Processes and Threads

The present slides are mainly adapted from «*Operating Systems: Internals and Design Principles*» *6/E by* William Stallings (Chapter 4). Some materials are obtained from the POSIX threads Programming tutorial by Blaise Barney.

Sistemi di Calcolo (Part II – Spring 2018)

Instructors: Daniele Cono D'Elia, Riccardo Lazzeretti

Special thanks to: Leonardo Aniello, Roberto Baldoni

Roadmap

- Processes: fork(), wait()
- Threads: resource ownership and execution
- Symmetric multiprocessing (SMP)
- Case study:
  - PThreads

Role of Processes

- Most requirements that an OS must meet can be expressed w.r.t. processes:
  - Interleaved execution
  - Resource allocation and policies
  - User creation of processes and inter-process communication

Process Elements

- A process is comprised of:
  - Program code (possibly shared)
  - A set of data
  - A number of attributes describing the state of the process *during execution*

Process Elements

- While the process is running it has a number of elements including
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data
  - I/O status information
  - Accounting information

Process Control Block

- Contains the process elements
- Created and managed by the operating system
- Allows support for multiple processes

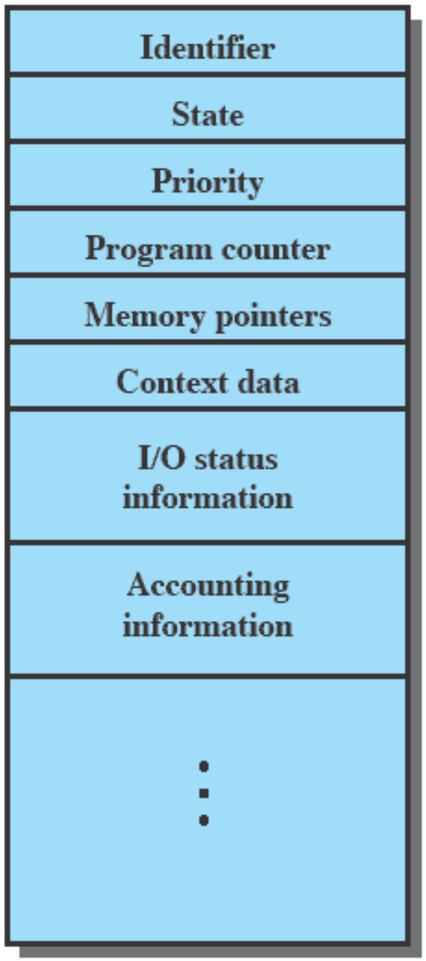


Figure 3.1 Simplified Process Control Block

Unix system calls
Creating new Processes

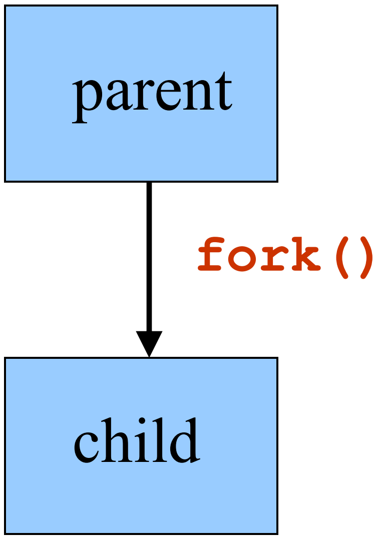
fork( ) - wait( ) - exit( )

Credits: Mirela Damian, Allan Gottlieb

Class notes: https://cs.nyu.edu/~gottlieb/courses/os202/class-notes.html

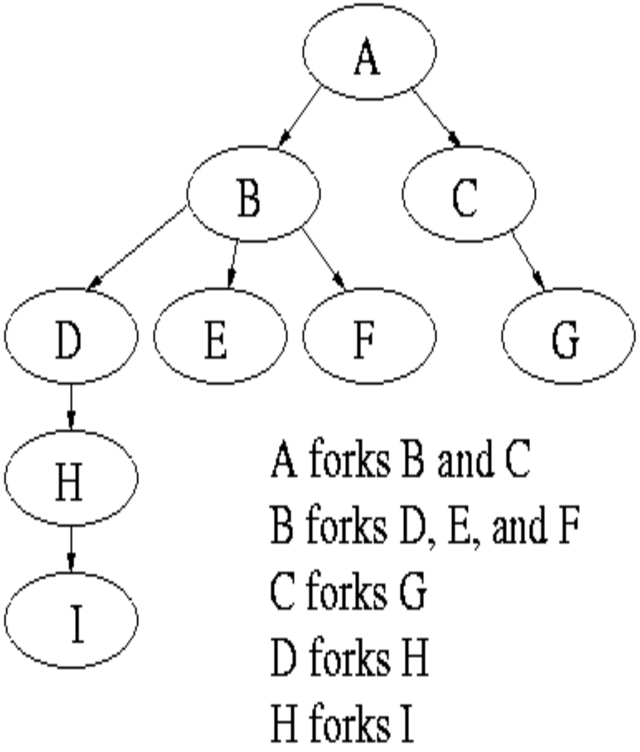
How To Create New Processes?

- Underlying mechanism
  - A process runs fork to create a child process
  - Parent and children execute concurrently
  - Child process is a duplicate of the parent process



Process Creation

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.



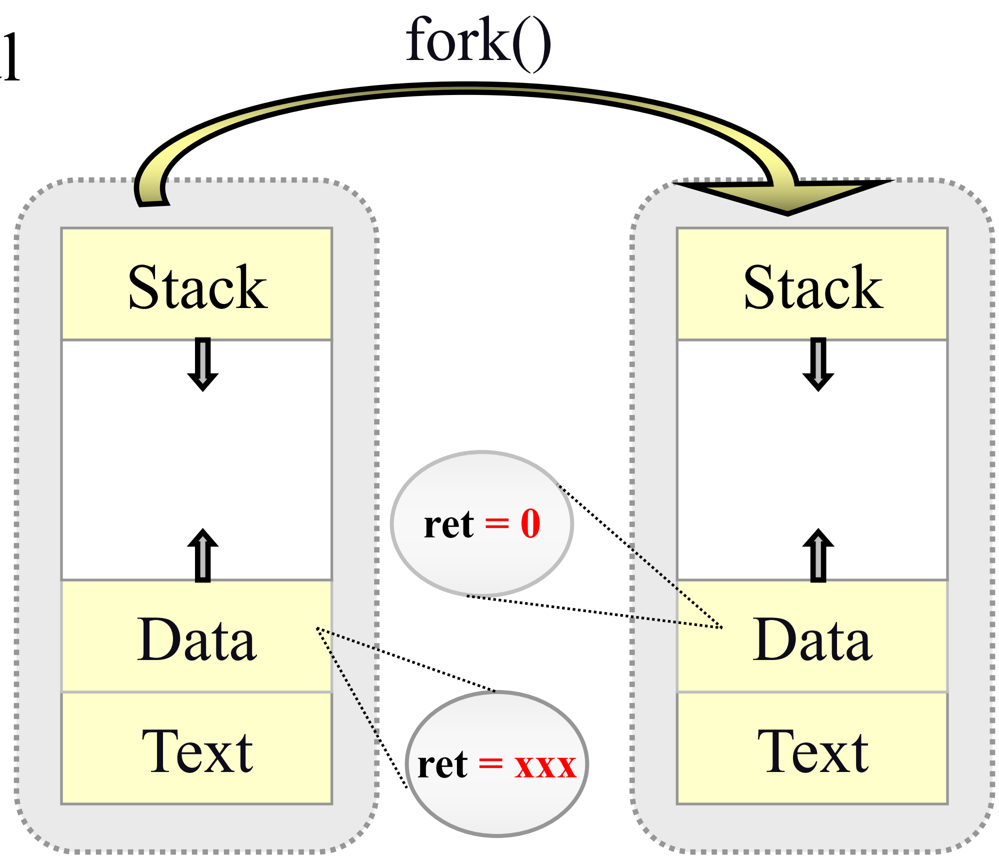
- A process can *choose* to wait for children to terminate. For example, if C issued a **wait**() system call, it would block until G finished.

Bootstrapping

- When a computer is switched on or reset, there must be an initial program that gets the system running
- This is the bootstrap program
  - Initialize CPU registers, device controllers, memory
  - Load the OS into memory
  - Start the OS running
- OS starts the first process (such as “init”)
- OS waits for some event to occur
  - Hardware interrupts or software interrupts (traps)

Fork System Call

- Current process split into 2 processes: parent, child
- Returns -1 if unsuccessful
- Returns 0 in the child
- Returns the child's identifier in the parent



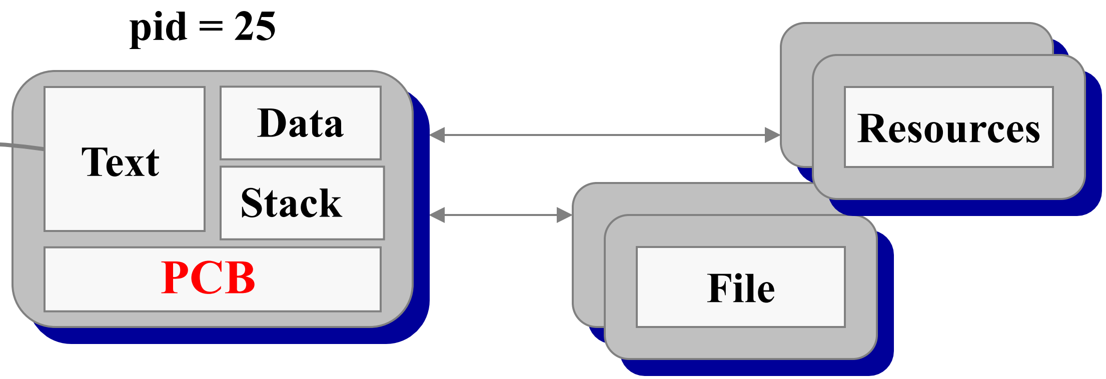
Fork System Call

- The child process inherits from parent
  - identical copy of memory
  - CPU registers
  - all files that have been opened by the parent

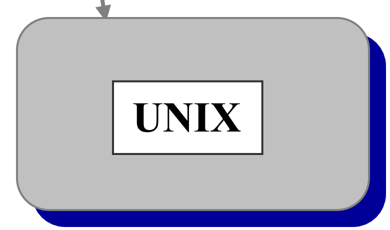
- Execution proceeds concurrently with the instruction following the fork system call

- The execution context (PCB) for the child process is a copy of the parent’s context at the time of the call

