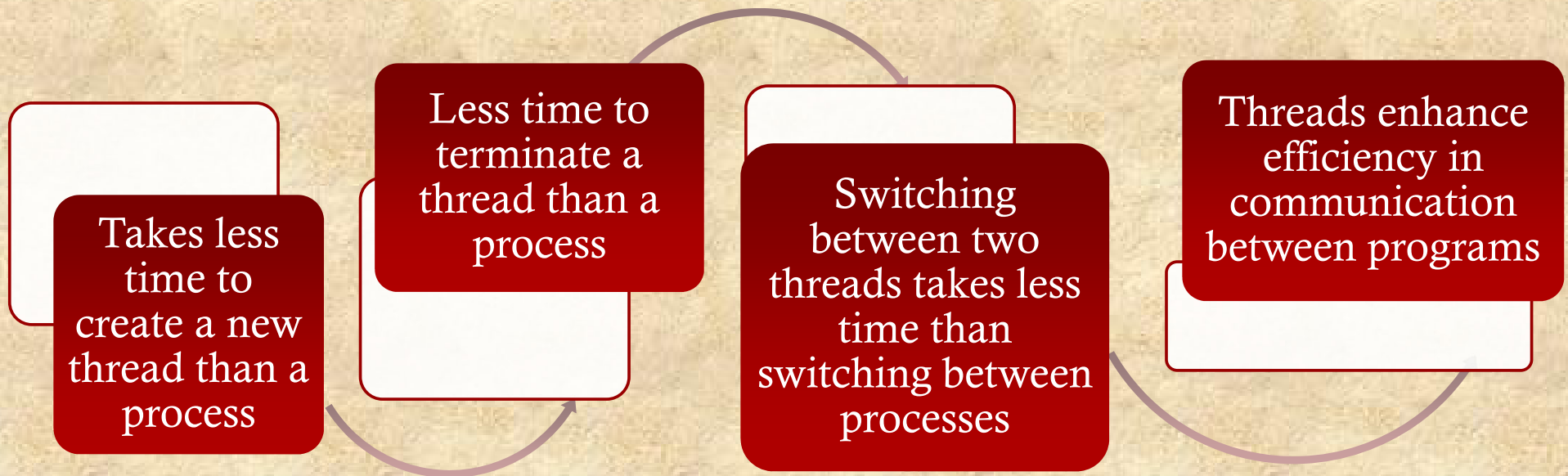


thread of execution of process

thread of execution of thread

thread of execution of thread

Key Benefits of Threads



Thread Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis

Most of the state information dealing with execution is maintained in thread-level data structures

- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

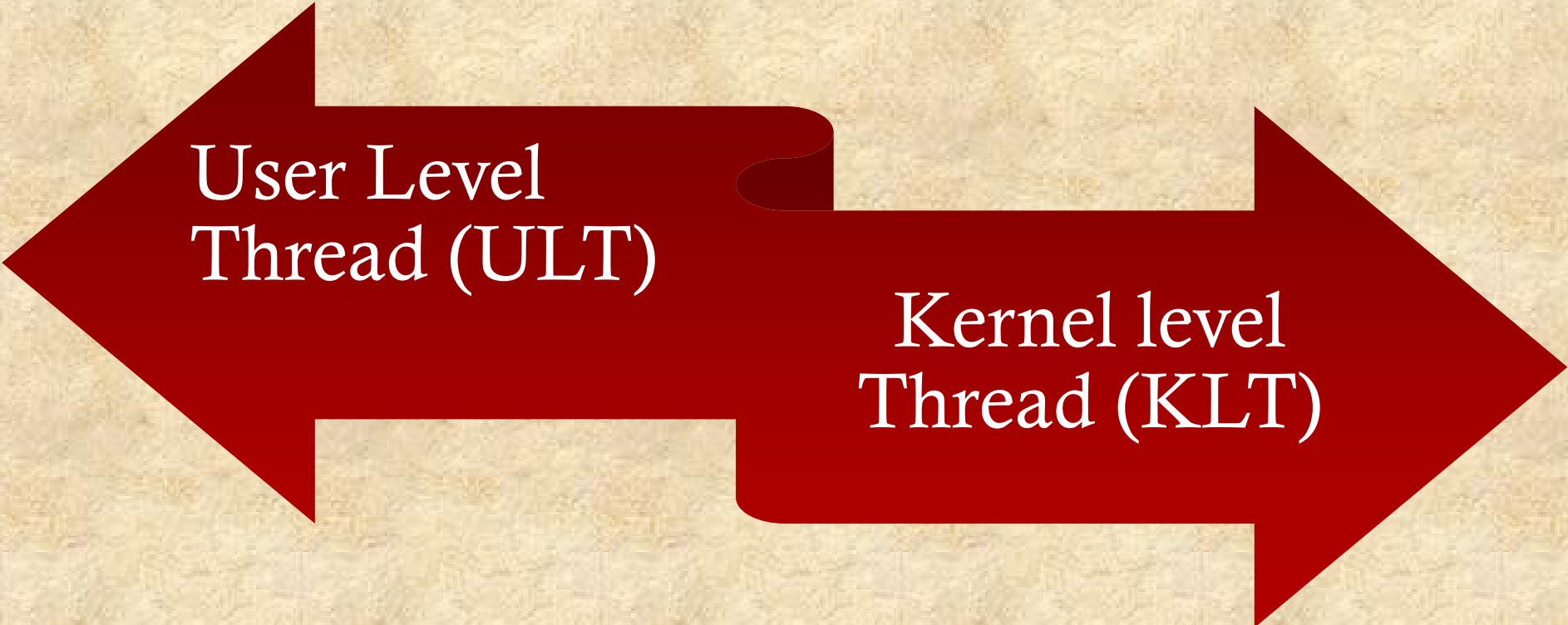
Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

Thread Synchronization

- It is necessary to synchronize the activities of the various threads
 - All threads of a process share the same address space and other resources
 - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads

