

2.1 Processes:

- Multiprocessors have two or more CPUs sharing the same physical memory
- **Process Model:**
 - Rapid switching back and forth is known as multiprogramming
 - there is only one program counter so when the process is running the logical program counter is loaded into the real program counter
 - then saved in the process' stored logical program counter in memory

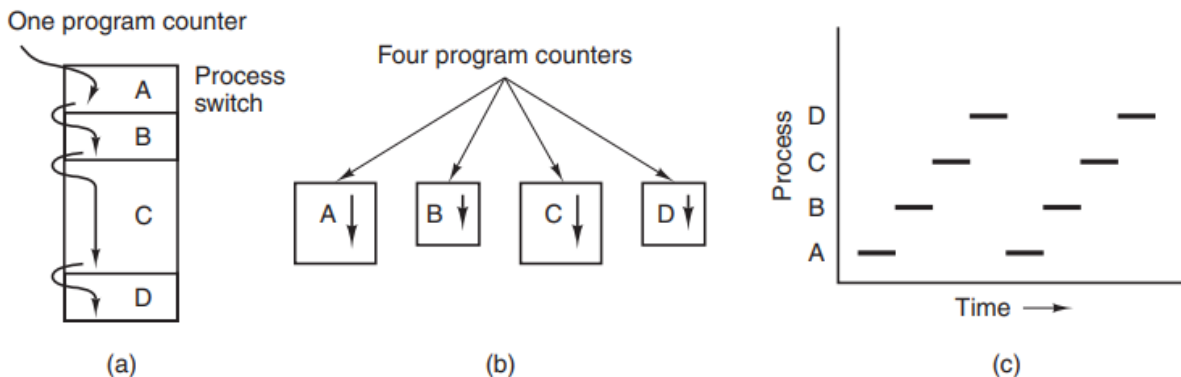


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

- processes must not be programmed with built in assumptions about timing
- processes have a program, input, output and a state, whereas a program is something that may be stored on a disk
- If a program is running twice, it counts as two processes... meaning if there are two documents that you would like to print and two printers available and done at the same time, it is two processes
- **Process Creation:**
 - Four principal events cause processes to be created:
 - System initialization
 - execution of a process-creation system call by a running process
 - Aa user request to create new process
 - initiation of a batch job
 - Daemons: processes that stay in the background to handle some activity such as email, web pages, news, and printing
 - fork creates an exact clone of the calling process
 - they have the same memory image
 - changing the memory image of the child fork must be done with `execve`
 - in both unix and windows, after a process is created, the parent and child have their own distinct address space, if a process changes a word the other process is unaware of the change
 - no writable memory is shared between these processes
 - child may share all of the parent's memory but the memory is shared copy-on-write meaning that whenever either of the two wants to modify part of the memory, that chunk is explicitly copied first to make sure the modification occurs in a private memory area
- **Process Termination:**

- **Processes will terminate due to the given conditions:**
 - normal exit (voluntary)
 - error exit (voluntary)
 - fatal error (involuntary) - often due to a program bug
 - killed by another process (involuntary) - kill call in unix and the process killing must have authorization to do so
- **Process Hierarchies:**
 - child processes can only have one parent
 - the parents and children form a process group and this allows for the processes to individually catch signals, ignore, or take default action (killed)
 - init - the root process of all processes
 - parents are given special tokens, known as handles, that are used to control the child
- **Process States:**
 - processes may generate some output that another process will use as input:
cat chapter1 chapter2 chapter3 | grep tree
concatenates and outputs three files, and running grep selects all lines containing the word "tree"
 - grep maybe ready to run but there is no input waiting for it which results in a block
 - blocking: cannot logically continue, waiting for input that is not available yet OR the process is stopped to allocate for another process
 - **Three process states:**
 - Running (actually using CPU)
 - Ready (runnable; temporarily stopped to let another process run)
 - Blocked (unable to run until some external event happens)