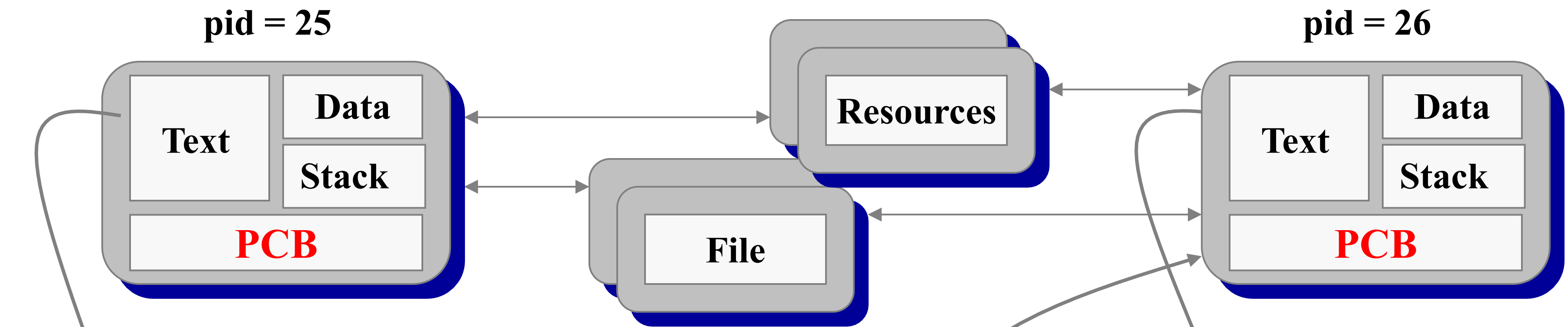
How fork Works (1)




```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

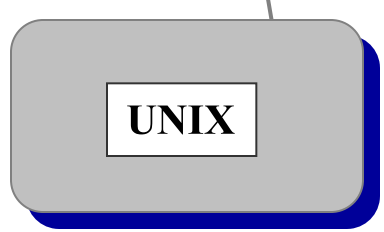


How fork Works (2)



ret = 26

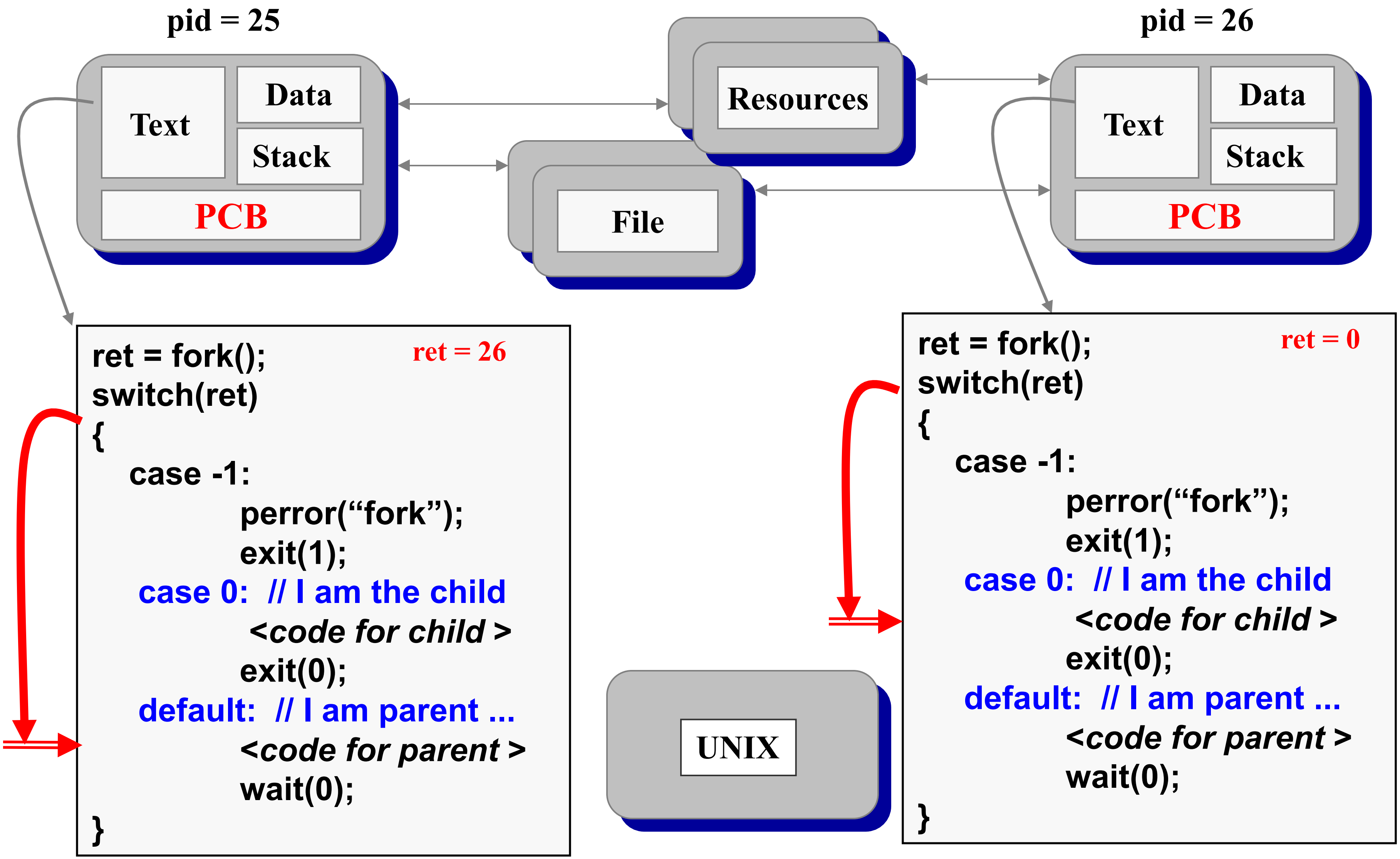
```
ret = fork();
switch(ret)
{
    case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
            <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```



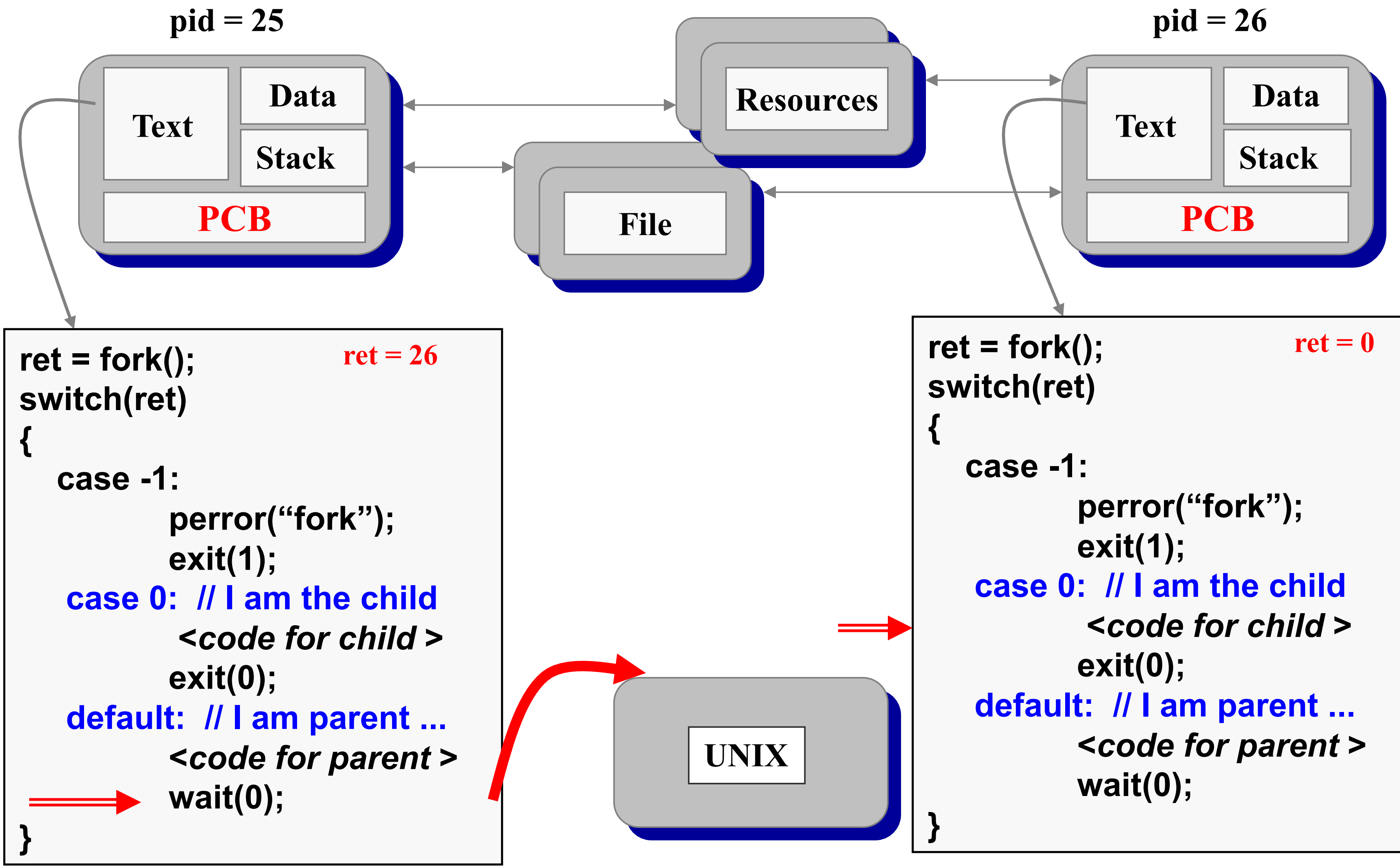
ret = 0

```
ret = fork();
switch(ret)
{
    case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
            <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

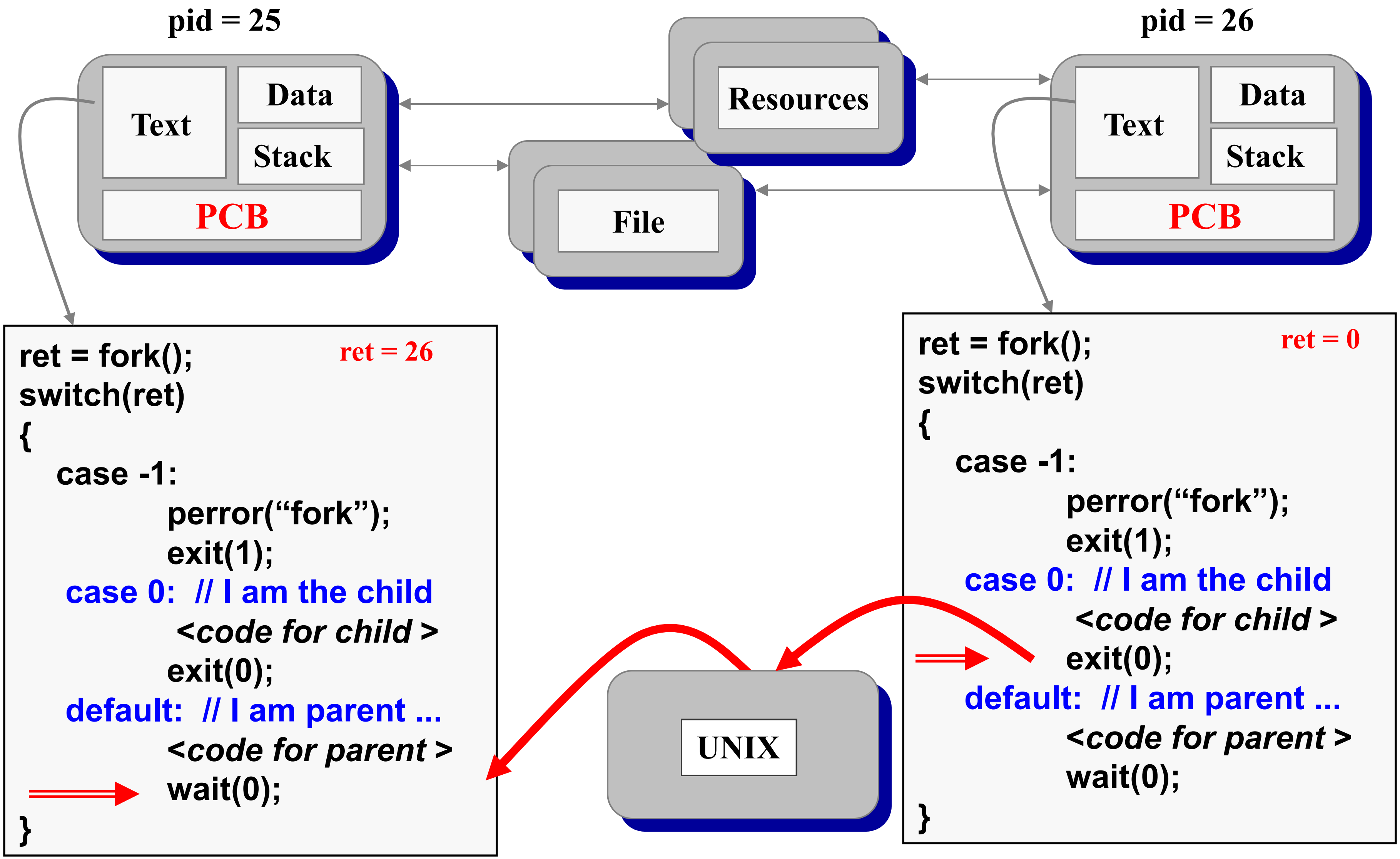
How fork Works (3)



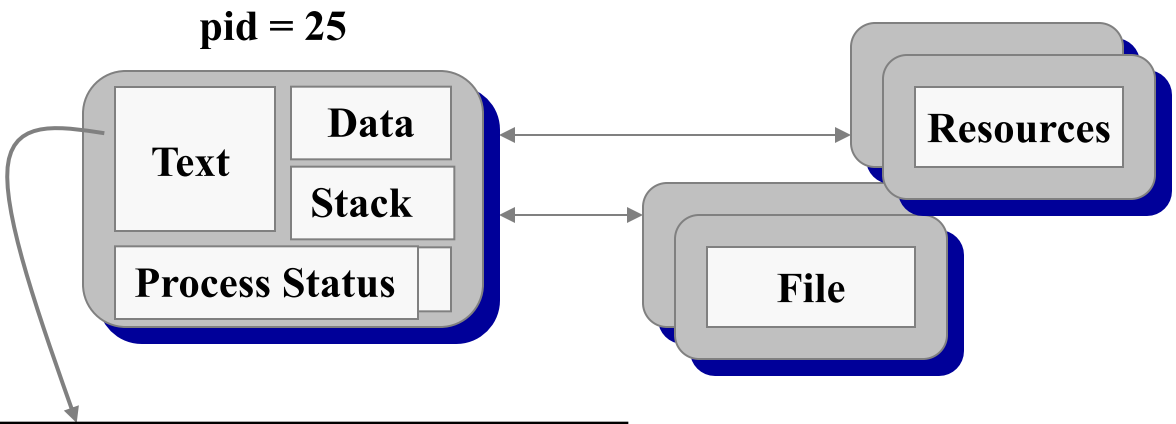
How fork Works (4)



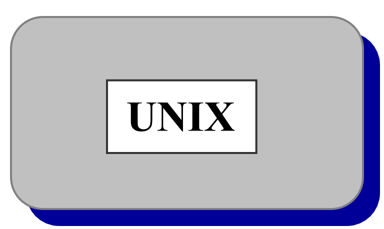
How fork Works (5)



How fork Works (6)




