Processes and Threads

Chapter 2: Sections 2.1 - 2.2

Process Control Block

- The kernel maintains a data structure to keep track of all the process information called the process control block or (PCB).
  - One per process
- The PCB also includes pointers to other data structures describing resources used by the process such as files (open files table) and memory (page tables).
- Maintains the state of the process
  - Over 170+ fields

Some Process Control Block

- Memory
- Open streams/files
- Devices, including abstract ones like windows
- Links to condition handlers (signals)
- Processor registers (single thread)
- Process identification
- Process state
- Priority
- Owner
- Which processor
- Links to other processes (parent, children)
- Process group
- Resource limits/usage
- Access rights

Process Control Block

- Every task also needs its own stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space

Process Control Block

- Different information is required at different times
- UNIX for example has two separate places in memory with the process control block and process stack. One of them is in the kernel the other is in user space.
- Why? User land data is only required when the process is running.

Why a kernel stack?

- Kernels can't trust addresses provided by user
- Address may point to kernel memory that is not accessible to user processes
- Address may not be mapped
- Memory region may be swapped out from physical RAM
- Leftover data from kernel ops could be read by process
  - Kernel-level heartbleed bug

Process Tables

- The OS holds the process control blocks in the process table
- Usually implemented as an array of pointers to process control block structures
  - Linux calls the PCB task_struct

Linux PCB Data Structure

```
1166 struct task_struct {
1167         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1168         void *stack;
1169         atomic_t usage;
1170         unsigned int flags;     /* per process flags, defined below */
1171         unsigned int ptrace;
1172
1173         int lock_depth;         /* BKL lock depth */
1174
1175 #ifdef CONFIG_SMP
1176 #ifdef __ARCH_WANT_UNLOCKED_CTXSW
1177         int oncpu;
1178 #endif
1179 #endif
1180
1181         int prio, static_prio, normal_prio;
1182         unsigned int rt_priority;
1183         const struct sched_class *sched_class;
1184         struct sched_entity se;
1185         struct sched_rt_entity rt;
1186
1187 #ifdef CONFIG_PREEMPT_NOTIFIERS
1188         /* list of struct preempt_notifier: */
1189         struct hlist_head preempt_notifiers;
1190 #endif
1191
1192         /*
1193          * fpu_counter contains the number of consecutive context switches
1194          * that the FPU is used. If this is over a threshold, the lazy fpu
1195          * saving becomes unlazy to save the trap. This is an unsigned char
1196          * so that after 256 times the counter wraps and the behavior turns
1197          * lazy again; this to deal with bursty apps that only use FPU for
1198          * a short time
1199          */
1200         unsigned char fpu_counter;
1201         s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
```

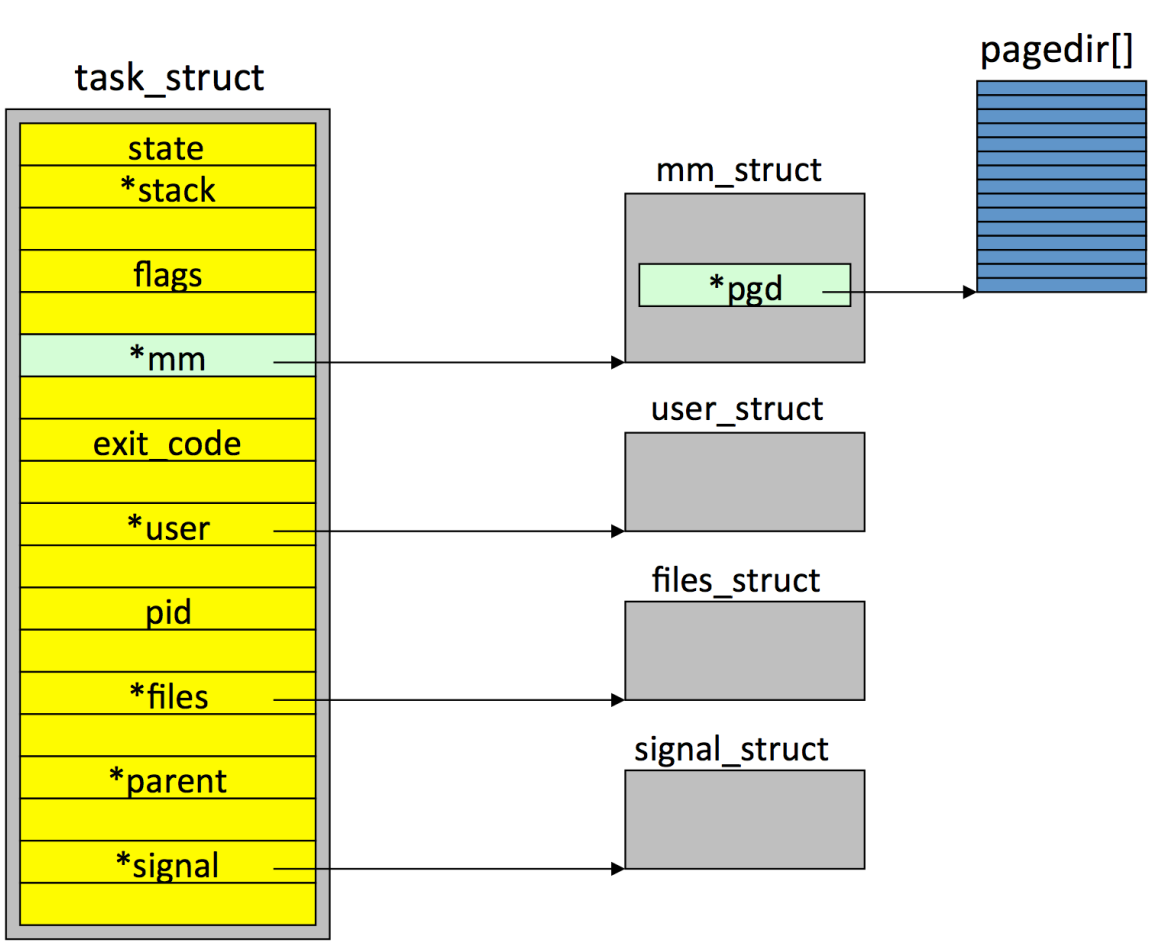
https://github.com/CSE3320/kernel-code/blob/master/linux-3.16.36/include/linux/sched.h

Linux Task_Struct

Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures



Pseudo parallelism

- In a single CPU system the CPU is only running one process at a time
  - Process are switched over the course of time giving the illusion of of parallelism.
- Contrast with a multiprocessor system in which multiple CPU share the same memory and execute processes in parallelism.

Process Model

- All running software on the computer is organized into a number of sequential processes.
- Conceptually each process has its own virtual CPU.
- In reality, the CPU is shared by the processes.
  - Multiprogramming: The rapid switching back and forth

The Process Model (2)

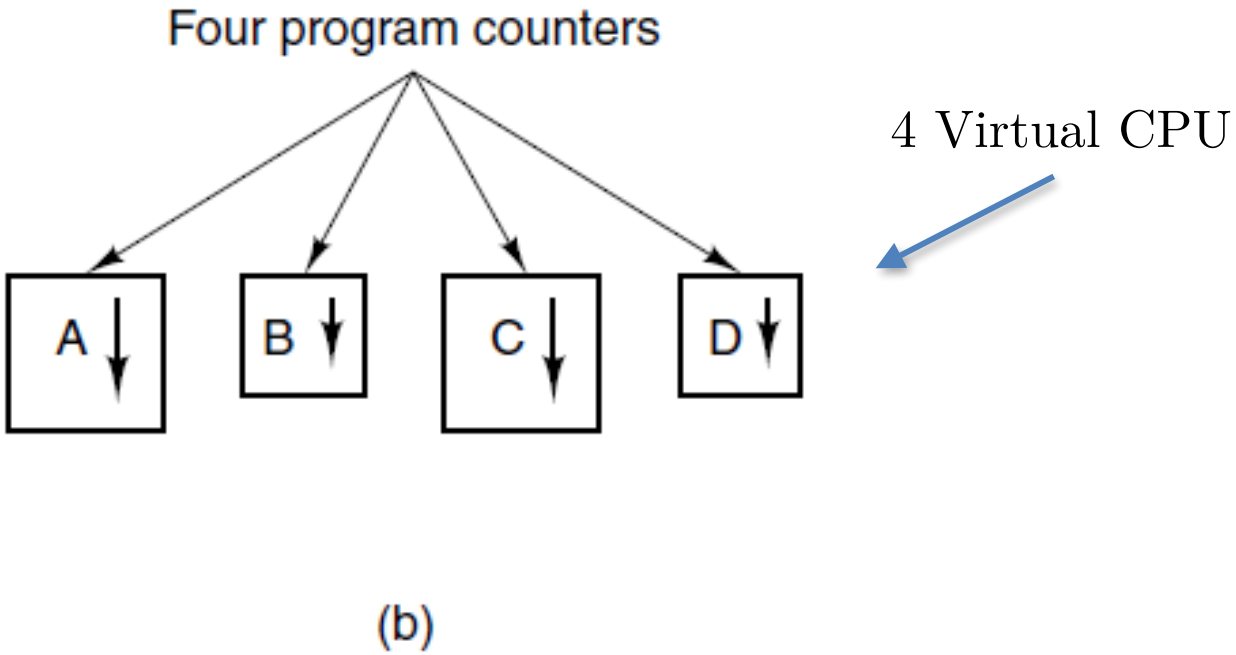


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (1)