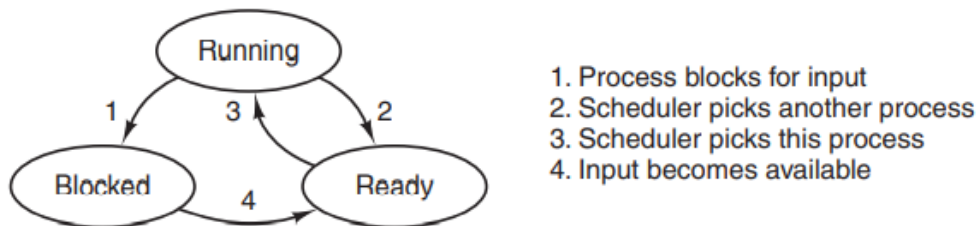


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

- Process scheduler: part of the operating system without the process even knowing about them

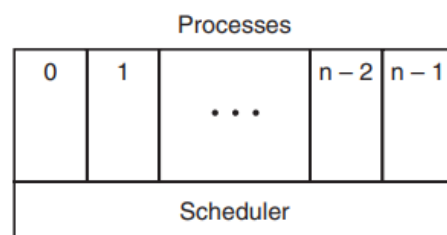


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

- **Implementation of Processes**

- Process table: an array of structures with one entry per process
 - contains important information like process' state, program counter, stack pointer, memory allocation, status of its open files, account and scheduling, important registers, and anything else and is switched to either running, blocked, or ready

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry.

- Interrupt vector: contains the address of the interrupt service procedure, when a procedure is interrupted all the information associated at the time is pushed onto this

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

- **Modeling Multiprogramming:**

- lets processes use the CPU when it would otherwise become idle
- example: a computer has 8gb of memory, and os and tables take up two and user programs take up 2gb. This allows for three user programs to be in memory at once, with an 80% average I/O wait, meaning our CPU utilization is $1 - 0.8^3$ which is roughly 49% usage, more gb would allow us to have a larger utilization figure. say 8gb added, this jumps it to 79% utilization

2.2 Threads

- **Thread Usage:**

- threads are known as microprocesses
- threads have the ability for the parallel entities to share an address space and all of its data among themselves

- easier and faster to create and destroy than a regular process
- when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, speeding up the application
- useful on systems with multiple CPUs

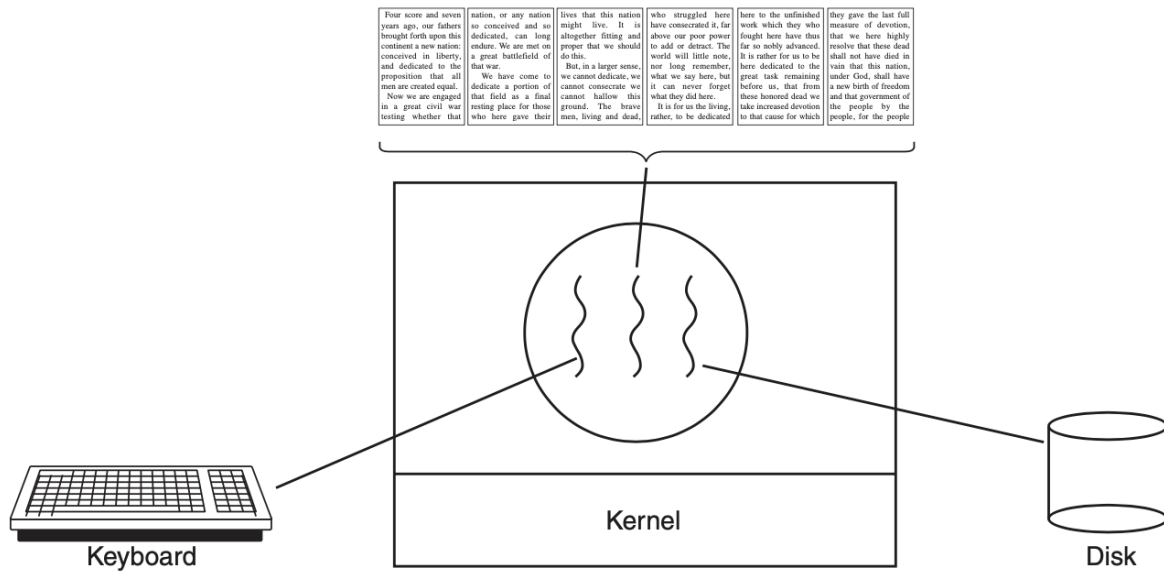


Figure 2-7. A word processor with three threads.