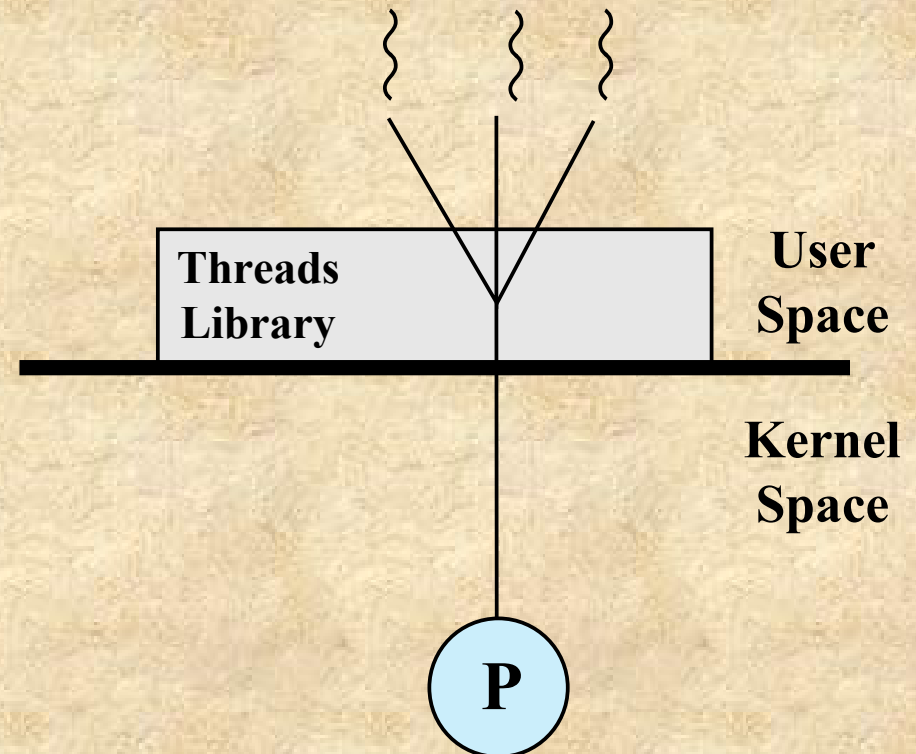


User Level
Thread (ULT)

Kernel level
Thread (KLT)

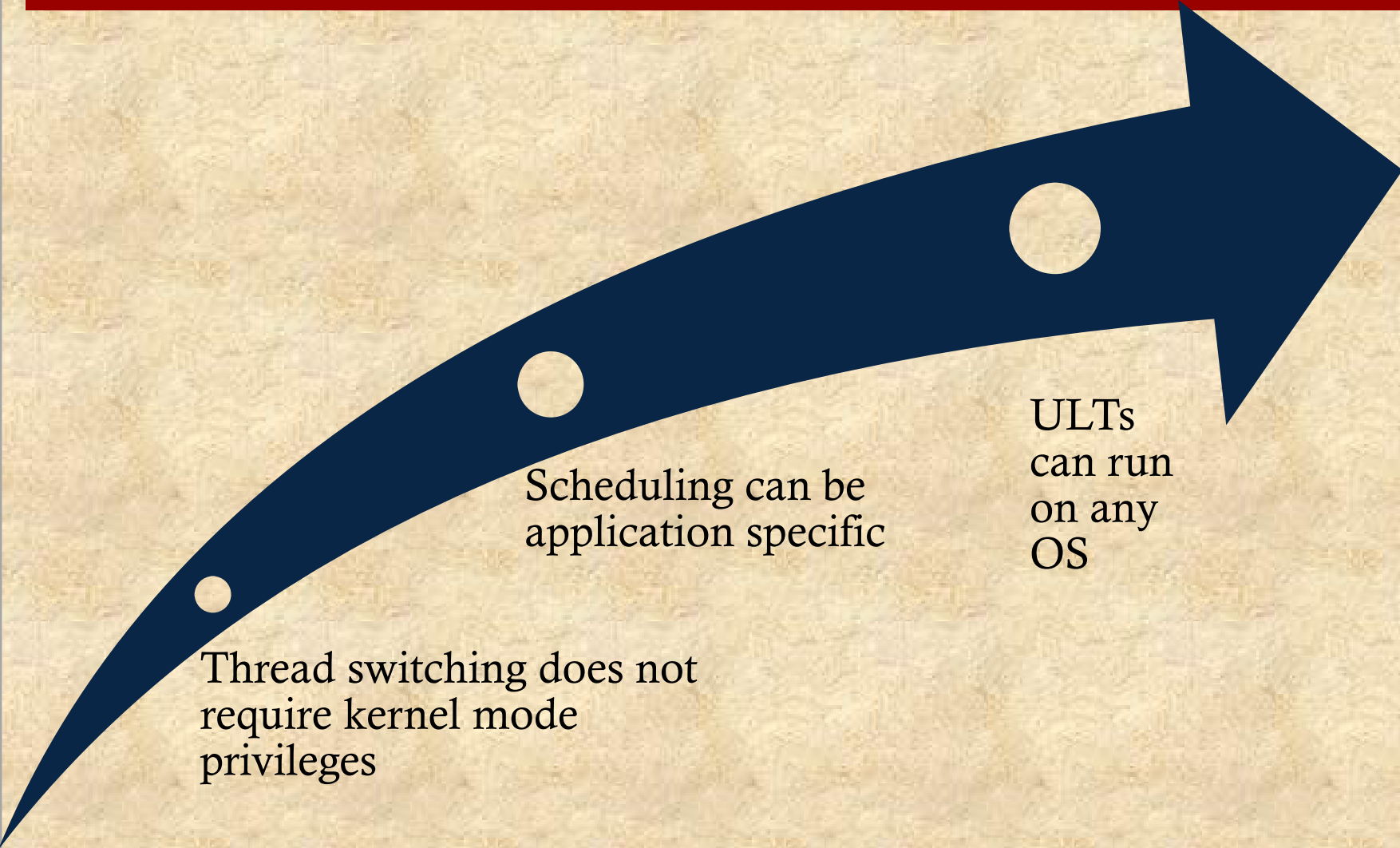
User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

Advantages of ULTs



Thread switching does not
require kernel mode
privileges

Scheduling can be
application specific

ULTs
can run
on any
OS

Disadvantages of ULTs

- In a typical OS many system calls are blocking
 - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
 - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Thread blocking



If a thread **BLOCKS**(eg I/O waiting) Process and its other threads all **BLOCK**

User-level threads that behave like this reside on a system not well-designed to be thread-ready i.e. it is not thread-safe. Solutions:

1. **Thread-ready [jacketing]**: turn **BLOCKing** system calls into non-blocking.
2. **Use multiprocessing** not multithreading (i.e. do not use threads).