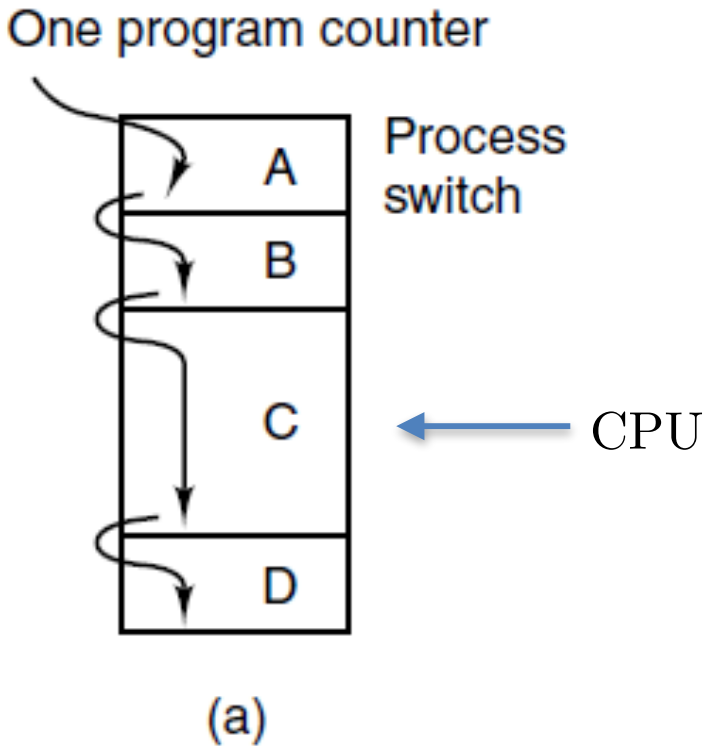


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (3)

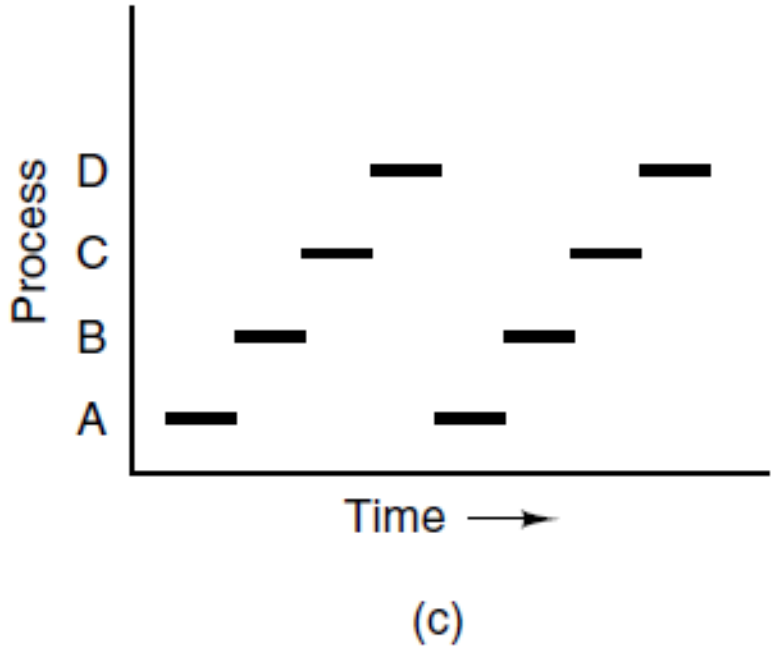
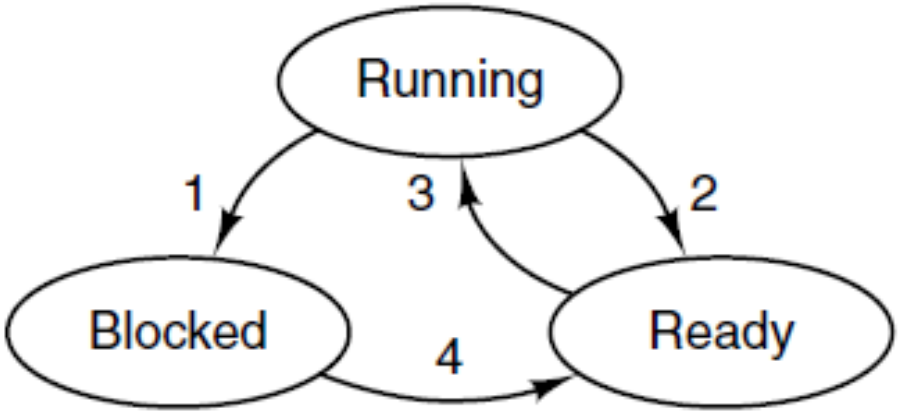


Figure 2-1. (c) Only one program is active at once.

Process States (2)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

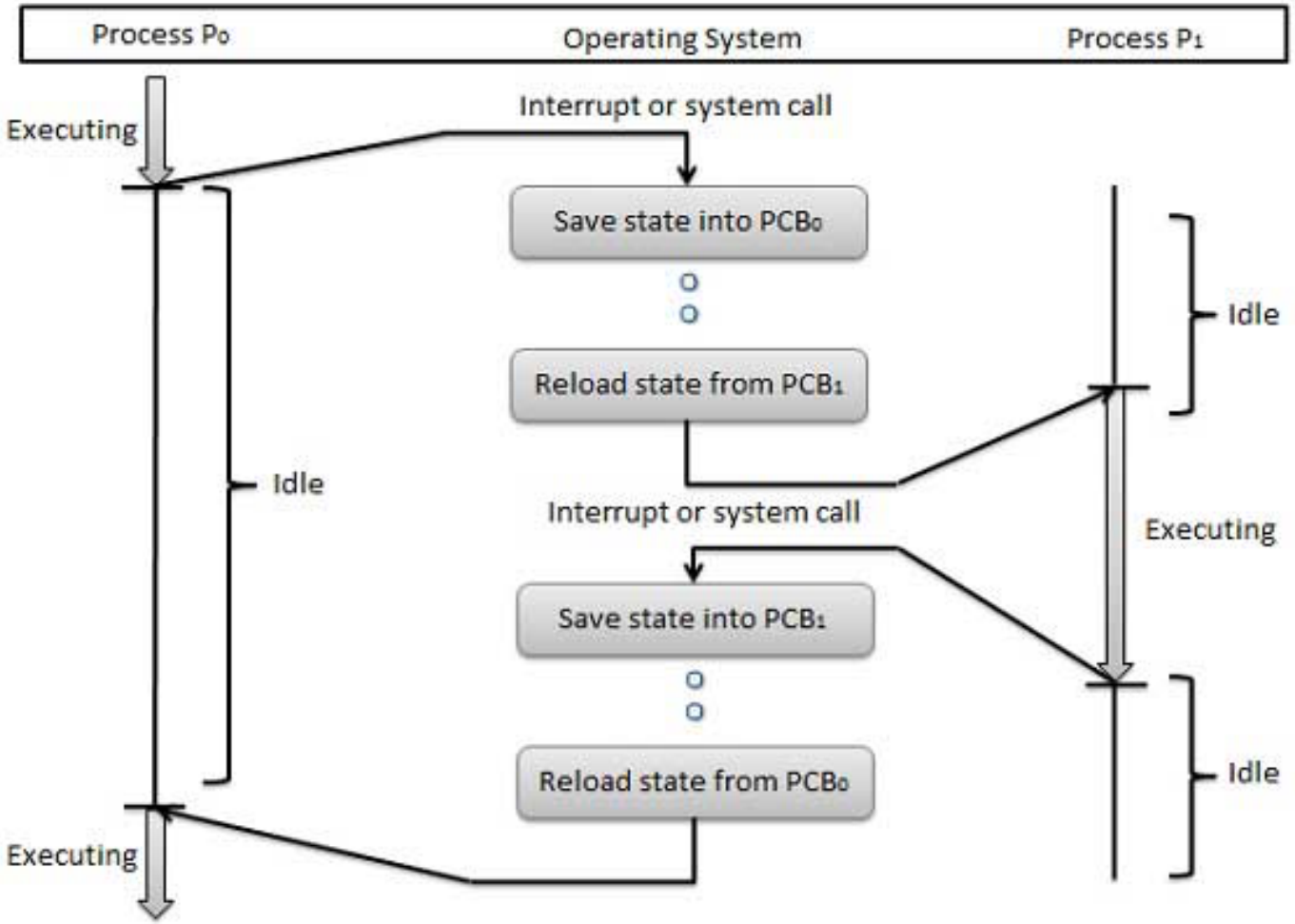
Context Switch

- A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one process or thread to another.
- A context is the contents of a CPU's registers and program counter at any point in time.