- If this were a single thread, any time a backup was created on the disk it would block any other process until it was finished appearing sluggish
- These three threads all share common memory and have access to the document being edited
- Caches store data that is heavily used in main memory rather than disk memory for faster retrieval
- **Web server example:**
 - dispatcher thread: reads incoming requests for work from the network
 - worker thread is used and writes a pointer to the message into a special word associated with that thread
 - dispatcher wakes up sleeping worker and moves it from blocked to ready state

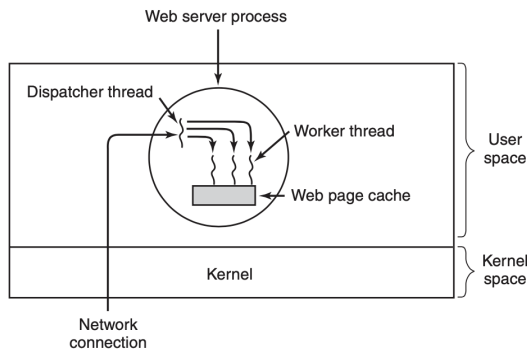


Figure 2-8. A multithreaded Web server.

- Worker checks to see if the webpage was cached, if not a read operation to get the page from the disk is done

Model	Characteristics
Threads	Parallelism, blocking system calls
Single Threaded process	No parallelism, blocking system calls
Finite-State Machine	Parallelism, non blocking system calls, interrupts

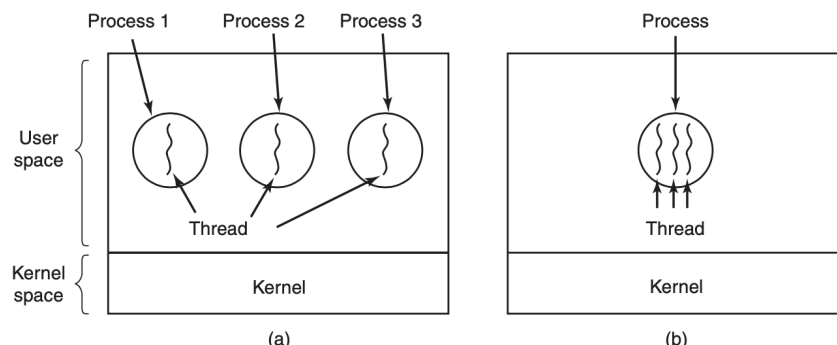
Example of what a thread would look like :

```
while(TRUE){
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while(TRUE){
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page){
        read_page_from_disk(&buf,
            &page);
    }
    return_page(&page);
}
```

- Classical Thread Model:

- Based on two independent concepts: resource grouping and execution
- Processes group related resources together
- Threads belong to a process and execute tasks
 - contains a program counter, registers, stack.
- Processes are used to group resources together; threads are the entities scheduled for execution on the CPU



- threads within the same process share and access all the memory within the process' address space, meaning one can read, write, or wipe out another thread's stack

Per-process items (shared by threads in a process)	Per-thread items (items private to each thread)
Address Space Global variables Open files Child Processes Pending alarms Signals and signal handlers Accounting information	Program Counter Registers Stack State