If 1 or 2 is too radical :

Windows' approach: **kernel-level threads(KLT)** instead of **ULT**

- if a thread X of process Y gets blocked on a CPU,
- another thread Z of process Y gets dispatched on that CPU's quantum

Overcoming ULT Disadvantages

Jacketing

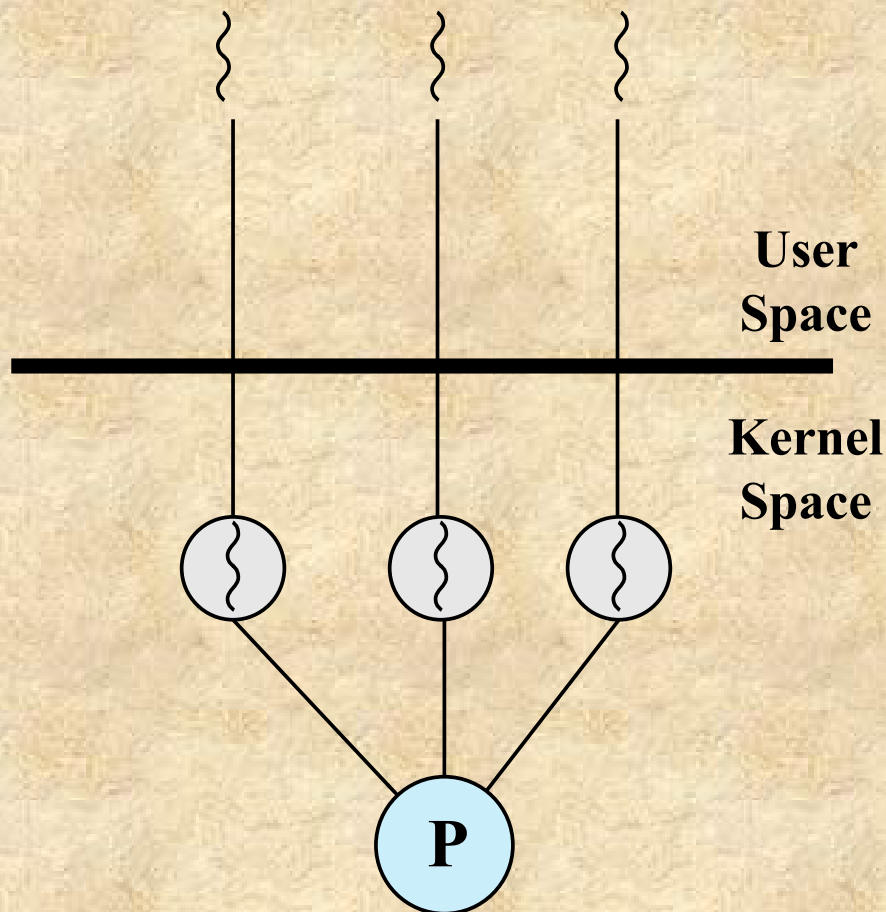
- Purpose is to convert a blocking system call into a non-blocking system call



Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
 - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

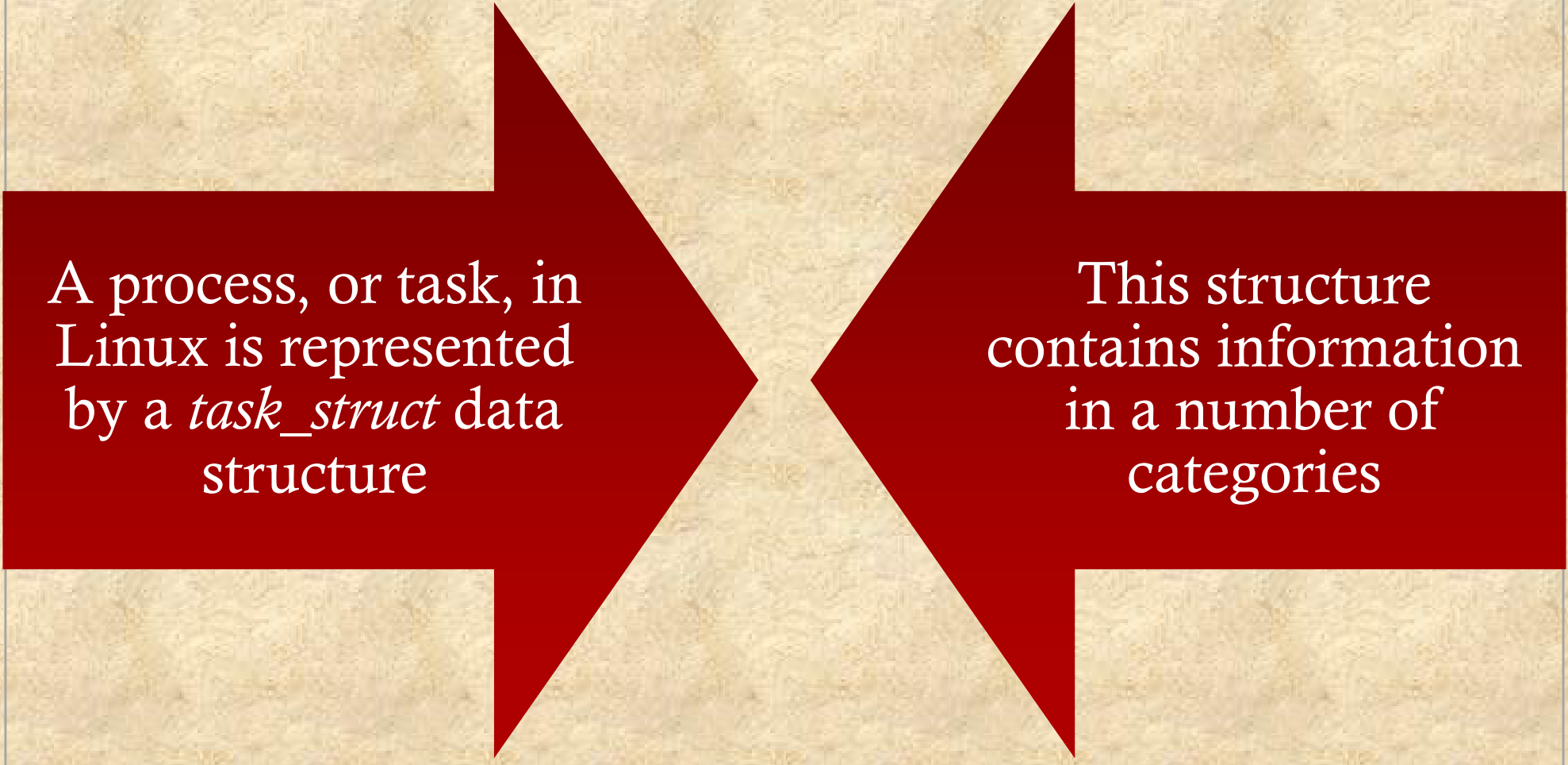
Disadvantage of KLTs

- ❏ The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μ s)