```
ret = fork();                    ret = 26
switch(ret)
{
   case -1:
              perror("fork");
              exit(1);
   case 0:  // I am the child
               <code for child >
              exit(0);
   default:  // I am parent ...
              <code for parent >
              wait(0);
              < … >
```



Orderly Termination: exit()

- To finish execution, a child may call **exit**(*number*)
- This system call:
  - Saves result = argument of exit
  - Closes all open files, connections
  - Deallocates memory
  - Checks if parent is alive
  - If parent is alive, holds the result value until the parent requests it (with wait); in this case, the child process does not really die, but it enters a zombie/defunct state
  - If parent is not alive, the child terminates (dies)

Waiting for the Child to Finish

- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: **wait()**
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: **waitpid()**
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Roadmap

- Processes: fork(), wait()
- Threads: resource ownership and execution
- Symmetric multiprocessing (SMP)
- Case study:
  - PThreads

Processes and Threads

- A process has two characteristics:
  - **Resource ownership** - 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

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.

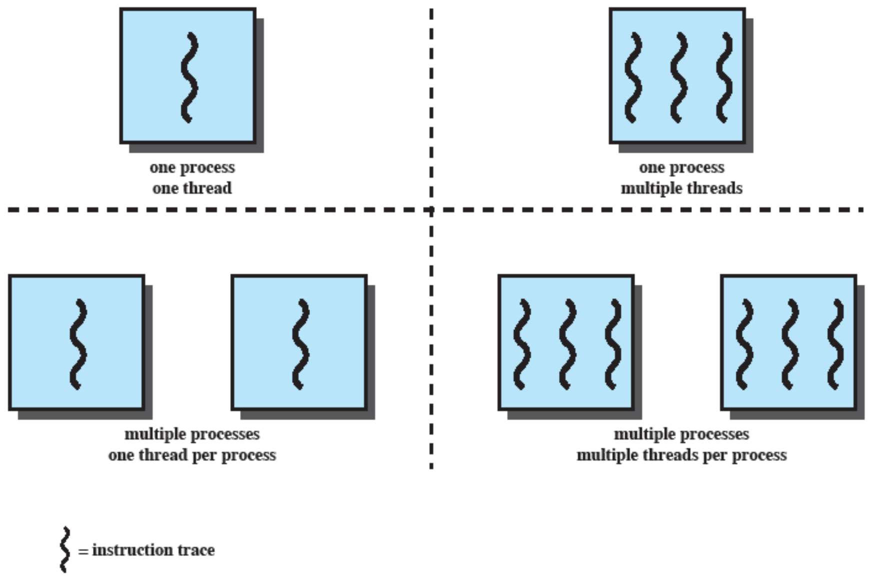


Figure 4.1 Threads and Processes [ANDE97]

Single Thread Approaches

- MS-DOS supports a single user process and a single thread
- Some UNIX support multiple user processes but only support one thread per process

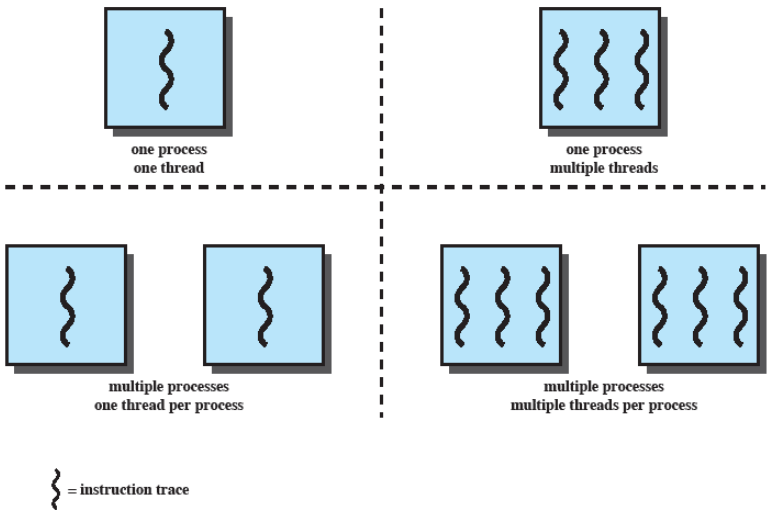


Figure 4.1 Threads and Processes [ANDE97]

Multithreading

- Often a Java run-time environment is a single process with multiple threads
- Multiple processes ***and*** threads are found in Windows, Solaris, and many modern versions of UNIX

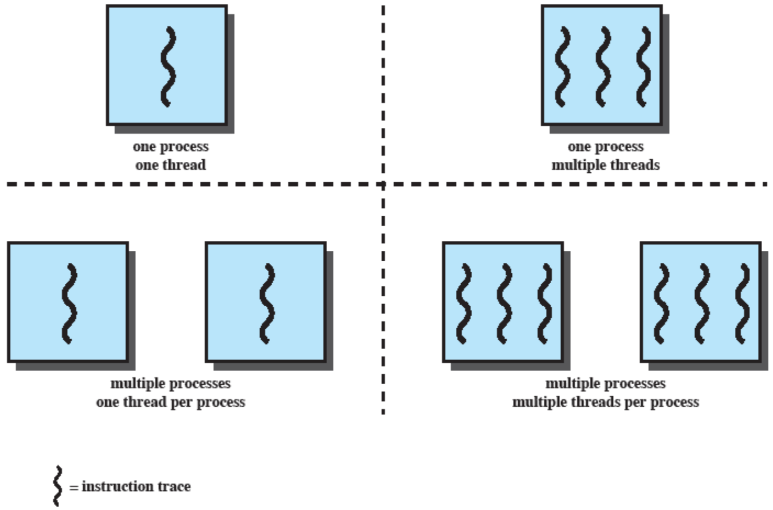


Figure 4.1 Threads and Processes [ANDE97]

Processes in Multithreaded OS