Context Switch in more detail

1. Suspend the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory.
2. Retrieve the context of the next process from memory and restoring it in the CPU's registers
3. Return to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

Context Switch



Process States (3)

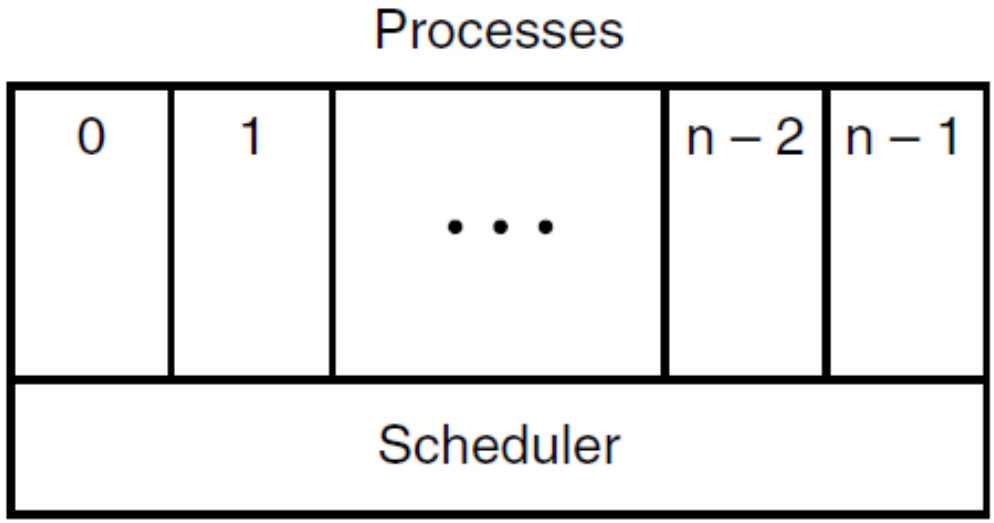
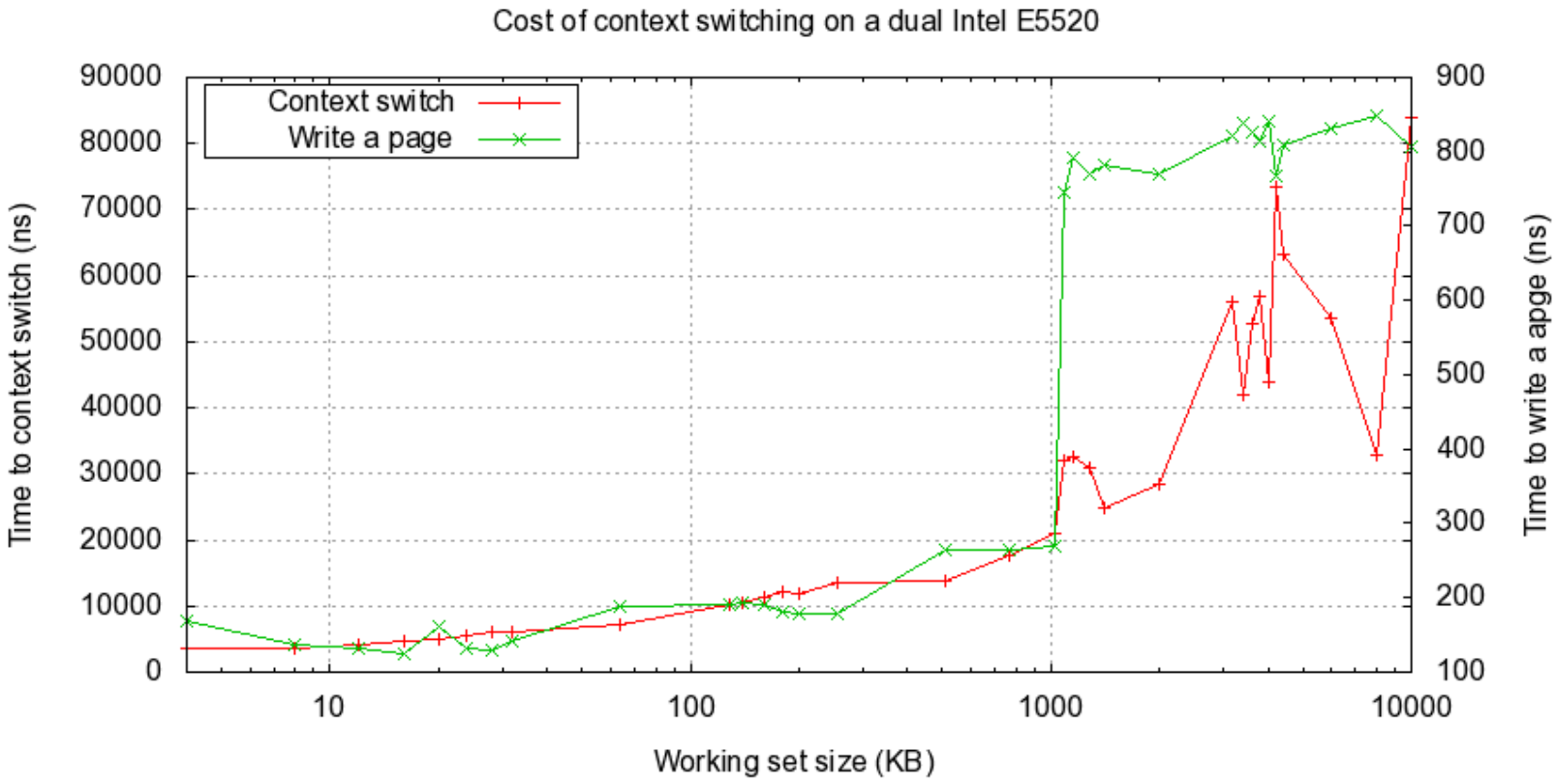


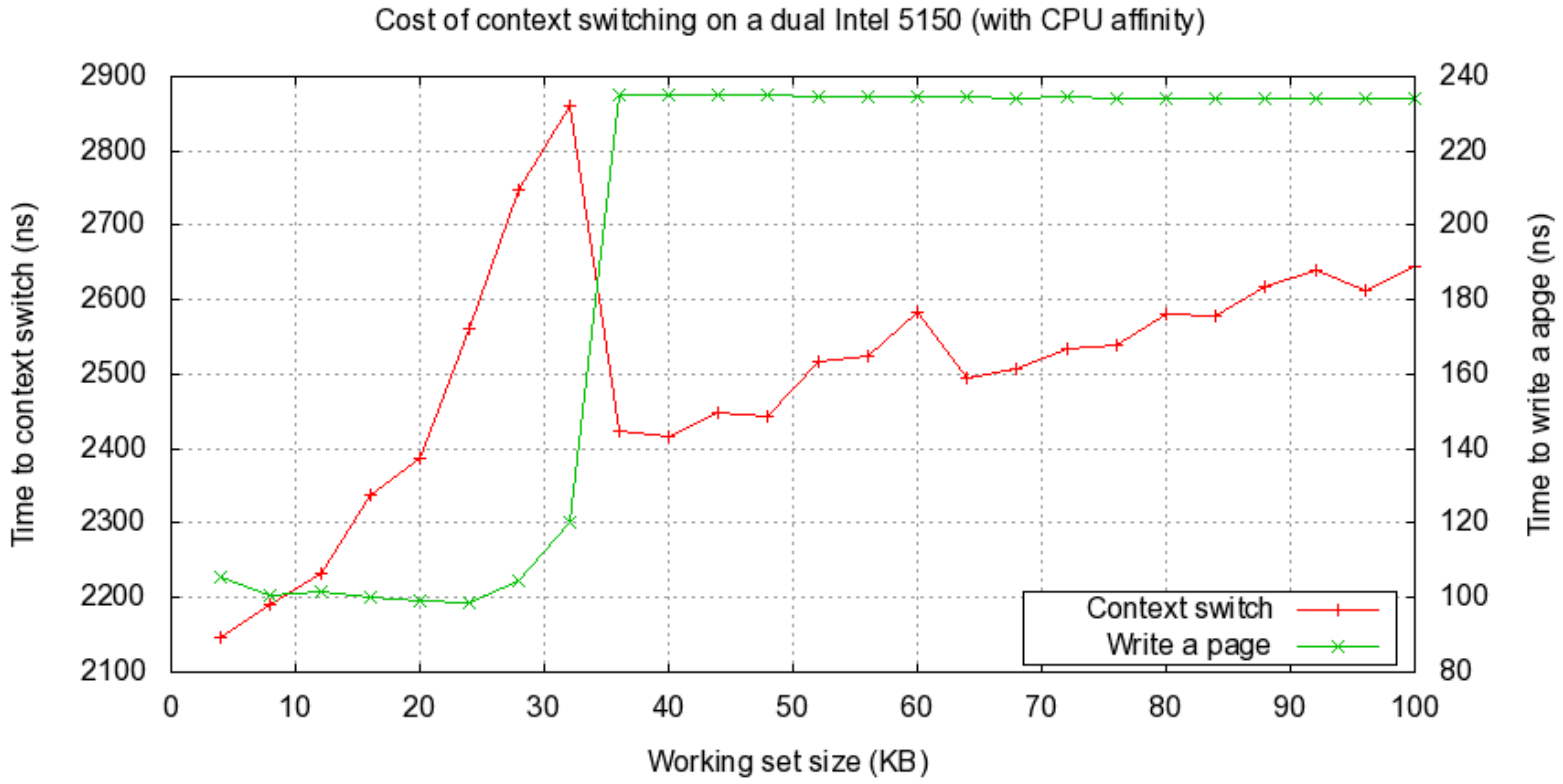
Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Cost of Context Switch



30µs of CPU overhead is a good rule of thumb for worst case context switch

Cost of Context Switch



CPU affinity can help context switching cost, but linux is pretty poor at scheduling CPU affinity.

Process States (1)

Three states a process may be in:

1.Running (actually using the CPU at that instant).

2.Ready (runnable; temporarily stopped to let another process run).

3.Blocked (unable to run until some external event happens).

CPU Utilization

- Multiprogramming improves CPU utilization
  - If the average process computes 20% of the time, then 5 processes are needed to fully utilize the CPU.
    - Unrealistic as it assumed that all 5 process won't be waiting for I/O at the same time.