Linux Tasks



A process, or task, in Linux is represented by a *task_struct* data structure

This structure contains information in a number of categories



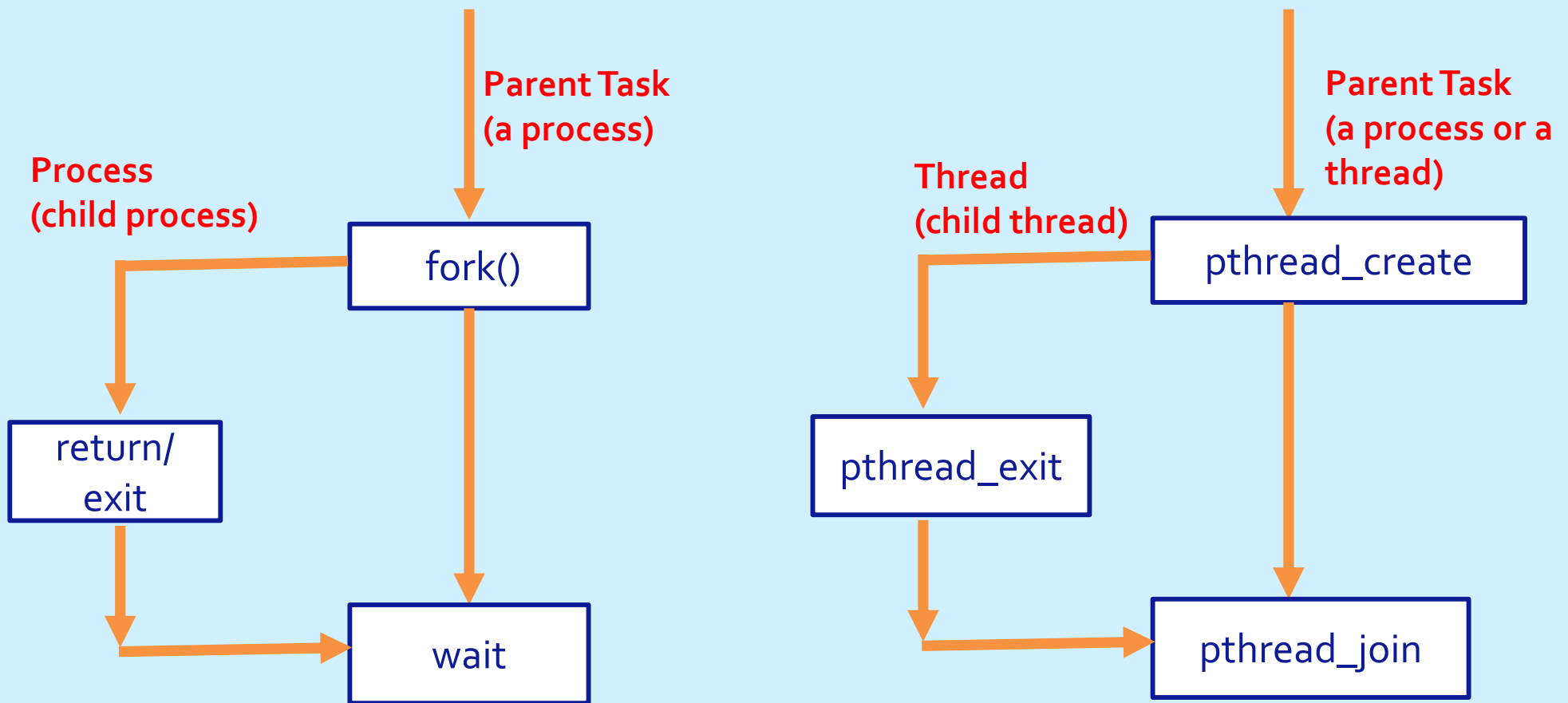
Practice 1

- 1.1. Can a thread be individually swapped? Say a process has five threads, and we want to swap one of the five. Can it be done?
- 1.2. When a process gets swapped what happens to its (five in the example above) threads?
- 1.3. When a process terminates, what happens to its threads?
- 1.4. Same as above if process suspends?
- 1.5. With reference to Slide 14, how many threads of execution are there in Fig 4.2 on the right box?
- 1.6. What is Jacketing?
- 1.7. Is Jacketing define in the context of Kernel Level threads? Explain.
- 1.8. When a thread gets blocked in Windows, what happens?
- 1.9. Rank KLT, ULT and Processes, Fastest (leftmost) to Slowest (rightmost)
- 1.10. With reference to question 4.1.5 and Fig 4.2 (right) how many threads were created?