- Thread's have one of several states: running, blocked, ready, or terminated

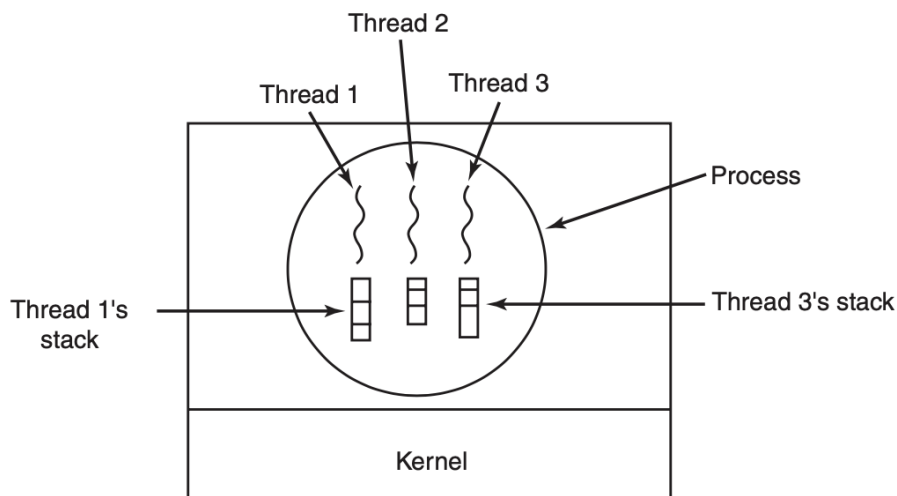


Figure 2-13. Each thread has its own stack.

- Initializing a thread with `thread_create()` begins the thread process
- We can use `thread_exit()` to exit a thread, and `thread_join()` to have a thread wait for a (specific) thread to exit
- `thread_yield()`: allows a thread to voluntarily give up the CPU to let another thread run (no clock interrupt)
- **POSIX Threads:**
 - Pthreads: package that allows you to use threads and contains 60 function calls
 - Important ones to know:

Thread Call	Description
-------------	-------------

Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

- Threads that are waiting for another to terminate use pthread_join() to join the two together, this means whichever one is waiting for the other to finish will call for the join
- Pthread_attr_init: creates the attribute structure associated with a thread and initializes it to the default values, values (such as priority) can be changed by manipulating fields in the attribute structure
- Pthread_attr_destroy: removes a thread's attribute structure, freeing up its memory

Example Code:

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid){
    /*function prints thread's identifier and then exits*/
    printf("Hello world. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]){
    /*the main program creates 10 threads and then exits*/
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i = 0; i < NUMBER_OF_THREADS; i++){
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL,
            print_hello_world, (void *)i);

        if(status != 0){
            printf("Oops. pthread_create returned error code
                %d\n", status);
            exit(-1);
        }
    }
}
```