- A virtual address space which holds the process image
- Protected access to
  - Processors
  - Other processes
  - Files
  - I/O resources

One or More Threads in Process

- Each thread has
  - An execution state (running, ready, etc.)
  - 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 process (all threads of a process share this)

One view…

- One way to view a thread is as an independent program counter operating *within* a process

Threads vs. processes

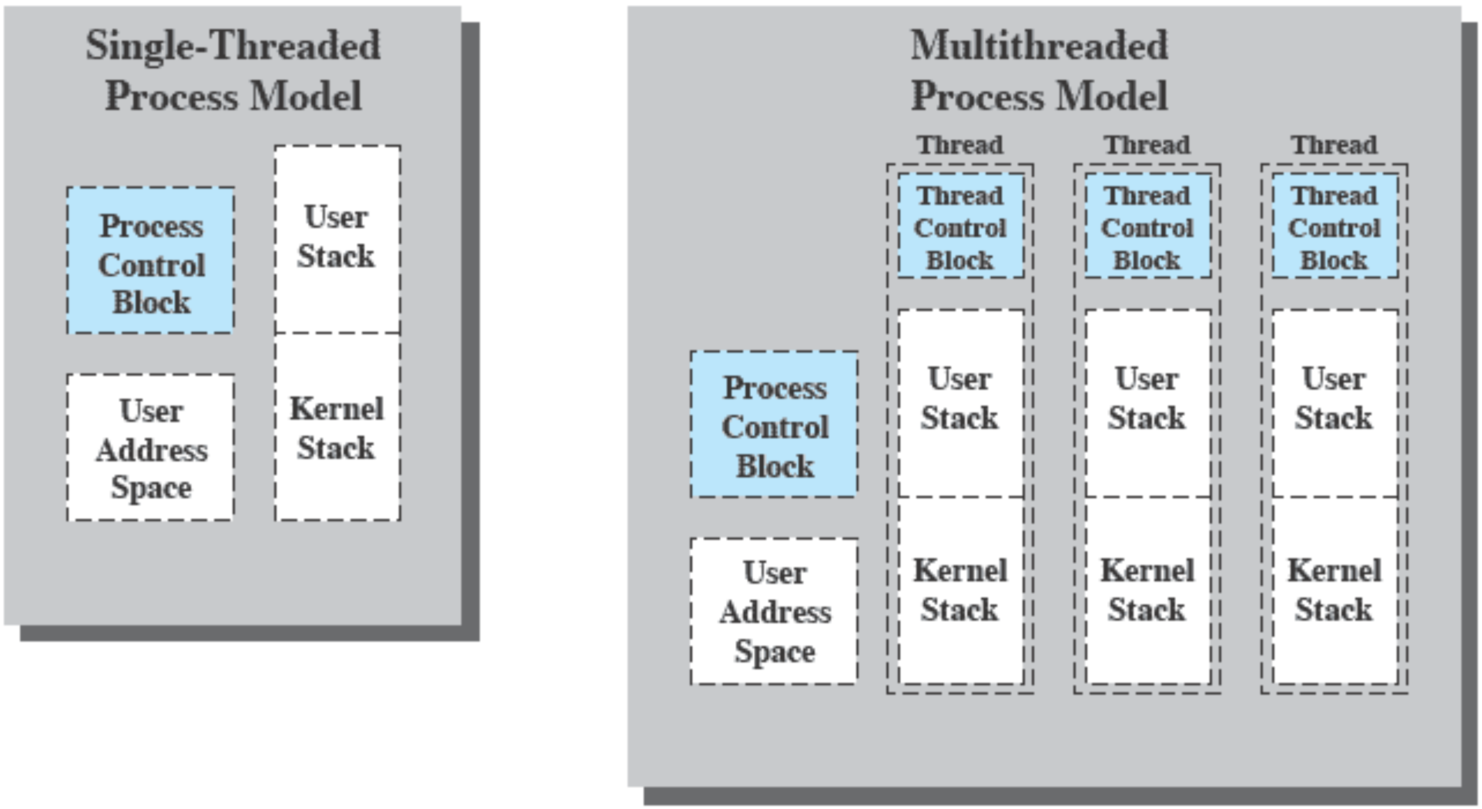
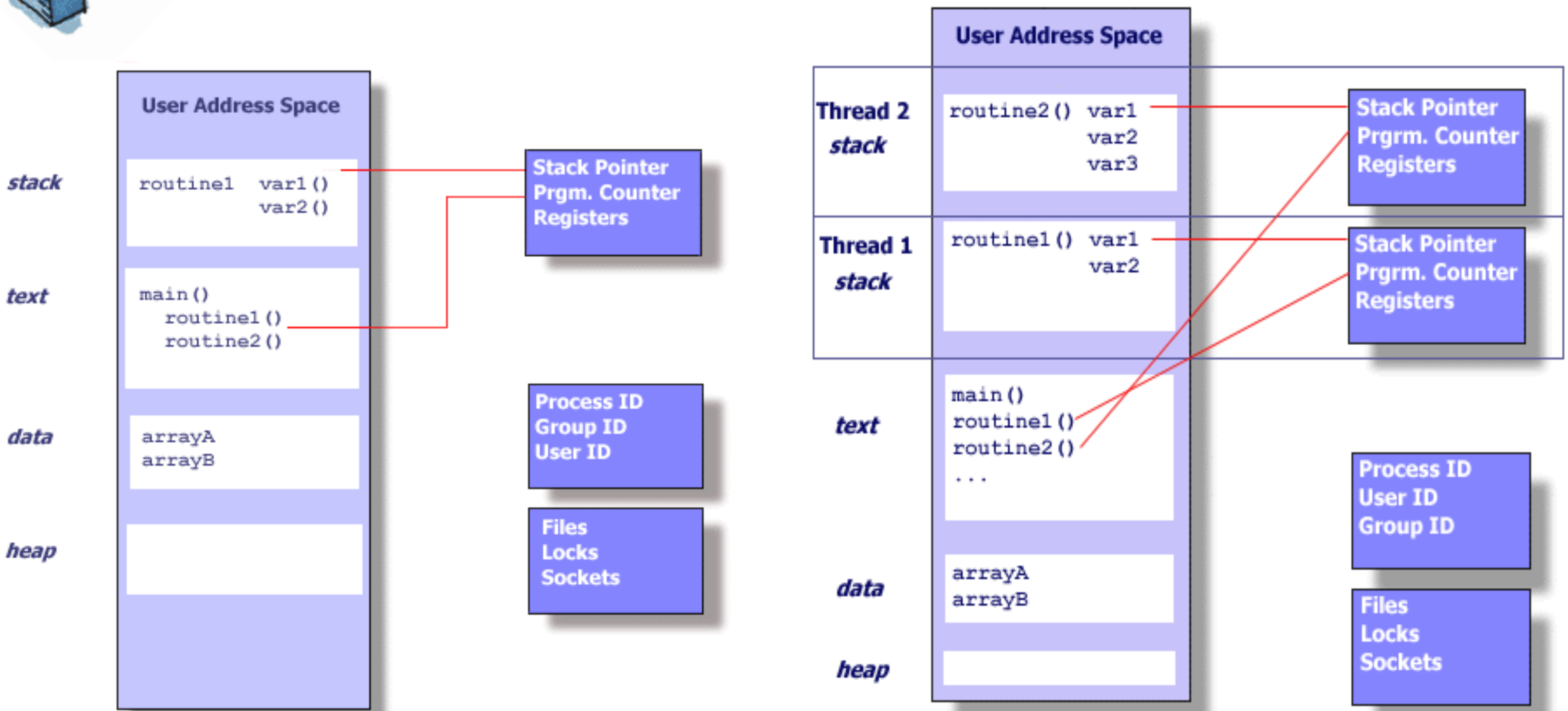


Figure 4.2 Single Threaded and Multithreaded Process Models

Unix Process vs thread



Benefits of Threads

- Takes less time to create or terminate a new thread than a process
- Switching between two threads takes less time that switching processes
- Threads can communicate with each other
  - without invoking the kernel

Thread use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
  - e.g., execution advances while a thread waits for I/O
- Modular program structure

Threads

- Several actions can affect all of the threads in a process
  - OS must manage these at the process level
- Examples:
  - Suspending a process involves suspending all threads of the process *(same address space!)*
  - Termination of a process terminates all threads within the process

Activities similar to Processes

- Threads have execution states and may synchronize with one another
  - Similar to processes
- We look at these two aspects of thread functionality in turn
  - States
  - Synchronisation

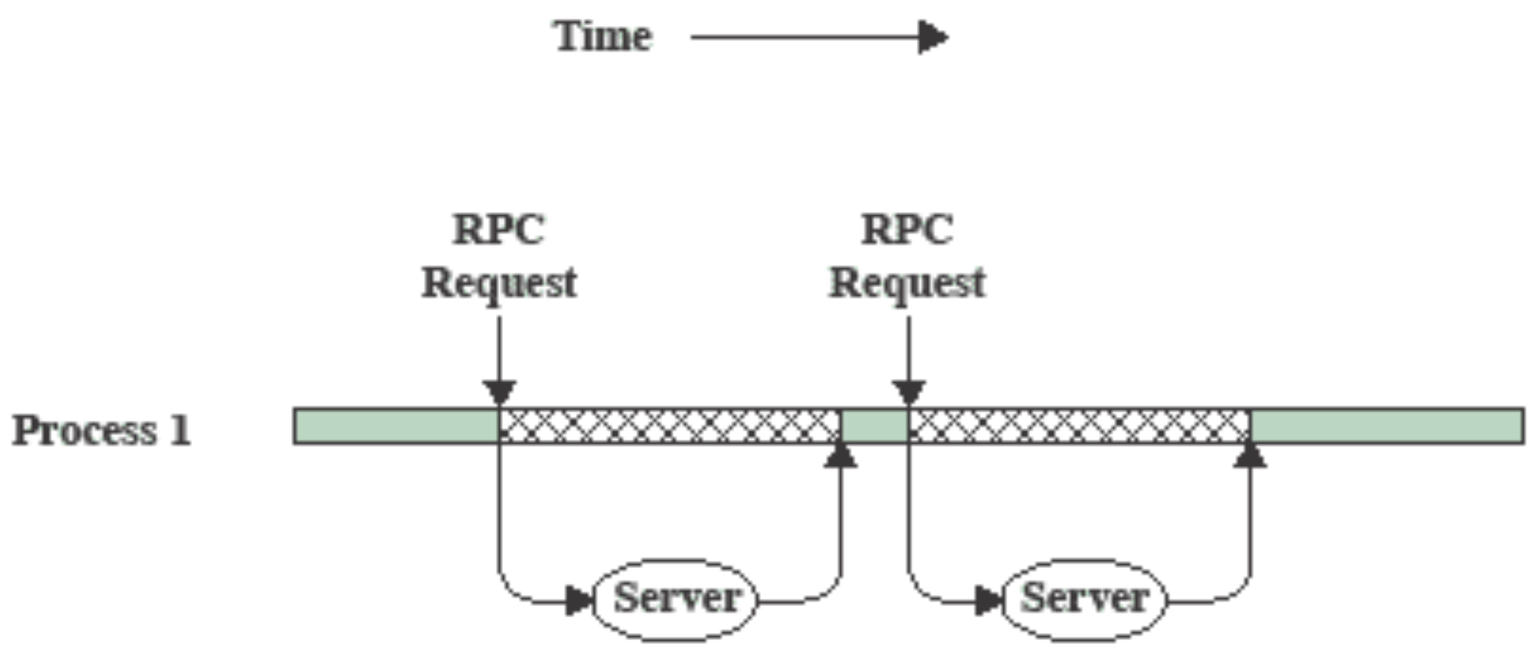
Thread Execution States

- States associated with a change in thread state
  - Spawn (another thread)
  - Block
    - Issue: can blocking a thread result in blocking some other thread, or even the whole process?
  - Unblock
  - Finish (thread)
    - Deallocate register context and stacks

Example: Remote Procedure Call

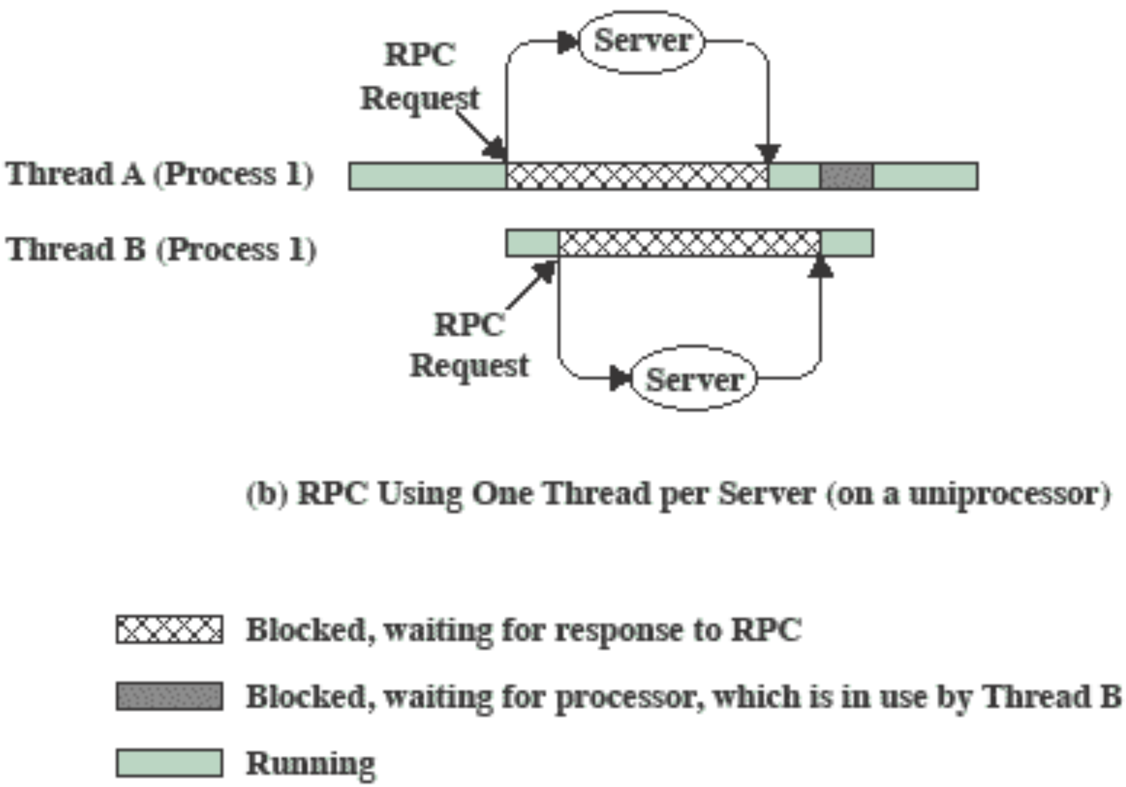
- Consider:
  - A program that performs two remote procedure calls (RPCs)
  - to two different hosts