CPU Utilization

» Better model is a probabilistic view.

» Suppose a process ends a fraction p of its time waiting for I/O

» With n processes in memory, the probability that all n processes are waiting for I/O is p^n

$$\text{CPU utilization} = 1 - p^n$$

Modeling Multiprogramming

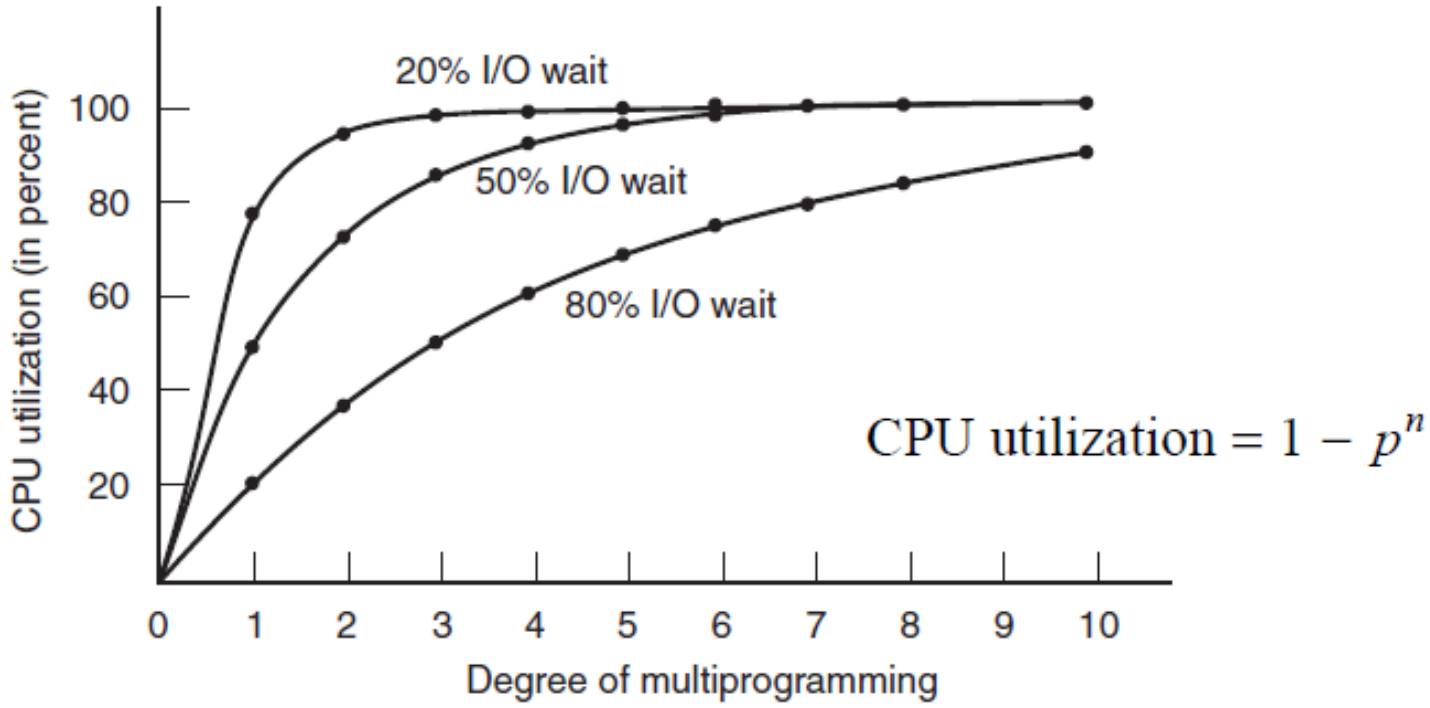


Figure 2-6. CPU utilization as a function of the number of processes in memory.

CPU Utilization

Example: Given 8GB of RAM, the OS taking 2GB of RAM and each user program taking 2GB we can run 3 user programs at once. With an 80% average I/O wait the CPU utilization is:

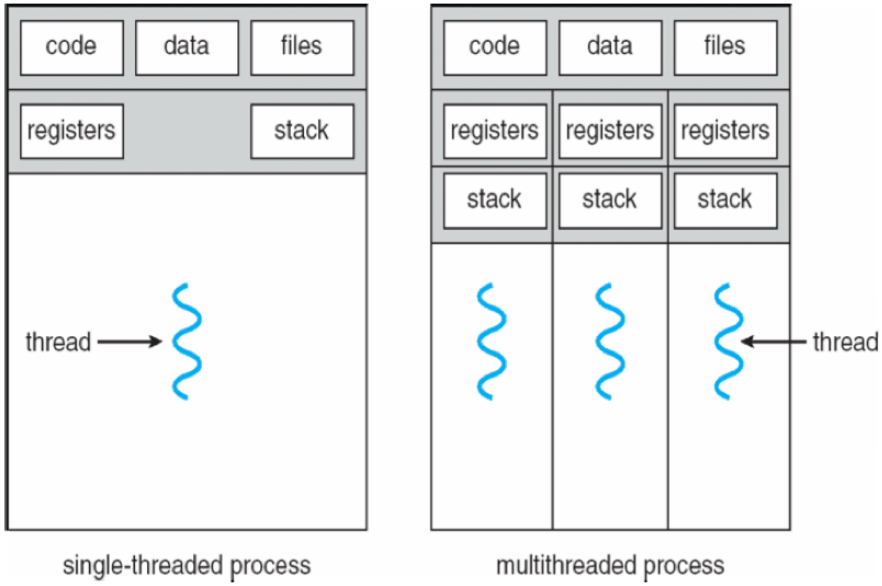
$$1 - 0.8^3 = 48.8\%$$

CPU Utilization

Example continued: Adding an additional 8 GB of memory would allow 7 processes at once and raise the CPU utilization:

$$1 - 0.8^7 = 79\%$$

Threads



» In traditional operating systems, each process has an address space and a single line of execution.

» Threads are multiple lines of execution in a shared address space

Thread benefits

» Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.

Thread benefits

- Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests
  - A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Thread Benefits

» When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

The Classical Thread Model (1)

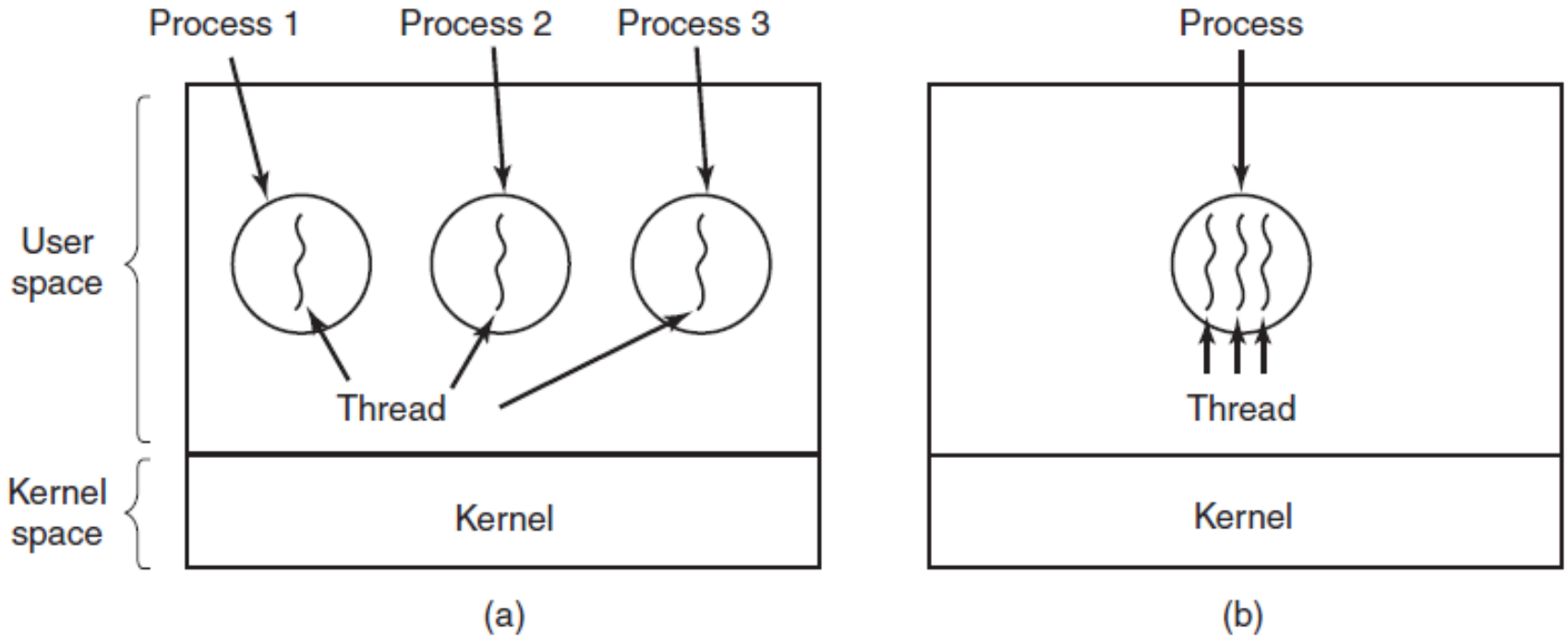


Figure 2-11. (a) Three processes each with one thread.
(b) One process with three threads.

The Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The Classical Thread Model (3)

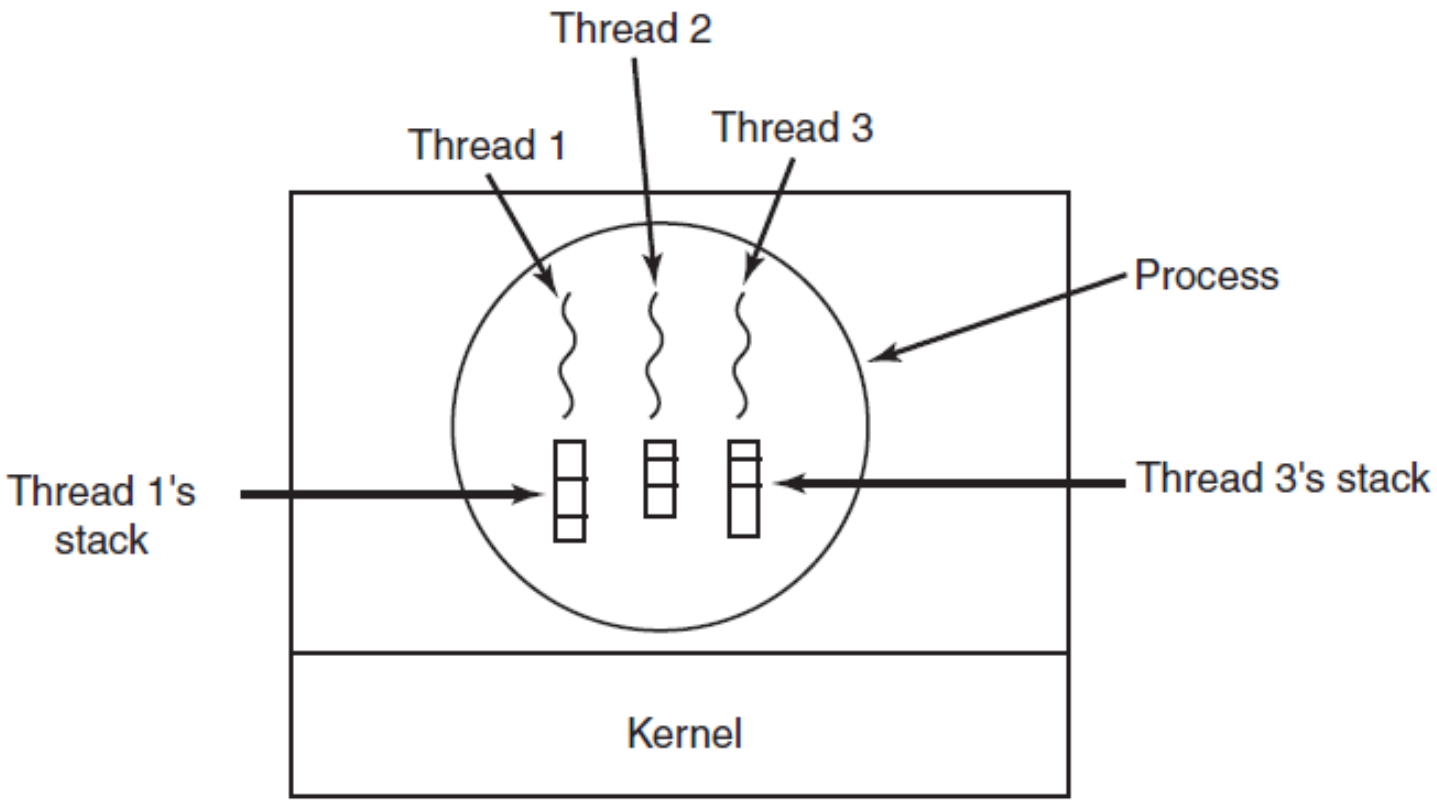


Figure 2-13. Each thread has its own stack.

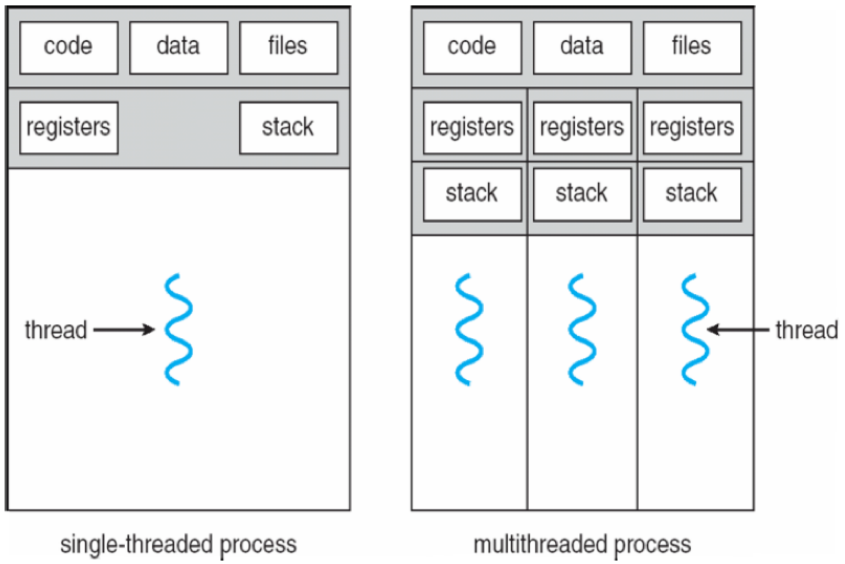
The Classical Thread Model

- Having multiple threads running in parallel is analogous to multiple processes running in parallel. With the exception:
  - In the former the threads share an address space and resources
  - In the latter the the processes share physical memory, disks, printers, etc.

Lightweight Processes

» Because threads share some of the characteristics of process, threads are sometimes referred to as lightweight processes.

The Classical Thread Model



» Different threads are not as independent as processes.

  » Shared address space means no protection between threads.

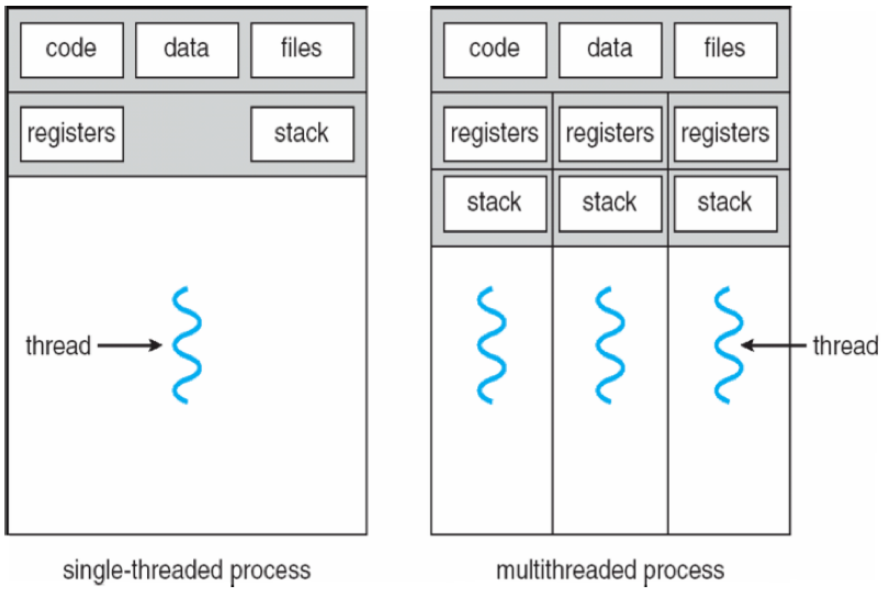
  » All threads share the same global variables.

The Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The Classical Thread Model



» Since each thread calls different procedures and has a different execution history, it needs its own stack.

» As with processes, each thread will follow the process lifecycle.

pthreads Overview

» What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.

pthreads Overview

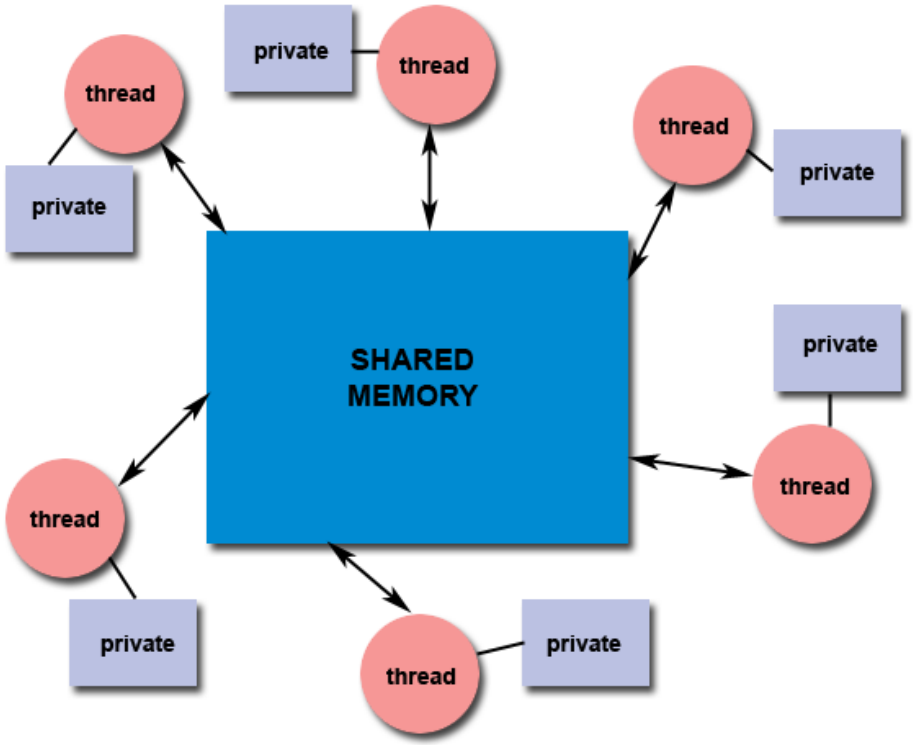
» For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

The important stuff

» pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header file and a thread library - though this library may be part of another library, such as libc, in some implementations.