```

    }
}
exit(NULL);
}

```

- **Implementing Threads in User Space:**

- Threads can be implemented in user space or kernel
- Threads can be placed entirely in the user space without kernel knowledge
 - threads are implemented through a library and is beneficial for machines that do not have multithreading

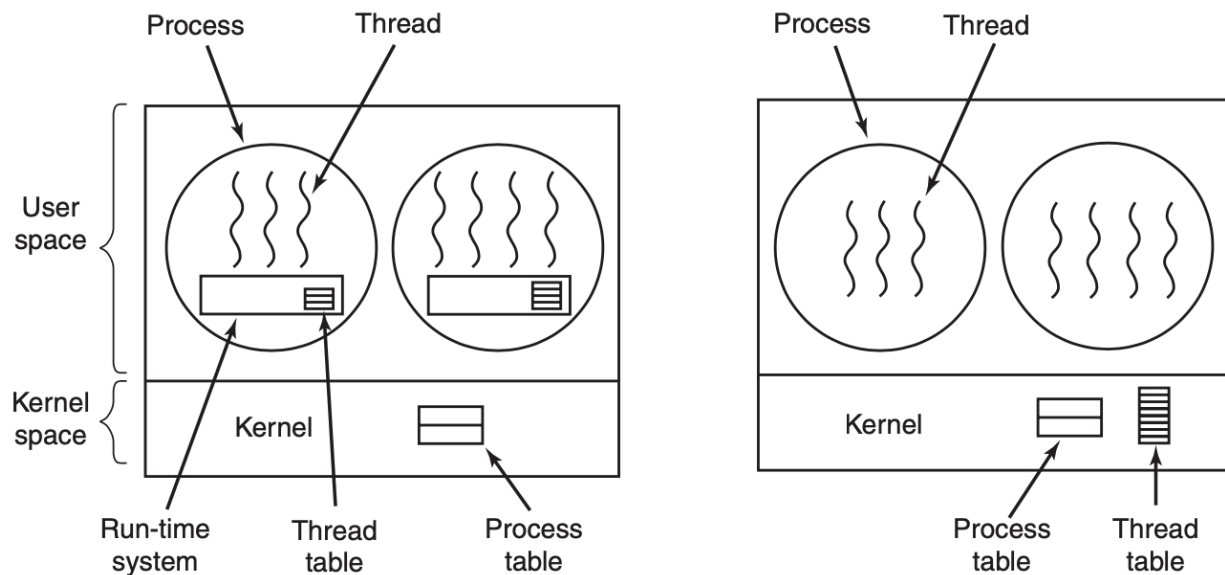


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

- Each process needs its own thread table to keep track of the threads in that process
- this method is faster than trapping into the kernel
- allow for customized scheduling algorithms
- scale better
- Issues:
 - blocking system call implementation
 - Select in unix allows the caller to tell whether a prospective read will block, read can be replaced with a new one that first does a select call and then does read only if safe
 - if a thread starts running, no other thread will process unless the first voluntarily gives up CPU time
- If a program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction

- **Implementing threads in the Kernel:**
 - No run-time system is needed
 - no thread table in each process (kernel tracks thread information)
 - All calls that might block a thread are implemented as system calls
 - Sometimes threads are recycled by being marked as not runnable, leaving the kernel data structure unaffected, and once a new one is needed it is reactivated
 - Issues:
 - when a multi threaded forks, we need to consider what will be done within that fork
 - Signals, which thread will handle the signal
- **Hybrid Implementations:**
 - Use Kernel level threads to multiplex user-level threads onto some or all of them

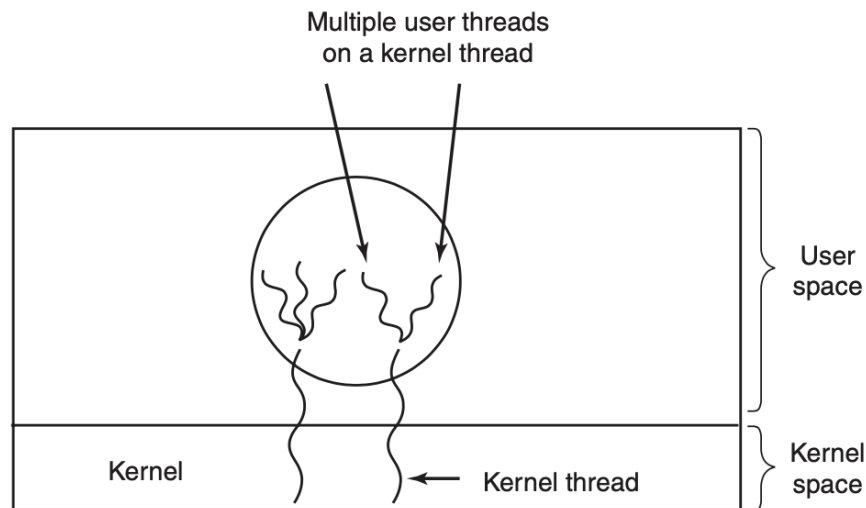
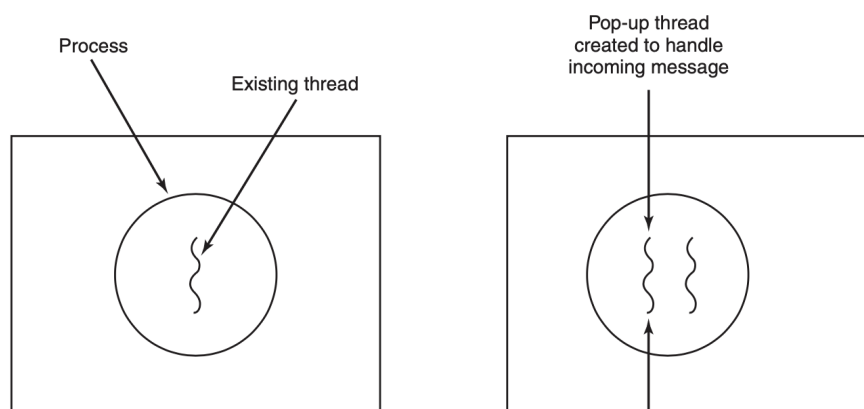