Module 2

Process vs Thread



Parent task, in theory can be a process or a thread

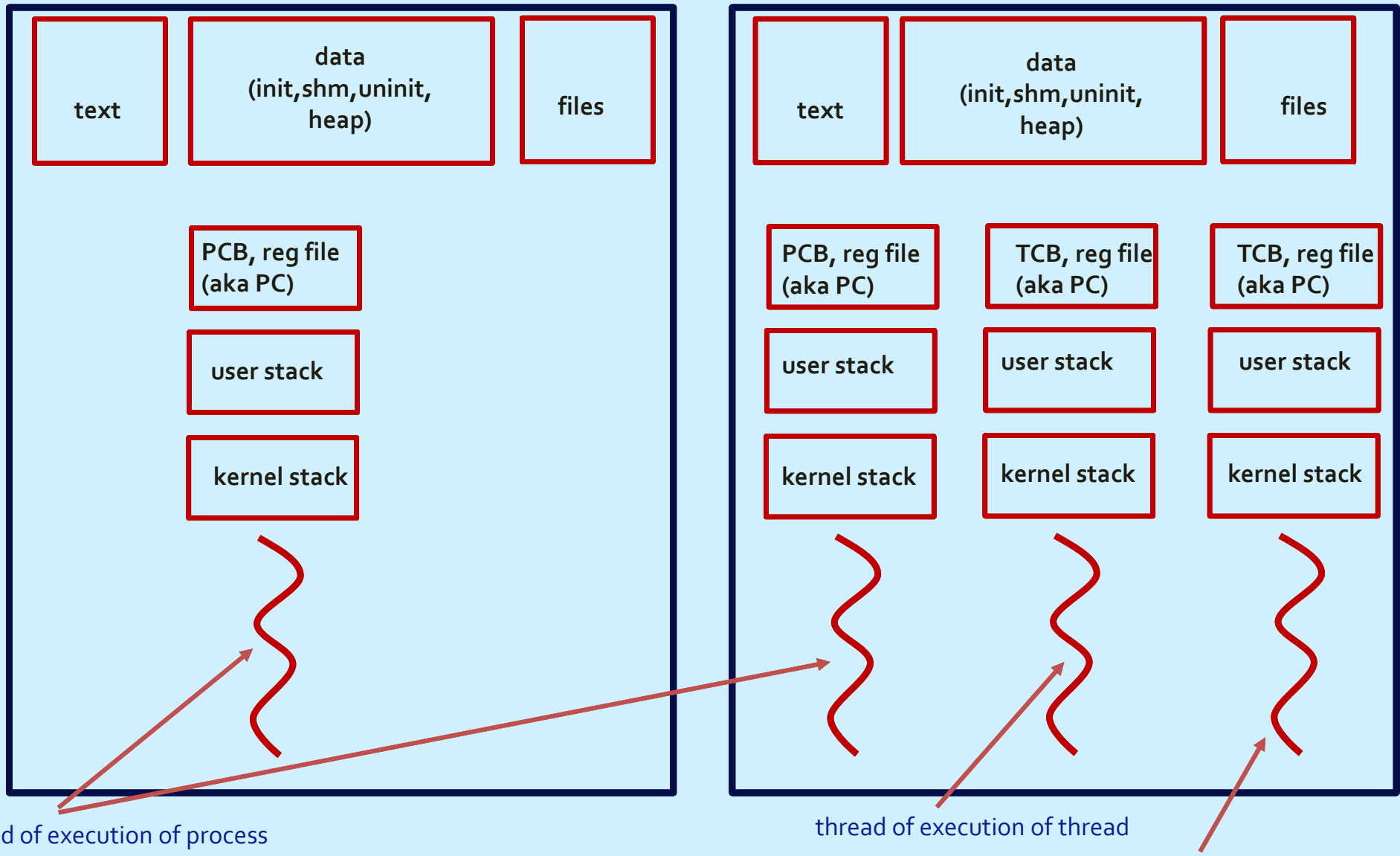
If parent is a process

If child is a process, the child knows who its parent is (getppid())

If child is a thread, the child knows who its parent is (getpid())

If parent is a thread

if child is a thread, the child CANNOT KNOW who its parent is (by default)





Header files in violet will be removed
from later Powerpoint slides

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef NTHREADS
#define NTHREADS 2
#endif
/* Read ReadmeS4.txt first */
void *StartT(void *ptr ) {
    printf("Hello from Thread  mainID=%2s of %2d ; PTID=%ld\n",
        (char*)ptr, NTHREADS, pthread_self());
    pthread_exit(NULL);
}
int main(void) { /* Process starts execution with main function */
    pthread_t thrd[NTHREADS];
    char *msg[2]={ "0", "1" }; /* if NTHREADS is not 2, modify this */
    int trc;
    long t;
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_create(&thrd[t], NULL, StartT, (void*) msg[t]);
        if (trc != 0) {
            printf("CREATE: t=%ld trc=%d\n",t,trc);
        }
    }
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_join(thrd[t],NULL);
        if (trc != 0) {
            printf("JOIN:    t=%ld trc=%d\n",t,trc);
        }
    }
    exit(0);
}
```

Imagine a
sleep(10)
before the
pthread_exit

thread01.c running



Some Notes

- pthread_create:** creates a thread;
first argument is control struct,
second argument TBD,
third arg is function to execute;
fourth argument is data to pass (a 0 or 1)
- pthread_join:** If the (parent) process does not wait for the threads to conclude and join it again, it (parent) might terminate. **Then the threads get killed**. If a process terminates, all of its threads terminate. If a process gets blocked all of its threads get blocked (Windows has a work around on this though).
- pthread_exit:** execution of the threads code; this does not mean that the thread itself is terminated; a pthread_join will let the parent process capture the thread to clean it up and terminate it gracefully later

NOTE THAT IN THE REMAINING EXAMPLES WE REMOVE FROM THE POWERPOINTS THE HEADER FILES

```
% gcc thread01.c -pthread -o t01
                        or alternatively,
% gcc thread01.c -lpthread -o t01
% ./t01
Hello from Thread  mainID= 1 of  2 ; PTID=140695613859584
Hello from Thread  mainID= 0 of  2 ; PTID=140695622252288
%
%
% gcc thread01.c -lpthread -D NTHREADS=1 -o thread01
% ./thread01
Hello from Thread  mainID= 0 of  1 ; PTID=140287936616192
```



Question 1. How many threads of execution are there in threado1.c immediately after the first for loop exits

to rephrase

Question 1 rephrased. How many sets of PC (program counters) coexist because of threado1.c (and its thread_create)?

threads

vs

#threads of execution
(aka sets of PCs and associated
processors states/ stack etc)