  - to obtain a combined result.

RPC Using Single Thread



(a) RPC Using Single Thread

RPC Using One Thread per Server



(b) RPC Using One Thread per Server (on a uniprocessor)

Multithreading on a Uniprocessor

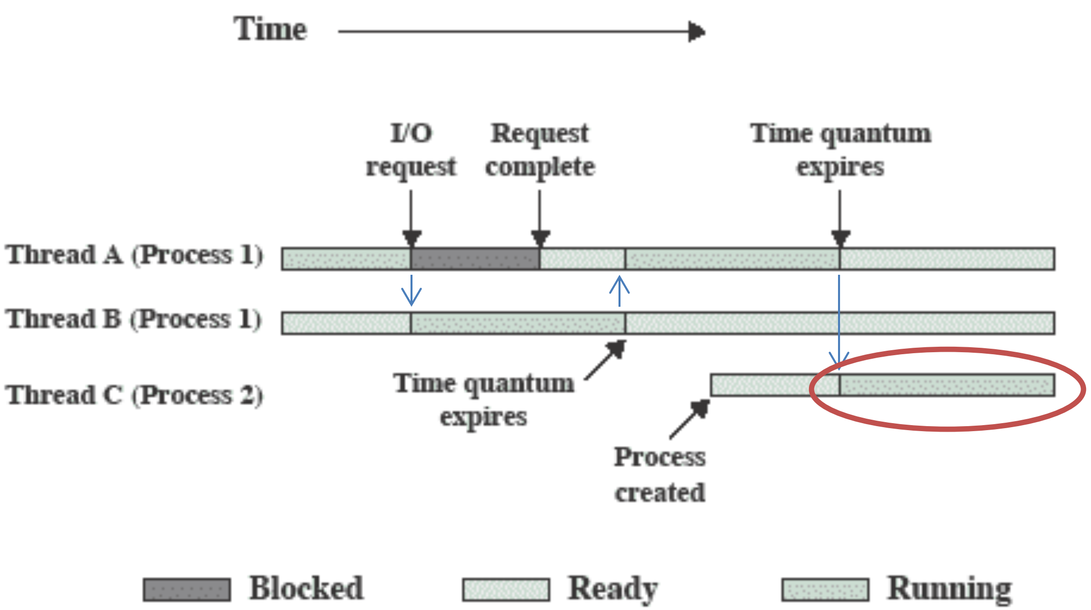


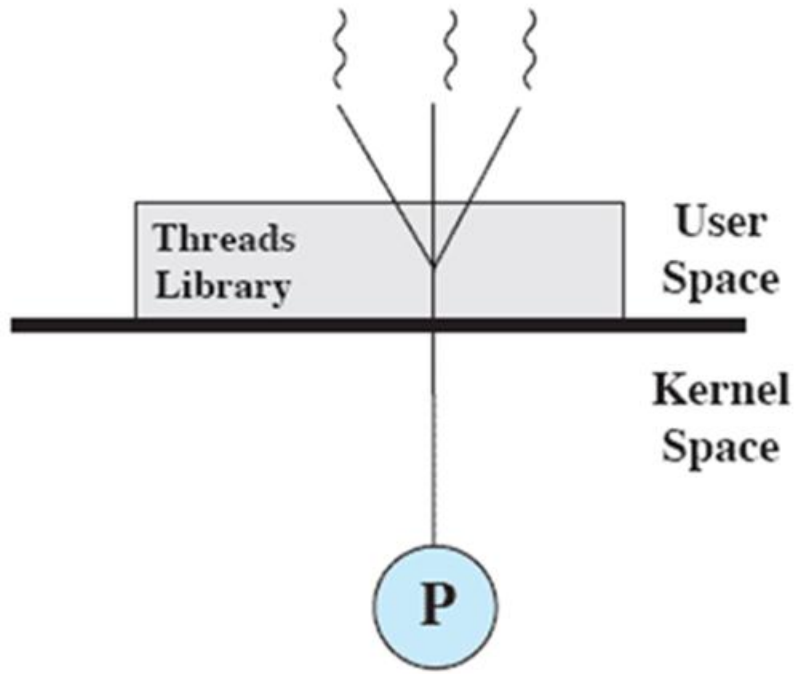
Figure 4.4 Multithreading Example on a Uniprocessor

Categories of Thread Implementation

- User Level Thread (ULT)
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes

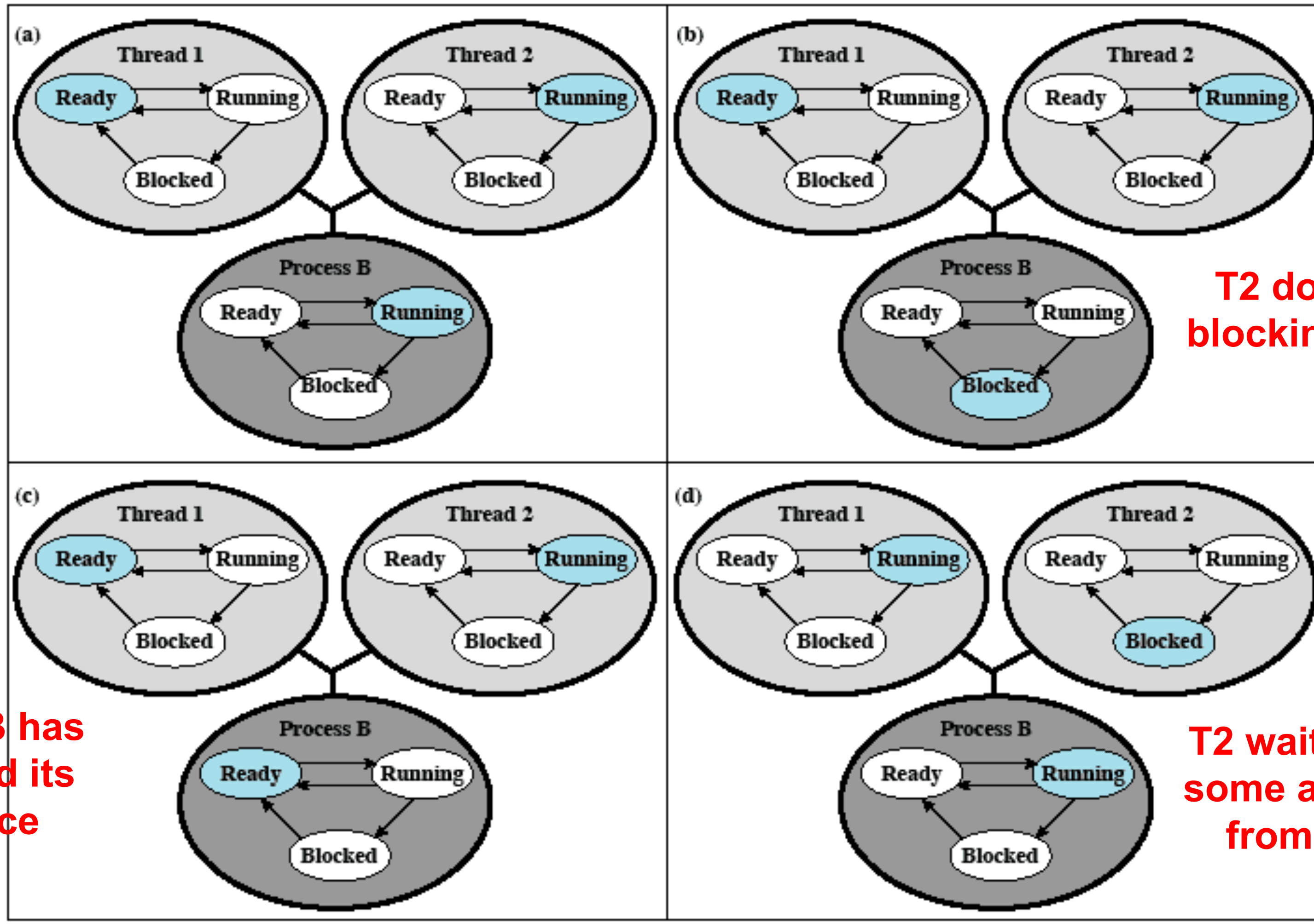
User-Level Threads

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



(a) Pure user-level

Relationships between ULT Threads and Process States



T2 does a blocking call

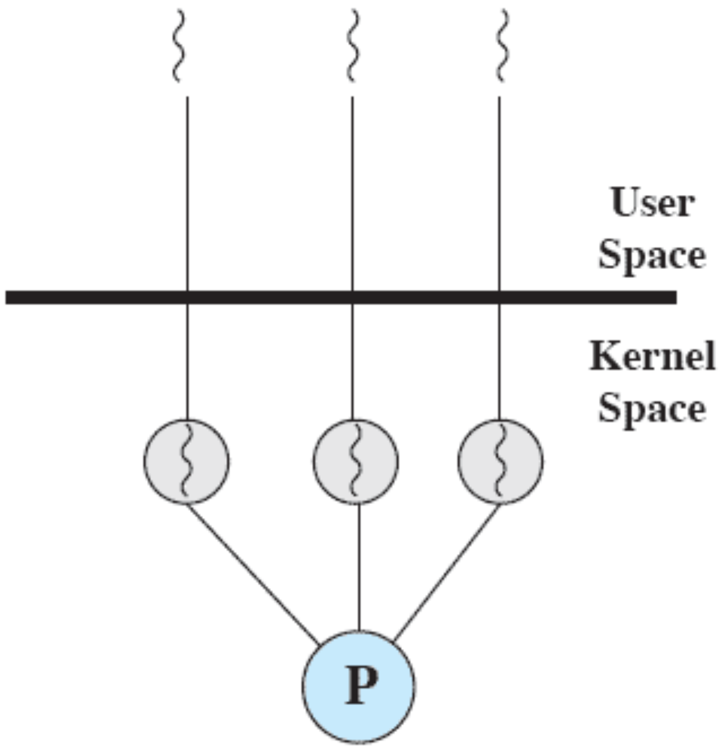
Process B has exhausted its time slice

T2 waits for some action from T1

Colored state is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Kernel-Level Threads



(b) Pure kernel-level

- Kernel maintains context information for the process and the threads
  - No thread management done by application
- Scheduling is done on a thread basis
- Windows is an example of this approach

Advantages of ULT

- Application-specific thread scheduling (i.e., independent of kernel)
- Thread switch does not require kernel privilege/switch to kernel mode
- ULTs run on any OS: implementation is done through a thread library at user level

Disadvantages of ULT

- A blocking systems call executed by a thread blocks *all threads* of the process
- Pure ULTs does not take full advantage of multiprocessors/multicores architectures

Advantages of KLT

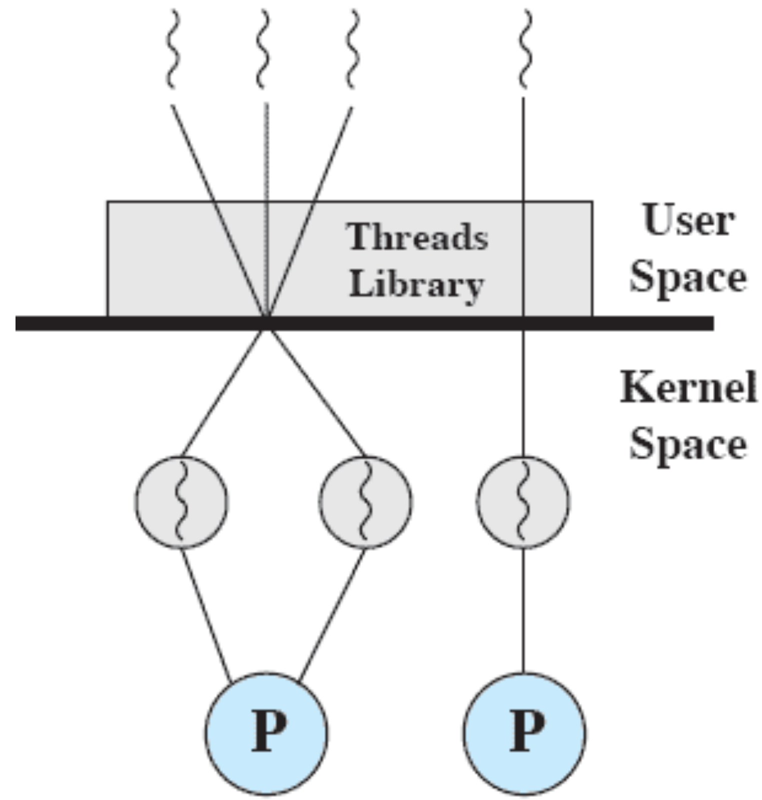
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

Disadvantage of KLT

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

Combined Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done within the application
- u ULTs are mapped onto k KLTs (k=u in Solaris)



(c) Combined

Threads & Processes: Possible Arrangements

Table 4.2 Relationship Between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