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

- **Scheduler Activations:**
 - Goal: mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space
 - Upcall: Kernel notifies process' run-time system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred, and the kernel activates the run-time system at a known starting address
- **Pop-Up Threads:**
 - When a message causes the system to create a new thread to handle a message
 - latency between message arrival and the start of processing can be made very short



- Usually ran in kernel space, rather than user space because it is faster
- **Making Single Threaded Code Multithreaded:**
 - Can result in us reading data incorrectly because of the timing of execution

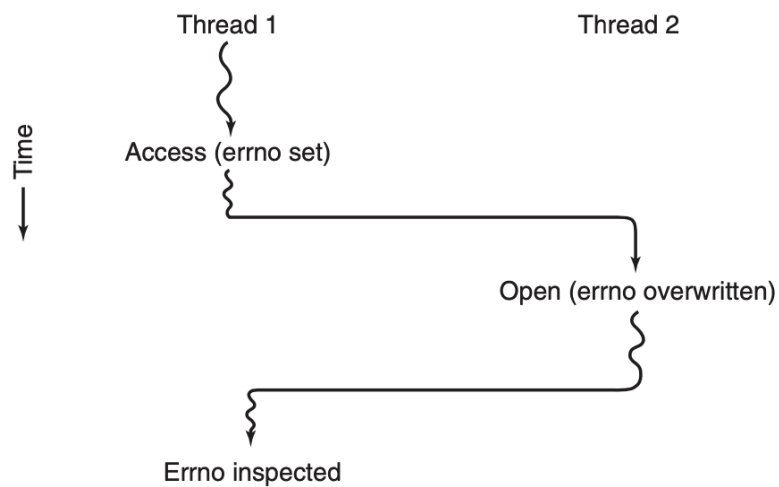
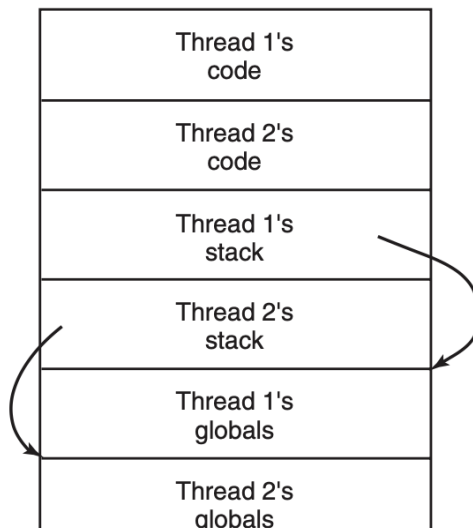


Figure 2-19. Conflicts between threads over the use of a global variable.

- **Solutions:**
 - prohibit global variables
 - assign each thread its own private global variables, allows for each thread to have private access to global variables



- New library procedures can be introduced to create, set, and read these threadwide, global variables: `create_global("bufptr");`
 - allocates storage for a pointer called `bufptr` on the heap or in special storage
 - only calling thread may access this global
 - Two calls are needed to access global variables, one to read, the other to write
 - reading: `set_global("bufptr", &buf);`
 - writing: `bufptr = read_global("bufptr");` returns address
- Library procedures are not reentrant, meaning that once a function is called another is not supposed to be called until the first is finished
- Memory allocation can be impacted as well, like `malloc` which uses crucial tables about memory