```
#include <pthread.h>
... /* lines deleted for brevity : see tar file for complete source code */
#ifndef NTHREADS
#define NTHREADS 2
#endif
void *StartT(void *ptr ) {
    printf("Hello from Thread  mainID=%2s of %2d ; PTID=%ld PID=%d TID=%d\n",
        (char*) ptr, NTHREADS, pthread_self(),getpid(),gettid());
    pthread_exit(NULL);
}
int main(void) {
    pthread_t thrd[NTHREADS];
    char *msg[2]={ "0", "1" };
    int trc;
    long t;
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_create(&thrd[t], NULL, StartT, (void*) msg[t]);
        if (trc != 0) {
            printf("CREATE: t=%ld trc=%d\n",t,trc);
        }
    }
    printf("Hello from the PROCESS  itself ...          "
        "          PID=%d TID=%d\n",getpid(),gettid());
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_join(thrd[t],NULL);
        if (trc != 0) {
            printf("JOIN:    t=%ld trc=%d\n",t,trc);
        }
    }
    exit(0);
}
```



One process creates two threads

processes is

threads is

threads of execution are:

Proof by example

```
% gcc thread02.c -lpthread -o t02
```

```
% ./t02
```

```
Hello from the PROCESS itself ...
```

```
PID=21149 TID=21149
```

```
Hello from Thread mainID= 0 of 2 ; PTID=140536727488256 PID=21149 TID=21150
```

```
Hello from Thread mainID= 1 of 2 ; PTID=140536719095552 PID=21149 TID=21151
```

thread03.c



```

/* all header lines erased for brevity */
void *StartT(void *ptr ) {
    printf("Hello from Thread  mainID=%2s of %2d ; PTID=%ld \n", (char*)ptr, NTHREADS, pthread_self());
    pthread_exit(NULL);
}

int main(void){
    pthread_t      thrd[NTHREADS];
    pthread_attr_t attr;
    size_t         stack_size;
    char           *msg[2]={ "0", "1" };
    int trc;
    long t;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_attr_getstacksize(&attr, &stack_size);
    printf("User Stack Size is      = %d \n", (int)stack_size);
    pthread_attr_setstacksize(&attr, (size_t) 2*stack_size);
    pthread_attr_getstacksize(&attr, &stack_size);
    printf("User Stack Size set to  = %d \n", (int)stack_size);
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_create(&thrd[t], &attr, StartT, (void*) msg[t]);
        if (trc != 0) printf("CREATE: t=%ld trc=%d\n", t, trc); /* some braces saved*/
    }
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_join(thrd[t], NULL);
        if (trc != 0) {
            printf("JOIN:    t=%ld trc=%d\n", t, trc);
        }
    }
    pthread_attr_destroy(&attr);
    exit(0);
}

```

thread03.c running



```
% ./t03
User Stack Size is      = 8388608
User Stack Size set to  = 16777216
Hello from Thread  mainID= 1 of  2 ; PTID=140342992238336
Hello from Thread  mainID= 0 of  2 ; PTID=140343009019648
```

threado4.c



```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#ifndef NTHREADS
#define NTHREADS 2
#endif

pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;
int globalid = -1;          /* this is an init variable stored in INIT space */
void *StartT(void *ptr) {
    int myid;                /* myid is stored in the USER STACK */
    pthread_mutex_lock( &mutx ); /* protect myid's access */
    globalid++; myid=globalid;
    printf("Hello from Thread  mainID=%2s , gid=%2d of %2d ; PTID=%ld \n",
           (char*) ptr, myid, NTHREADS, pthread_self());
    pthread_mutex_unlock( &mutx );
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thrd[NTHREADS];
    char *msg[2]={ "0", "1" };
    int trc;
    long t;
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_create(&thrd[t], NULL, StartT, (void*) msg[t]);
        if (trc != 0) printf("CREATE: t=%ld trc=%d\n",t,trc);
    }
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_join(thrd[t],NULL);
        if (trc != 0) printf("JOIN:  t=%ld trc=%d\n",t,trc);
    }
    pthread_mutex_destroy(&mutx);
    exit(0);
}

```



```
globalid++; myid=globalid;  
printf("Hello from Thread  
mainID=%2s , gid=%2d of %2d  
; PTID=%ld \n",  
      (char*) ptr, myid,  
      NTHREADS, pthread_self());
```

pthread_mutex_lock(&mutex)

```
globalid++; myid=globalid;  
printf("Hello from Thread  
mainID=%2s , gid=%2d of %2d  
; PTID=%ld \n",  
      (char*) ptr, myid,  
      NTHREADS, pthread_self());
```

pthread_mutex_unlock(&mutex)

threado4.c running



```
% ./t04
Hello from Thread  mainID= 0 , gid= 0 of  2 ; PTID=140003321071360
Hello from Thread  mainID= 1 , gid= 1 of  2 ; PTID=140003312678656
```

There are three IDs associated with a thread

- (a) the output of `pthread_self()` aka **PTID**,
- (b) the output of `msg` of `main` passed to `ptr` of `StartT` aka **mainID**
- (c) the global variable value of `globalid`, initialized to -1 and then incremented once by each process and stored into the local (USER STACK) variable `myid` and printed under the banner **gid** (for global ID)

thread05.c



```

/* header lines erased */
pthread_mutex_t  mtx = PTHREAD_MUTEX_INITIALIZER;

void *StartT(void *ptr ) {
    int myid;
    pthread_mutex_lock( &mtx ); /* protect myid's access. Does it need it? */
    myid = (int)*((int *) ptr);
    printf("Hello from Thread LmainID=%2d          of %2d ; PTID=%ld\n",
           myid, NTHREADS, pthread_self());
    pthread_mutex_unlock( &mtx );
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thrd[NTHREADS];
    int      tid [NTHREADS];
    int trc;
    long t;
    for(t=0; t < NTHREADS ; t++) tid[t]=t;
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_create(&thrd[t], NULL, StartT, (void*) &tid[t]);
        if (trc != 0) printf("CREATE: t=%ld trc=%d\n",t,trc);
    }
    for(t=0; t < NTHREADS ; t++) {
        trc = pthread_join(thrd[t],NULL);
        if (trc != 0) printf("JOIN:   t=%ld trc=%d\n",t,trc);
    }
    pthread_mutex_destroy(&mtx);
    exit(0);
}

```

thread05.c running