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

Roadmap

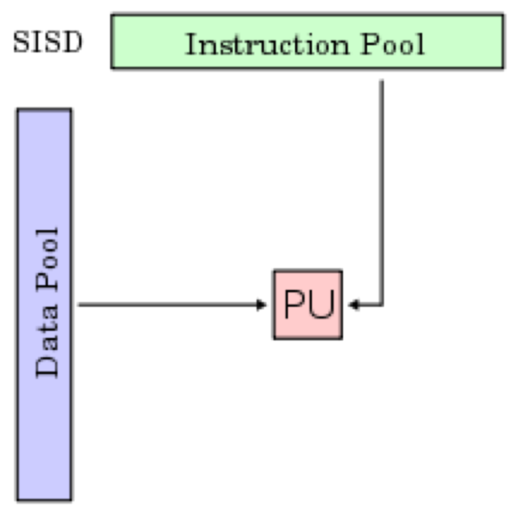
- Processes: fork(), wait()
- Threads: resource ownership and execution
- Symmetric multiprocessing (SMP)
- Case study:
  - PThreads

Traditional View

- Traditionally, the computer has been viewed as a sequential machine
  - A processor executes instructions one at a time in sequence
  - Each instruction is a sequence of operations
- Some popular approaches to parallelism
  - Symmetric MultiProcessors (SMPs)
  - Clusters

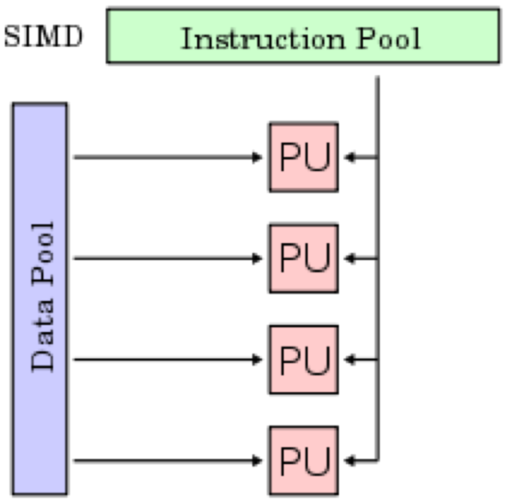
Categories of Computer Systems (Flynn's Taxonomy)

- Single Instruction Single Data (SISD)
  - Single processor executes a single instruction stream to operate on data stored in a single memory



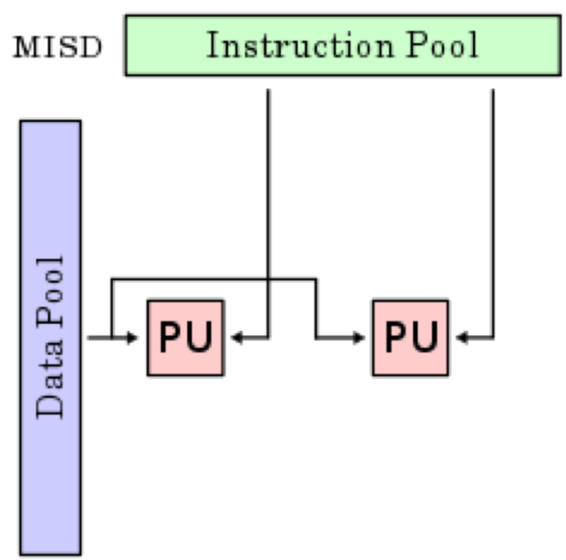
Categories of Computer Systems

- Single Instruction Multiple Data (SIMD)
  - Each instruction is executed on a different set of data by the different processors



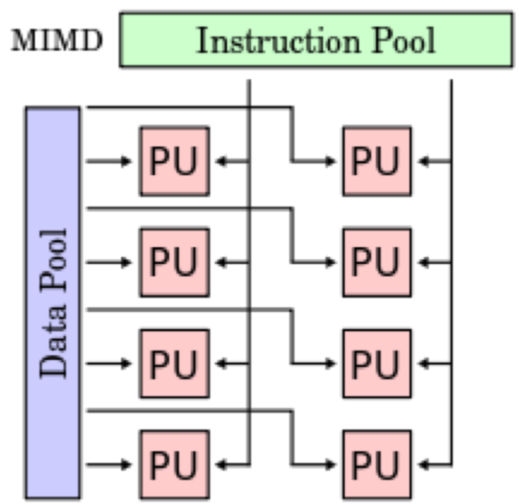
Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each executing a different instruction sequence



Categories of Computer Systems

- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets



Parallel Processor Architectures

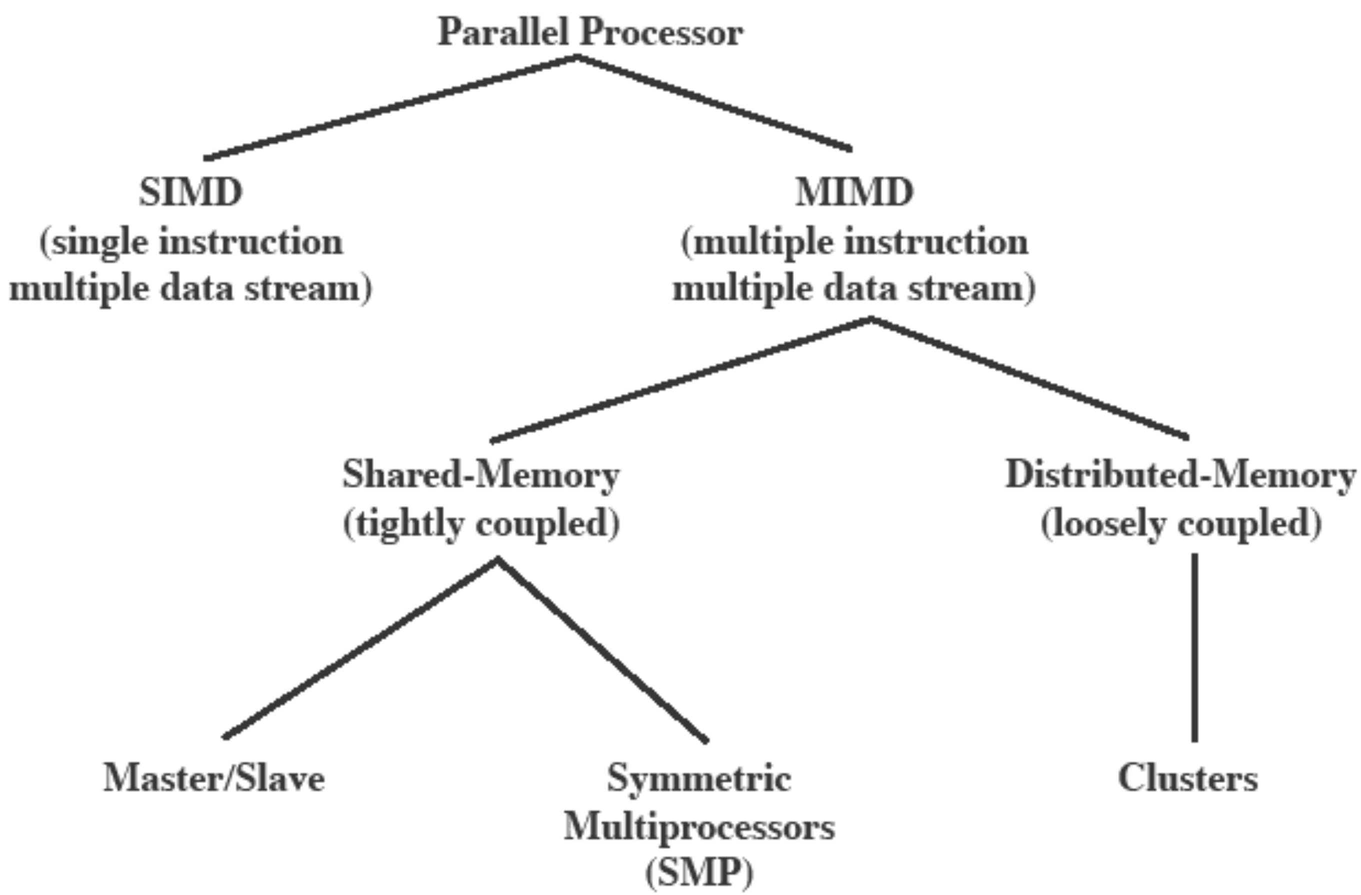


Figure 4.8 Parallel Processor Architectures

Typical SMP organization

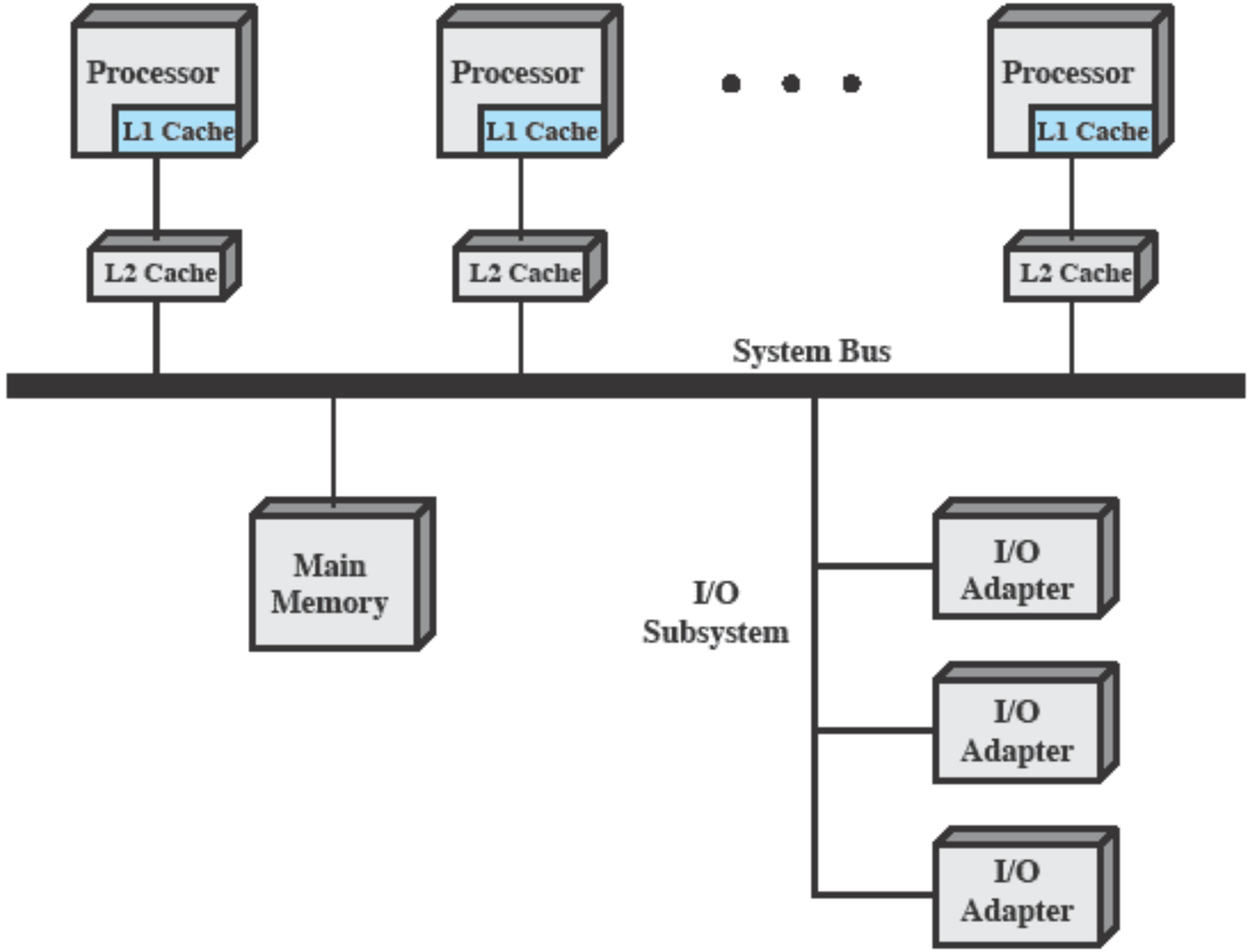


Figure 4.9 Symmetric Multiprocessor Organization

Multiprocessor OS Design Considerations

- The key design issues include
  - Simultaneous concurrent processes or threads
  - Scheduling
  - Synchronization
  - Memory Management
  - Reliability and Fault Tolerance

Roadmap

- Processes: fork (), wait()
- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Case study:
  - PThreads

POSIX Threads (PThreads)

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.
- Pthreads are C language programming types defined in the `pthread.h` header/include file.

Why Use Pthreads

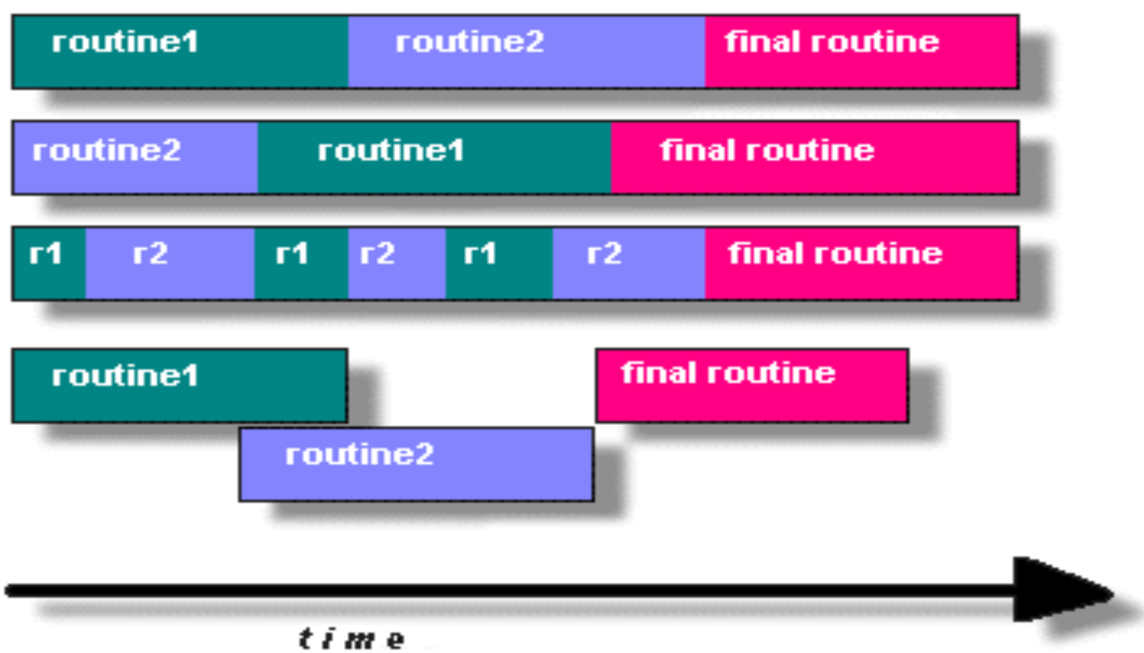
- The primary motivation behind Pthreads is improving program performance