Shared Memory Model

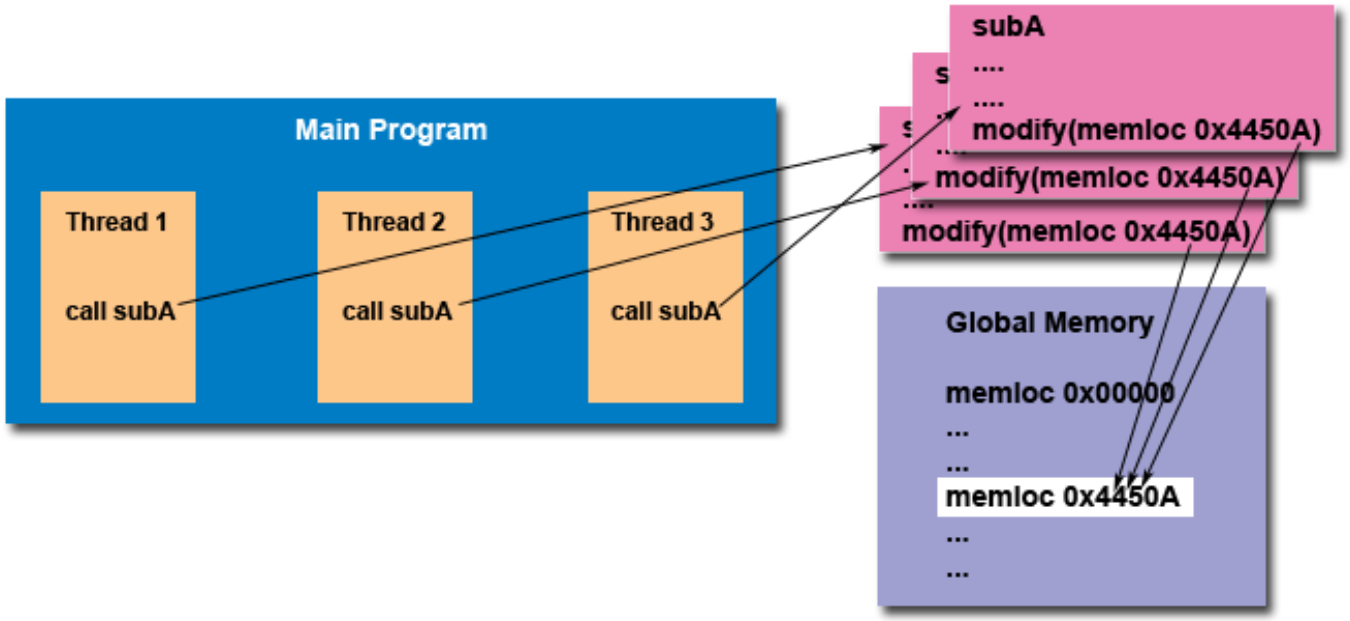


- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.

Thread Safeness

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose that your application creates several threads, each of which makes a call to the same library routine:
- This library routine accesses/modifies a global structure or location in memory.

Thread Safeness



- As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.

Thread Safeness

- The implication to users of external library routines is that if you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.
- Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc.

Potentially Thread Unsafe

All functions defined by this volume of POSIX.1-2008 shall be thread-safe, except that the following functions need not be thread-safe

asctime()
basename()
catgets()
crypt()
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()
encrypt()
endgrent()
endpwent()
endutxent()
ftw()
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mblen()
mbtowc()
mrand48()
nftw()
nl_langinfo()
ptsname()
putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strsignal()
strtok()
system()
ttyname()
unsetenv()
wctomb()

Potentially Thread Unsafe

If a glibc function is not thread-safe then the man page will say so, and there will (most likely) be a thread safe variant also documented.

See, for example, man strtok:

SYNOPSIS #include

```
char *strtok(char *str, const char *delim);

char *strtok_r(char *str, const char *delim, char **saveptr);
```

pthreads API

» The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

1. **Thread management**: Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

pthreads API

3. **Condition variables**: Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
4. **Synchronization**: Routines that manage read/write locks and barriers.

Naming Convention

Naming conventions: All identifiers in the threads library begin with pthread_. Some examples are shown below:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

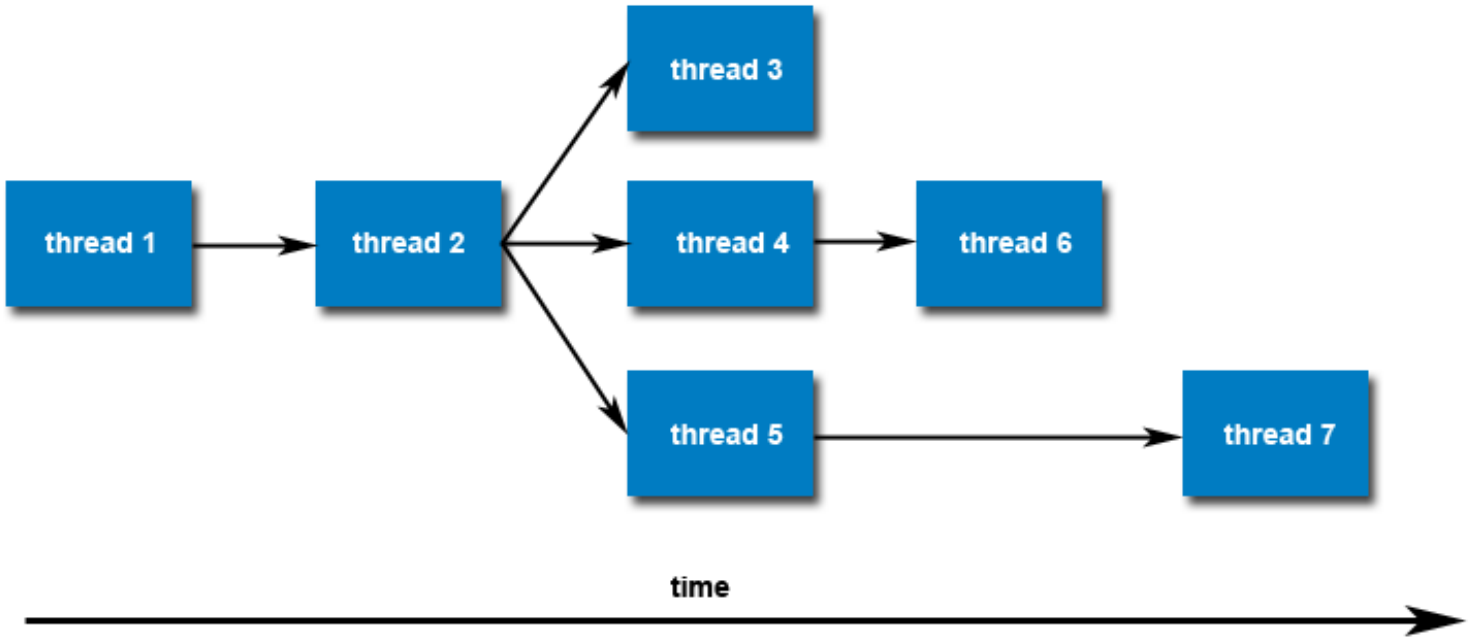
Creating Threads

```
pthread_create (thread,attr,start_routine,arg)
```

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
  - `pthread_create` arguments:
  - `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
  - `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - `start_routine`: the C routine that the thread will execute once it is created.
  - `arg`: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Thread Hierarchy

Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Thread Attributes

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.

Terminating Threads

» There are several ways in which a thread may be terminated.

```
pthread_exit   (status)
pthread_cancel (thread)
```

  - The thread returns normally from its starting routine. Its work is done.
  - The thread makes a call to the `pthread_exit` subroutine - whether its work is done or not.
  - The thread is canceled by another thread via the `pthread_cancel`

Terminating Threads

- The `pthread_exit()` routine allows the programmer to specify an optional termination status parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).
- In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()` - unless, of

Terminating Threads

- There is a definite problem if `main()` finishes before the threads it spawned if you don't call `pthread_exit()` explicitly. All of the threads it created will terminate because `main()` is done and no longer exists to support the threads.
- By having `main()` explicitly call `pthread_exit()` as the last thing it does, `main()` will block and be kept alive

pthread example

» \

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

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

This example creates 5 threads with the `pthread_create()` routine. Each thread prints a "Hello World!" message, and then terminates with a call to `pthread_exit()`.

Thread Arguments

» The `pthread_create()` routine permits the programmer to pass one argument to the thread start routine.

– For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread create()` routine.