```
% gcc thread05.c -lpthread -D NTHREADS=10 -o t05
% ./t05
Hello from Thread LmainID= 0          of 10 ; PTID=139857456494336
Hello from Thread LmainID= 5          of 10 ; PTID=139857414530816
Hello from Thread LmainID= 7          of 10 ; PTID=139857397745408
Hello from Thread LmainID= 8          of 10 ; PTID=139857389352704
Hello from Thread LmainID= 9          of 10 ; PTID=139857380960000
Hello from Thread LmainID= 4          of 10 ; PTID=139857422923520
Hello from Thread LmainID= 2          of 10 ; PTID=139857439708928
Hello from Thread LmainID= 6          of 10 ; PTID=139857406138112
Hello from Thread LmainID= 1          of 10 ; PTID=139857448101632
Hello from Thread LmainID= 3          of 10 ; PTID=139857431316224

%% ./t05
Hello from Thread LmainID= 2          of 10 ; PTID=140421060101888
Hello from Thread LmainID= 0          of 10 ; PTID=140421076887296
Hello from Thread LmainID= 1          of 10 ; PTID=140421068494592
Hello from Thread LmainID= 3          of 10 ; PTID=140421051709184
Hello from Thread LmainID= 4          of 10 ; PTID=140421043316480
Hello from Thread LmainID= 8          of 10 ; PTID=140421009745664
Hello from Thread LmainID= 6          of 10 ; PTID=140421026531072
Hello from Thread LmainID= 7          of 10 ; PTID=140421018138368
Hello from Thread LmainID= 5          of 10 ; PTID=140421034923776
Hello from Thread LmainID= 9          of 10 ; PTID=140421001352960
```

Two executions of same exec file

threado6.c running



```
% gcc thread06.c -lpthread -D NTHREADS=10 -o t06
% ./t06
Hello from Thread DmainID= 0 , gid= 0 of 10 ; PTID=139763826288384
Hello from Thread DmainID= 4 , gid= 1 of 10 ; PTID=139763792717568
Hello from Thread DmainID= 2 , gid= 2 of 10 ; PTID=139763809502976
Hello from Thread DmainID= 1 , gid= 3 of 10 ; PTID=139763817895680
Hello from Thread DmainID= 3 , gid= 4 of 10 ; PTID=139763801110272
Hello from Thread DmainID= 5 , gid= 5 of 10 ; PTID=139763784324864
Hello from Thread DmainID= 6 , gid= 6 of 10 ; PTID=139763775932160
Hello from Thread DmainID= 7 , gid= 7 of 10 ; PTID=139763767539456
Hello from Thread DmainID= 9 , gid= 8 of 10 ; PTID=139763750754048
Hello from Thread DmainID= 8 , gid= 9 of 10 ; PTID=139763759146752
% ./t06
Hello from Thread DmainID= 0 , gid= 0 of 10 ; PTID=140040824014592
Hello from Thread DmainID= 6 , gid= 1 of 10 ; PTID=140040773658368
Hello from Thread DmainID= 7 , gid= 2 of 10 ; PTID=140040765265664
Hello from Thread DmainID= 3 , gid= 3 of 10 ; PTID=140040798836480
Hello from Thread DmainID= 9 , gid= 4 of 10 ; PTID=140040748480256
Hello from Thread DmainID= 4 , gid= 5 of 10 ; PTID=140040790443776
Hello from Thread DmainID= 1 , gid= 6 of 10 ; PTID=140040815621888
Hello from Thread DmainID= 5 , gid= 7 of 10 ; PTID=140040782051072
Hello from Thread DmainID= 2 , gid= 8 of 10 ; PTID=140040807229184
Hello from Thread DmainID= 8 , gid= 9 of 10 ; PTID=140040756872960
```

Two executions of same exec file. Did you expect anything else?



```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef NTHREADS
#define NTHREADS 10
#endif
pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;
#define N 10000
typedef struct {
    int  tnum; /* number of threads */
    int  tid; /* thread ID */
    int  tstat; /* state/status of thread */
    int  n; /* other data at will */
} tstruct;
tstruct tdata[NTHREADS];
double sum = (double)0;

int main(int argc, char **argv){
    pthread_t thrd[NTHREADS];
    pthread_attr_t attr;
    int trc, nn;
    long t;
    if (argc > 1) nn = atoi(argv[1]);
    else nn = N;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for(t=0; t < NTHREADS; t++) {
        tdata[t].tnum = NTHREADS;
        tdata[t].tid = t;
        tdata[t].tstat = -1;
        tdata[t].n = nn;
    }
    for(t=0; t < NTHREADS; t++) {
        trc = pthread_create(&thrd[t], &attr, StartT, (void*) &tdata[t]);
        if (trc != 0) {
            printf("CREATE: t=%ld trc=%d\n", t, trc);
        }
    }
    for(t=0; t < NTHREADS; t++) {
        trc = pthread_join(thrd[t], NULL);
        if (trc != 0) {
            printf("JOIN: t=%ld trc=%d\n", t, trc);
        }
    }
    printf("sum is %12.10f for n=%d \n", 4.0*sum, nn);
    pthread_mutex_destroy(&mutx);
    pthread_attr_destroy(&attr);
    exit(0);
}
```

```
void *StartT(void *ptr ) {
    int myid, mynt, n;
    int myn, left, right, i;
    double mysum = (double)0;
    double sgn = (double)1;
    pthread_mutex_lock( &mutx );
    myid = ((tstruct *) ptr)->tid;
    mynt = ((tstruct *) ptr)->tnum;
    n = ((tstruct *) ptr)->n;
    pthread_mutex_unlock( &mutx );
    myn = n / mynt;
    if (myid == (mynt-1)) /* last thread gets also the remainder */
        myn = n - (n/mynt)*(mynt-1);
    left = (n/mynt) * myid;
    right = left + myn;
    printf("Hello myid=%d of %d for n=%d\n", myid, mynt, n, myn);
    if ((left % 2) == 1)
        sgn = -sgn;
    for(i= left; i < right; ++i, sgn=-sgn){
        mysum += sgn/(2*i+1.0);
    }
    pthread_mutex_lock( &mutx );
    sum += mysum;
    pthread_mutex_unlock( &mutx );
    pthread_exit(NULL);
    return(NULL);
}