- Can be created with much less OS overhead
- Need fewer system resources to run
- Timing comparison (next slide)
  - forking processes vs `pthread_create()`
  - timings reflect 50K process/thread creations (unit: s)

Threads vs Forks

PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Designing Threaded Programs as in Parallel Programming

- To take advantage of Pthreads, a program should be organized into discrete, independent tasks that can execute concurrently
  - E.g., if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



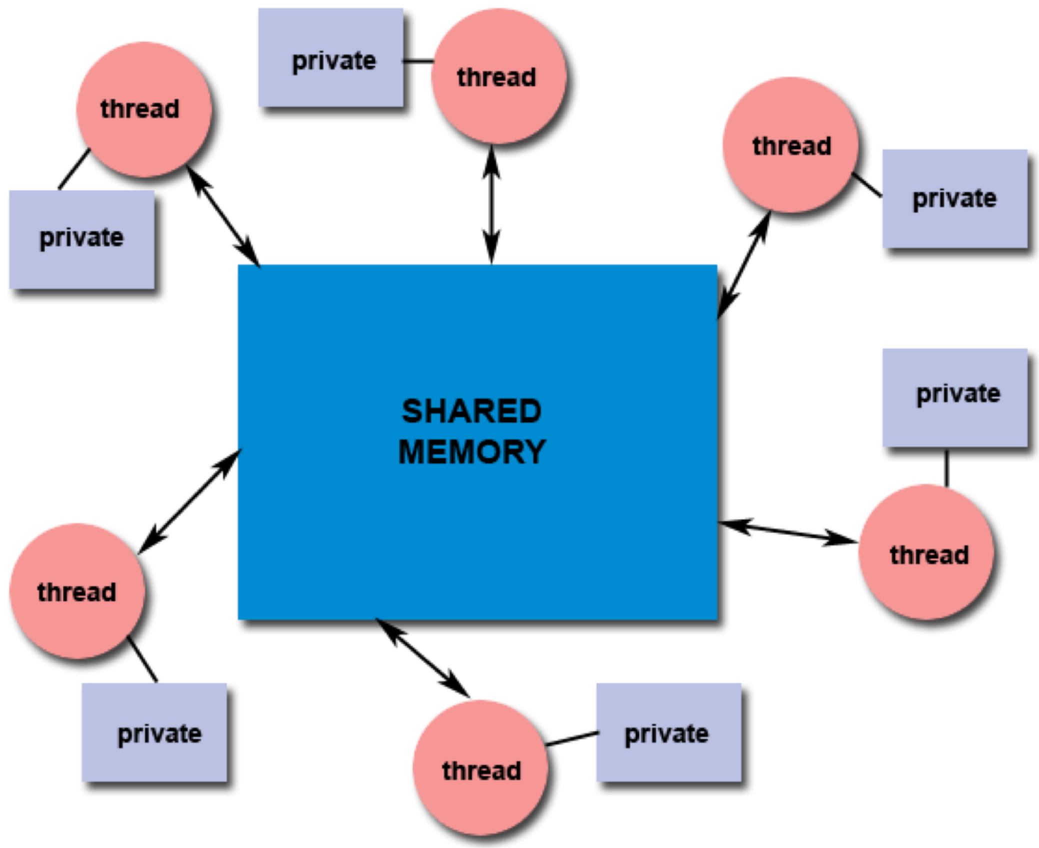
Models for Threaded Programs

- Manager/worker
  – A *manager* thread assigns work to other threads, the *workers*. Manager handles input and hands out the work to the other tasks
- Pipeline
  – A task is broken into a series of suboperations, each handled in series, but concurrently, by a different thread

Shared-memory Model

- All threads have access to the *same* global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access to (i.e., protecting) globally shared data

Shared-memory Model



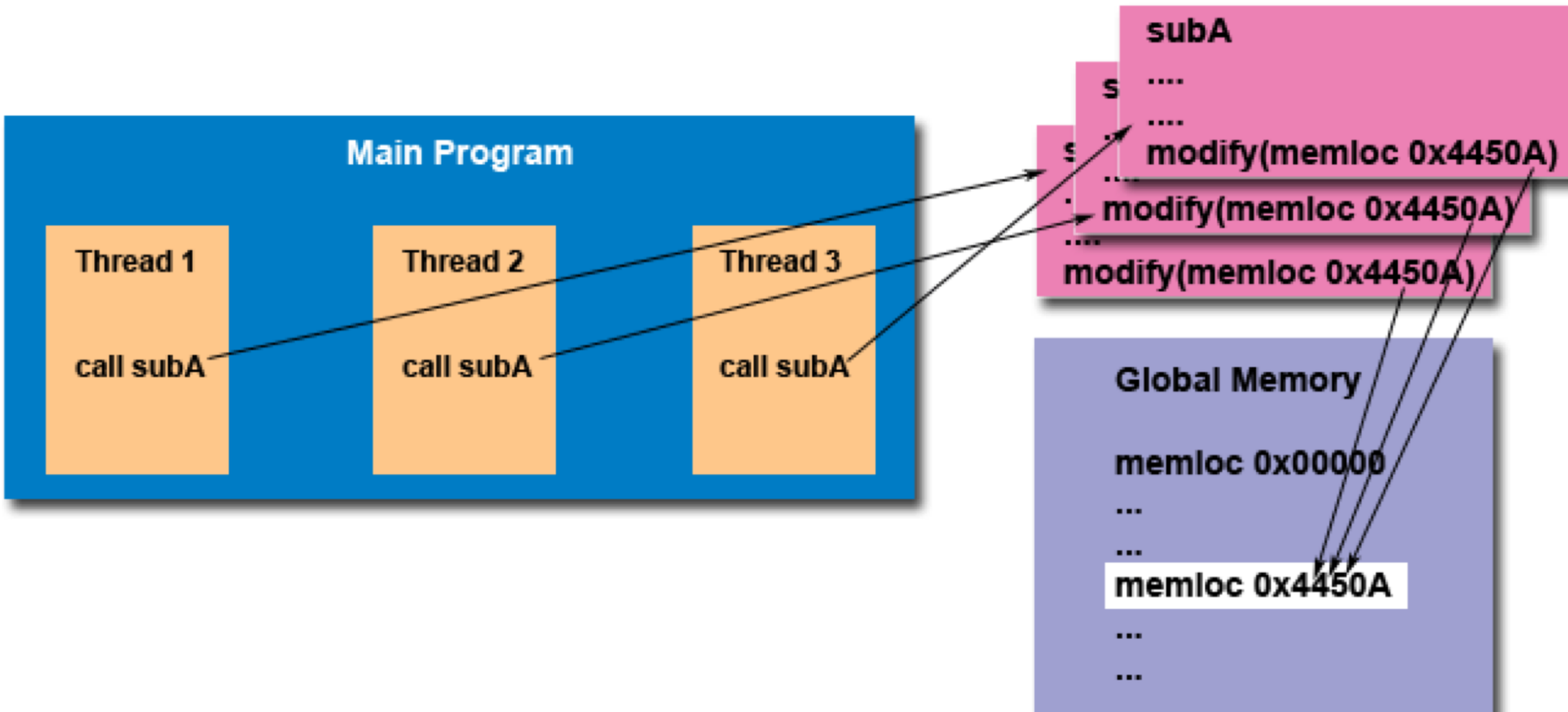
Thread Safety

- A code is thread-safe when multiple threads can execute it simultaneously without unintended interactions
  - (without *clobbering* shared data)
  - (without creating *race conditions*)

Thread Safety

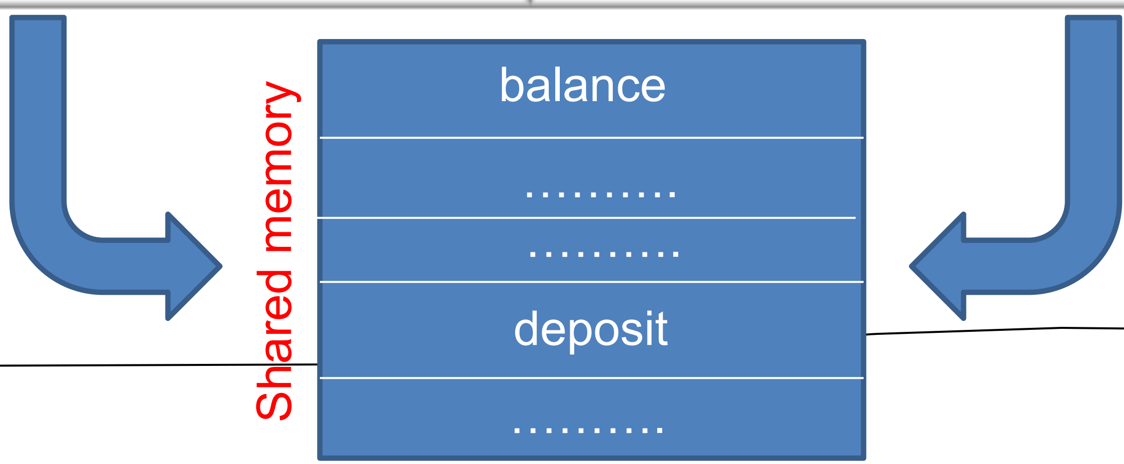
- Example: an application creates several threads, each of which makes a call to the same library routine:
  - The library routine accesses/modifies a global structure or location in memory
  - As each thread calls this routine, it is possible that they may try to modify this structure/location at the same time
  - If the routine does not employ some sort of *synchronization mechanism* to prevent data corruption, then it is not thread-safe

Thread Safety



Thread Safety

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200



Pthreads: Creating Threads

- A program's main() method comprises a single, default thread.