Thread Argument Example

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

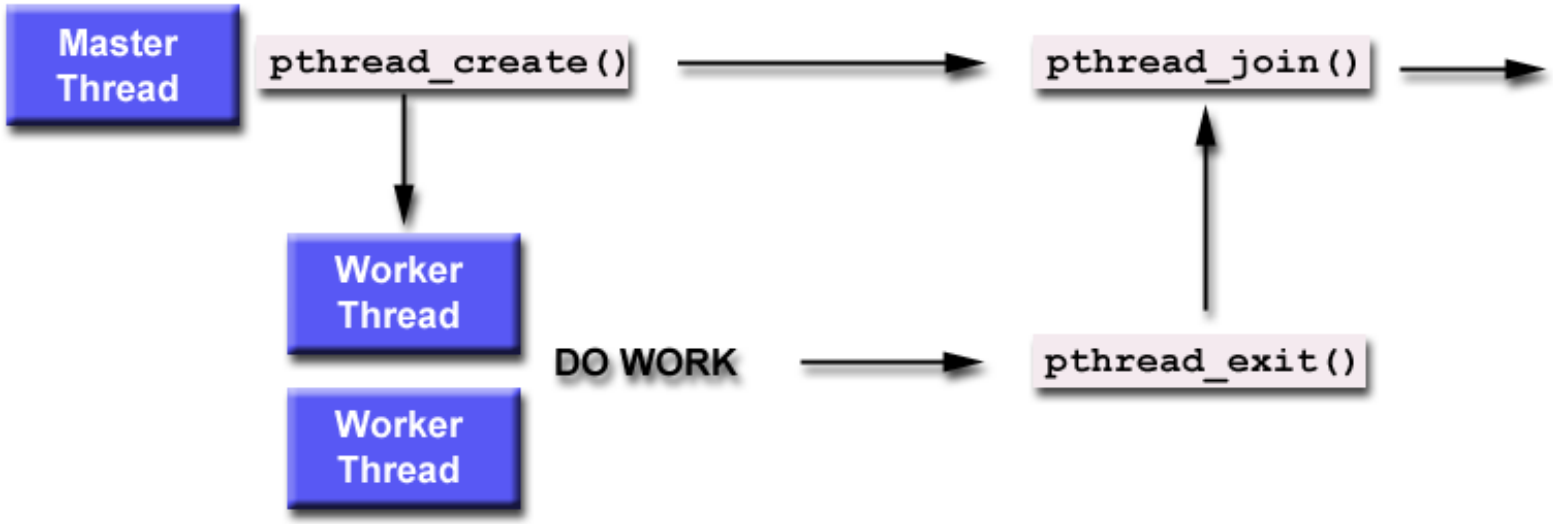
Joining and Detaching Threads

» Joining is one way to accomplish synchronization between threads.

» The `pthread_join()` subroutine blocks the calling thread until the specified threadid thread terminates.

» The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call

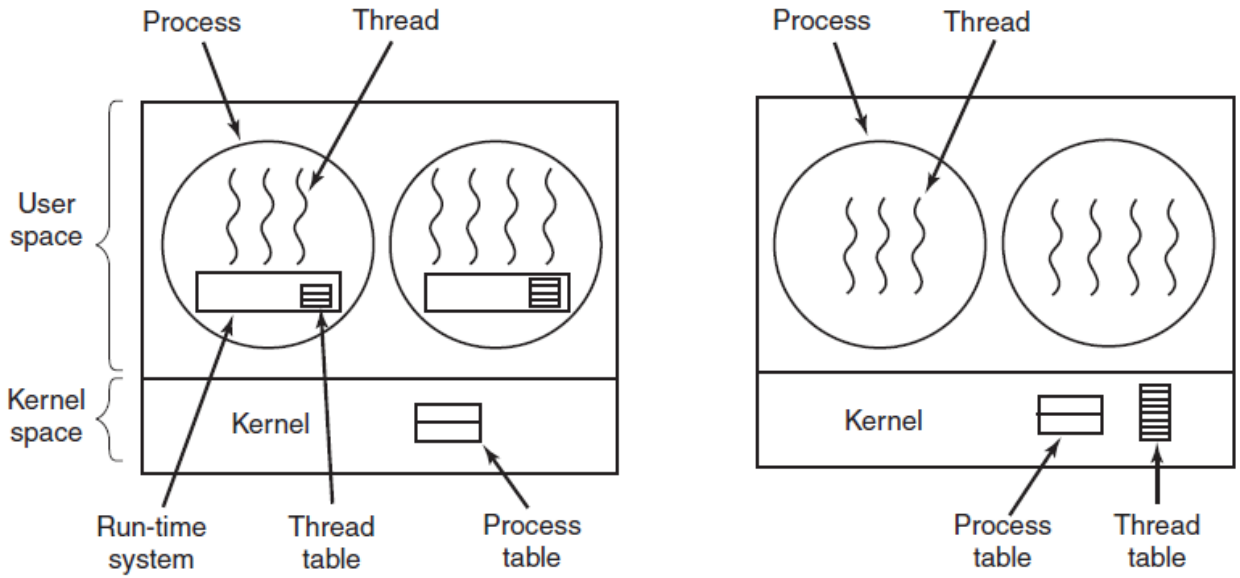
Joining and Detaching Threads



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

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join()
                is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

This example demonstrates how to "wait" for thread completions by using the Pthread join routine.

Implementing Threads



» Two main places to implement threads in an OS:

  » user space

  » kernel space

User Space Threads

- Advantage:
  - OS doesn't need to be thread aware and can be implemented on any OS
- Each process needs its own private thread table to track threads
- Runtime tracks when threads are ready to run or block.

User Space Threads

- Since context switching doesn't involving trapping to the kernel it is at least an order of magnitude faster.
- Each process can have a custom thread scheduling algorithm
- Scale better since kernel threads require table space and stack space in the kernel. Can be a problem with a large number of threads.

User Space Threads

- Some major disadvantages.
  - Blocking calls made by a thread will cause all threads to block.
    - Could wrap all system calls and preface them with `select()`
  - A page fault (discussed in chapter 3) caused by one thread will block all threads.

User Space Threads

- Some major disadvantages continued
  - No preemptive scheduling of threads. Each thread must yield on its own.
  - Threads are most useful in application that block often, for example multithreaded web server.

Kernel Space

- Kernel maintains a thread table.
  - Creating / destroying a thread requires a kernel call.
  - Kernel table holds threads registers, state and other information
- All calls that might block are implemented as system call, at a greater cost.

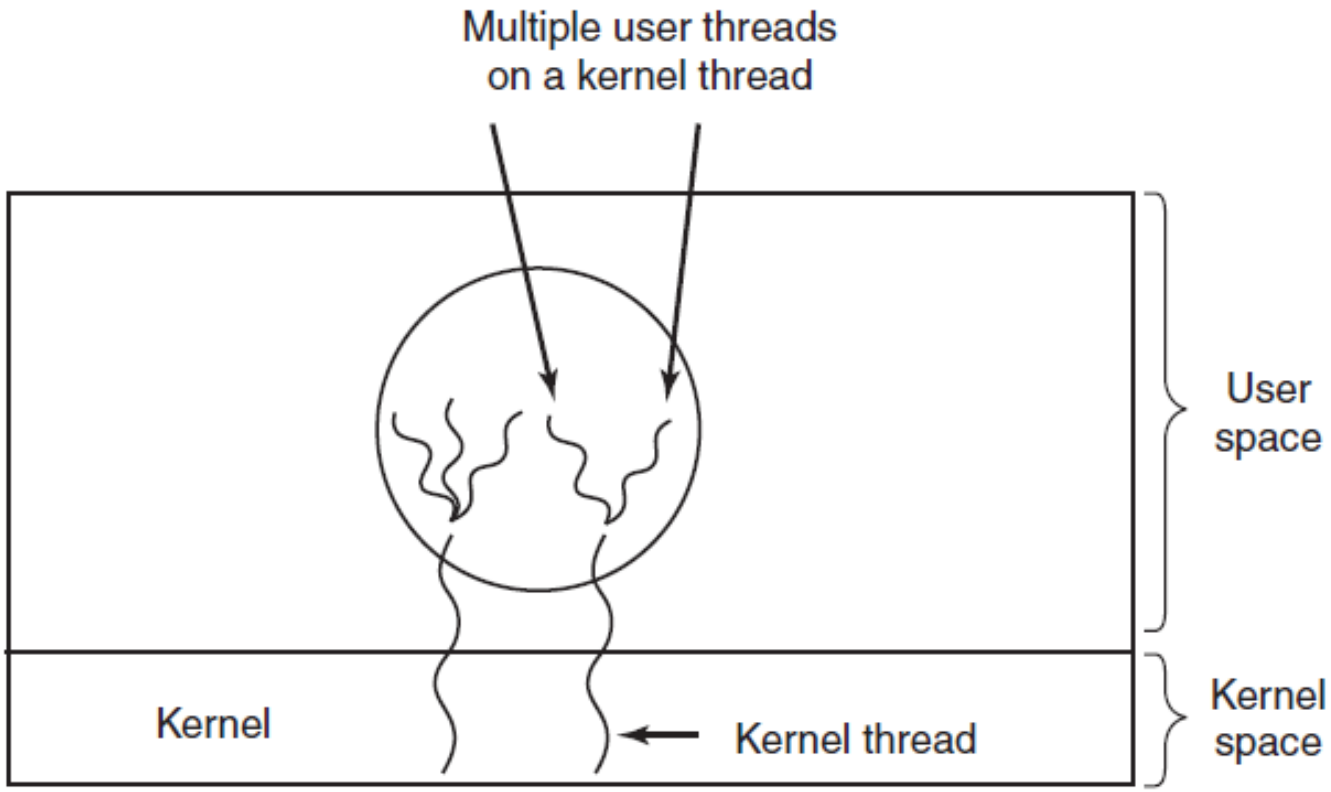
Kernel Space

- When a thread blocks, the OS can choose to run:
  - A ready thread from the current process
  - A ready thread from another process
- Due to large cost of creating a kernel thread, OS can recycle threads.