double compi(int n ) {
    double sum = (double)0;
    double one = (double)1;
    double sgn = (double)1;
    int i;
    for(i= 0; i < n; ++i, ++one, ++one, sgn=-sgn){
        sum += sgn/(one);
    }
    return(sum*4.0);
}
```

thread11.c running



Approximating pi using the sum $\text{Sum } (i=0 \text{ to } n) \ (-1)^i \cdot \frac{1}{(2i+1)} = \pi/4$

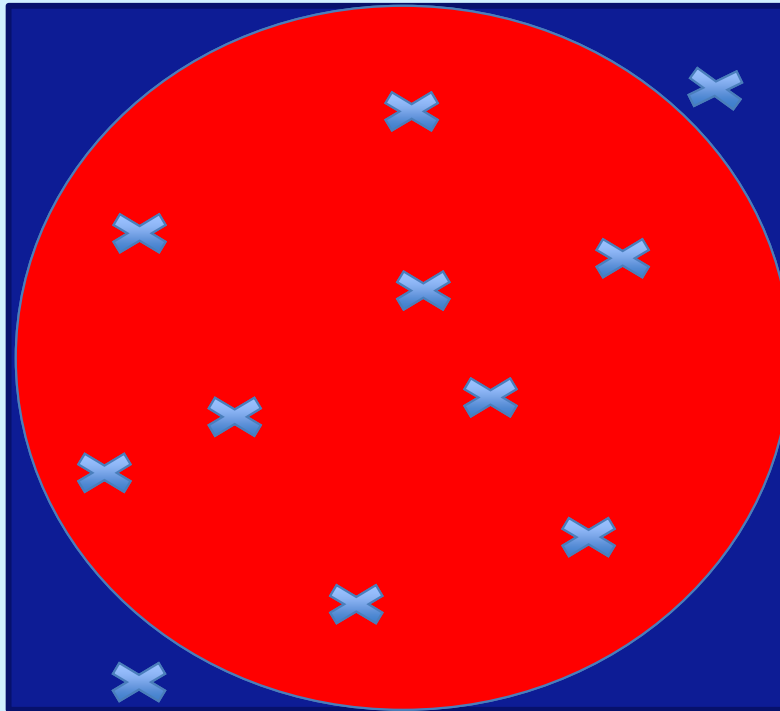
(a) Not splitting the sum, (b) Splitting into 2, (c) into 4 and into (d) 8
and using respectively multithreading with

(a) 1 thread, (b) 2threads, (c) 4threads, and (d) 8 threads respectively
Execfiles are t111, t112, t114, t118 respectively...

```
% time ./t111 1000000000
sum is 3.1415926526 for n=1000000000
real    0m2.224s
% time ./t112 1000000000
sum is 3.1415926526 for n=1000000000
real    0m1.138s

% time ./t114 1000000000
sum is 3.1415926526 for n=1000000000
real    0m0.609s

% time ./t118 1000000000
sum is 3.1415926526 for n=1000000000
real    0m0.466s
```



$$m / n \sim \pi / 4 \implies \pi \sim 4 * m / n$$

$$\text{Above: } 4 * 9 / 11 \sim 3.27$$

Square Side : $a=1$

Circle Radius: $r=0.5$

Square Area : $a \times a = 1$

Circle Area: $\pi \times r \times r = \pi / 4$

Circle Area/Square Area = $\pi / 4$

Experiment(n)

1. Throw n darts at random into the square area.
2. Record the number of darts m inside the red circle ($m \leq n$)
3. What are the chances that a dart hits inside the circle?
4. What is the ratio $4 * m / n$ observed in that experiment?
5. $4 * m / n$ is an approximation of π

thread12.c running



Approximating pi using the Monte carlo version **[jacketed]**

Execfiles are t121, t122, t124, t128 respectively... but t122 execution not shown for brevity

```
% time ./t121 1000000000
```

```
Hello myid=0 of 1 for n=1000000000 myn=1000000000
```

```
sum is 3.1415946880 for n=1000000000
```

```
real    0m12.084s
```

```
% time ./t124 1000000000
```

```
Hello myid=0 of 4 for n=1000000000 myn=250000000
```

```
Hello myid=1 of 4 for n=1000000000 myn=250000000
```

```
Hello myid=2 of 4 for n=1000000000 myn=250000000
```

```
Hello myid=3 of 4 for n=1000000000 myn=250000000
```

```
sum is 3.1415995720 for n=1000000000
```

```
real    0m3.288s
```

```
% time ./t128 1000000000
```

```
Hello myid=0 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=5 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=6 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=7 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=2 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=3 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=1 of 8 for n=1000000000 myn=125000000
```

```
Hello myid=4 of 8 for n=1000000000 myn=125000000
```

```
sum is 3.1416130640 for n=1000000000
```

```
real    0m2.249s
```



Approximating pi using the Monte carlo version **[non-jacketed]**
 Execfiles are t131, t132, t134, t138 respectively... but problem size is reduced by a factor of 100! Only t131 and t132 attempted

```
% time ./t131 1000000000
Hello myid=0 of 1 for n=1000000000 myn=1000000000
sum is 3.1415209680 for n=1000000000
real    0m27.619s
% time ./t132 1000000000
Hello myid=0 of 2 for n=1000000000 myn=500000000
Hello myid=1 of 2 for n=1000000000 myn=500000000
sum is 3.1418392680 for n=1000000000
real    6m11.887s
% time ./t134 1000000000
Hello myid=0 of 4 for n=1000000000 myn=250000000
Hello myid=1 of 4 for n=1000000000 myn=250000000
Hello myid=2 of 4 for n=1000000000 myn=250000000
Hello myid=3 of 4 for n=1000000000 myn=250000000
sum is 3.1417184160 for n=1000000000
real    3m33.760s
```

Observation: Times are dependent on the blocking time of a non-jacketed function call. They are universally worse than those of thread11.c.

```
real    6m11.887s      : real time is wall-clock time it takes to complete bash shell command
user    6m26.449s      : user time is CPU time (over all cores/processors) [user mode]
sys     5m54.155s      : sys time is CPU time (over all cores/processors) [system mode]
user + sys : CPU time used by process over all cores/processors
We are only reporting real time; rerun the experiments to obtain all timing results!
```



Practice 2

2.1. Grab the tar file associated with Subject 4 from the course web-page Section C (the area that is password protected), untar the tar file on an AFS machine, compile, run and experiment with the code available in the tar file. If you encounter any compilation issues, improvise and let me know!

NJIT

A white curved line that starts under the 'N', arches over the 'J' and 'I', and ends under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department