- `pthread_create()` creates a new thread and makes it executable
  - The maximum number of threads that a process can create is implementation dependent
  - Once created, threads are *peers*, and can create other threads as well

Pthreads: Terminating Threads

- Several ways to terminate a thread, e.g.:
  - The thread is complete, i.e., the function it started with reaches a `return` statement
  - `pthread_exit()` is called
  - `exit()` is called *(affects the entire program!)*
  - The main terminates without executing `pthread_exit()` [caveat: `pthread_detach()`]
- `pthread_exit()` is called once a thread has completed its work and it is no longer required to exist.

Pthread: Terminating Threads (cont)

- If the main thread finishes before any other thread does, the other threads will continue executing if `pthread_exit()` was used to terminate the main, or if `pthread_detach()` was used on them
- `pthread_exit()` doesn't free resources (e.g., any file opened inside the thread will stay open), so bear cleanup in mind!

Pthread Example (1/2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void* printHello(void *arg)  {
   int threadID = *(int*) arg;
   printf("Hey! It's me, thread #%d!\n",
          threadID);
   pthread_exit(NULL);
}
```

Pthread Example (2/2)

```
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int ret, t;
   for (t=0; t<NUM_THREADS; t++) {
      printf("In main: creating thread %d\n", t);
      int *arg = malloc(sizeof(int));
      *arg = t;
      ret = pthread_create(&threads[t], NULL,
            printHello, (void*)arg);
      if (ret != 0) {
        printf("ERROR: code %d\n", ret); exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

One Possible Execution

In main: creating thread 0
In main: creating thread 1
Hey! It's me, thread #0!
In main: creating thread 2
Hey! It's me, thread #2!
Hey! It's me, thread #1!
In main: creating thread 3
In main: creating thread 4
Hey! It's me, thread #3!
Hey! It's me, thread #4!

Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```