Kernel Space

- Some problems:
  - What happens when a process forks?
    - Kernel copies each parent threads?
    - Kernel implements one thread?
  - Signals are sent to processes. Which thread handles it?

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

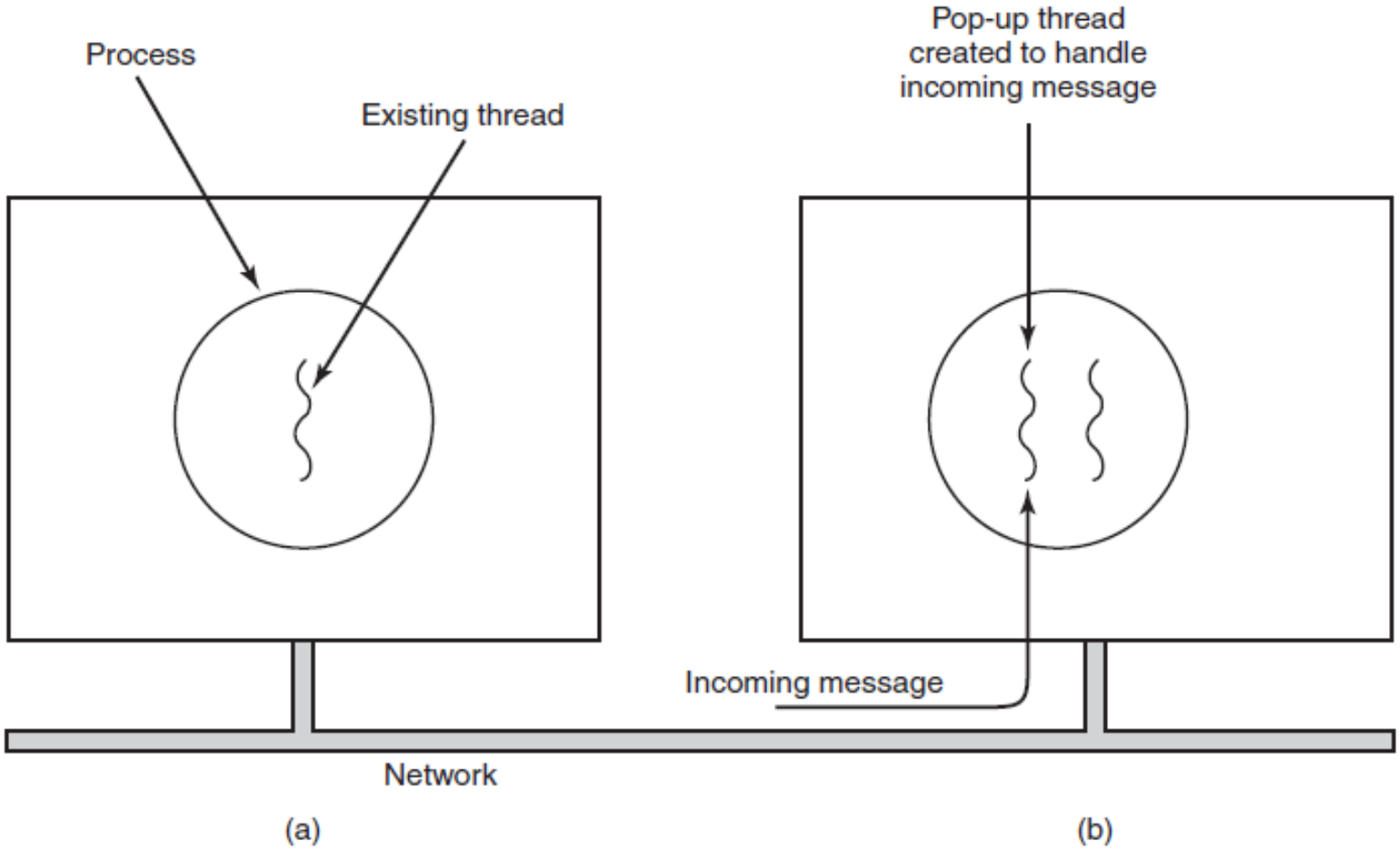


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)

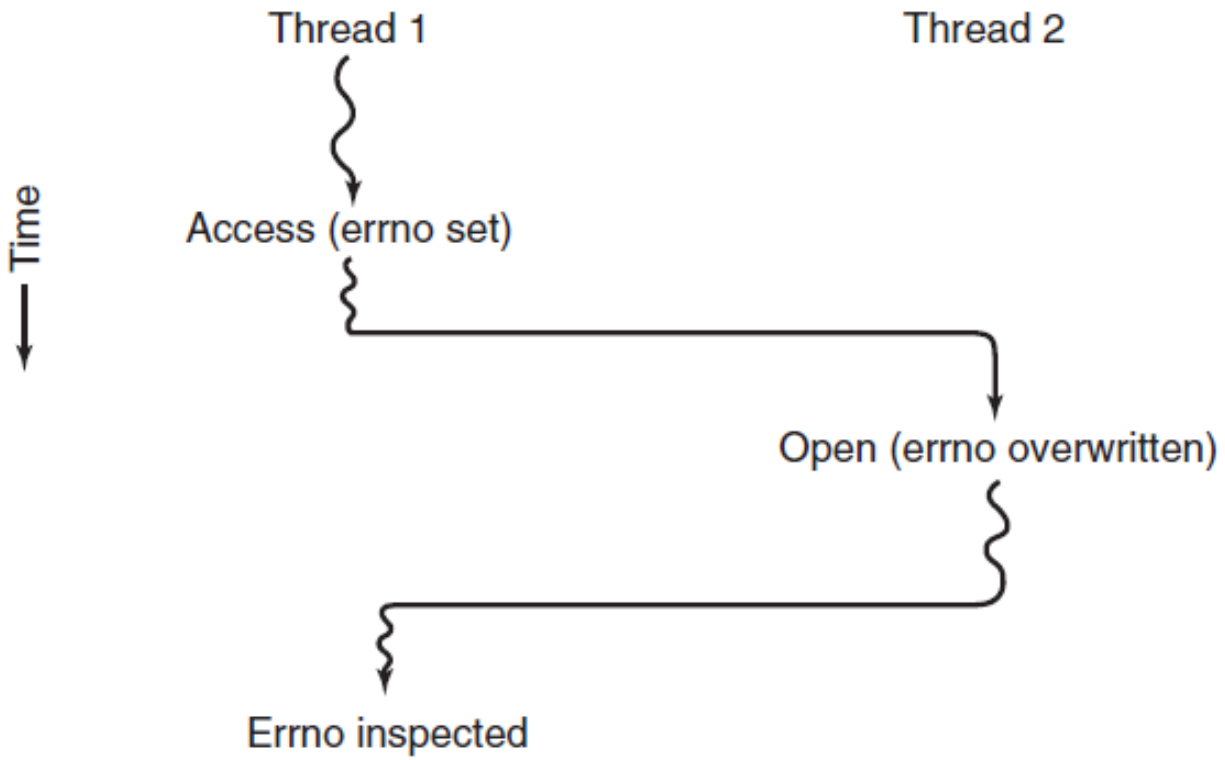


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2)

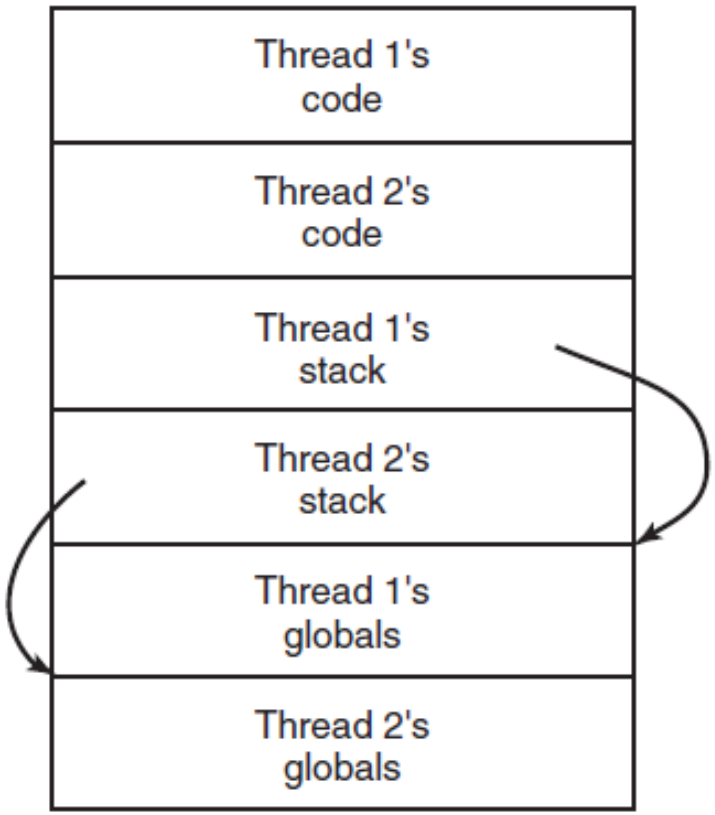


Figure 2-20. Threads can have private global variables.