- Handling signals appropriately is of great difficulty as well, we are not sure which thread something may need to go to
- Stacks are not dynamic because the kernel is aware and cannot provide them with more memory if needed

2.3 Interprocess Communication

- **Three issues:**
 - how one process can pass information to another
 - two or more processes do not get in each other's way
 - proper sequencing when dependencies are present
- **Race conditions:**

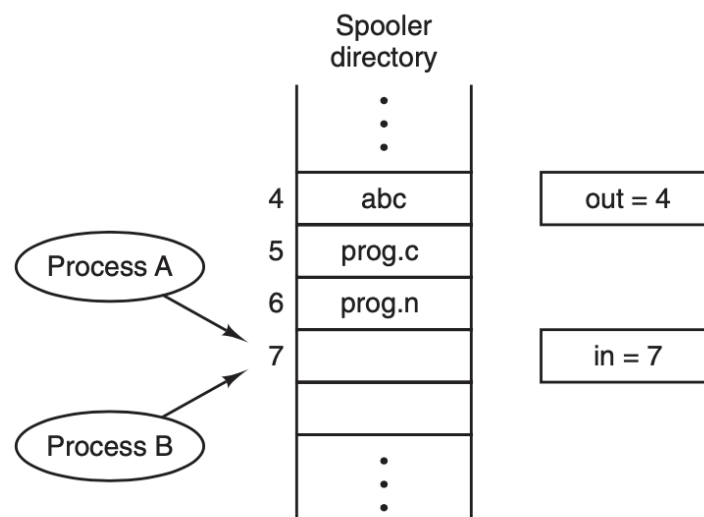


Figure 2-21. Two processes want to access shared memory at the same time.

- Both processes think that 7 is a free spot and are unaware of each other, Process B continues and says 8 is the next free slot, but when A runs again it will overwrite the data in 7 and that information will be lost
- Race conditions: where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- **Critical Regions:**
 - Mutual exclusion: some way of making sure that if one process is using a shared variable or file, the other process will be excluded from doing the same thing
 - Critical region: place where shared memory is accessed
 - Four conditions to hold:
 - No two processes may be simultaneously inside their critical regions
 - No assumptions may be made about speeds or the number of CPUs
 - No process running outside its critical region may block any process
 - No process should have to wait forever to enter its critical region

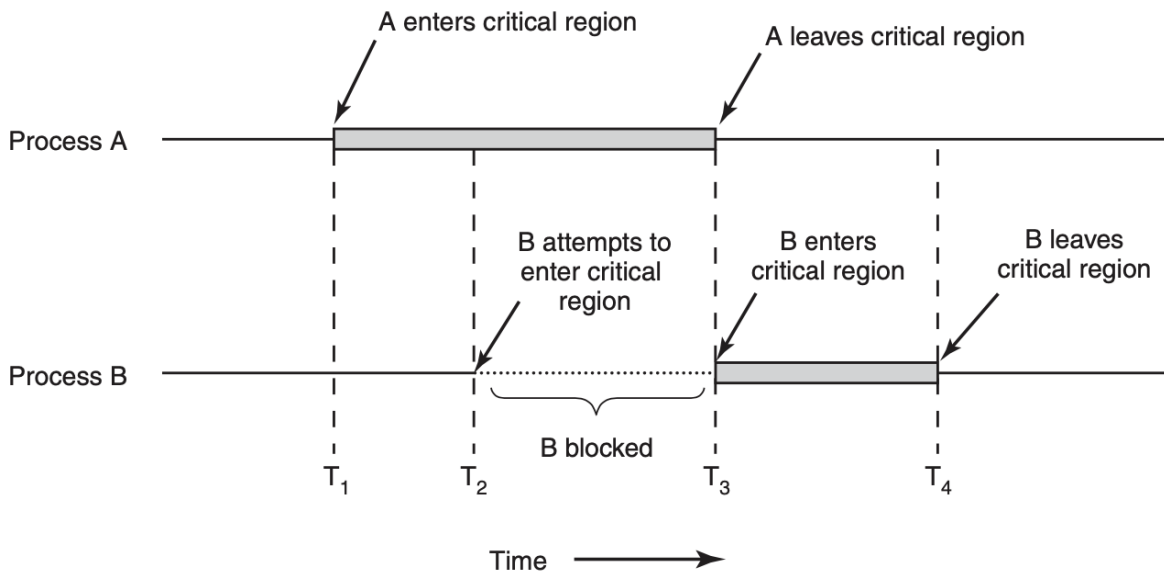


Figure 2-22. Mutual exclusion using critical regions.

- **Mutual Exclusion with Busy Waiting:**
 - **Disabling interrupts:**
 - simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it
 - Not a great idea, because we shouldn't give user processes this type of power
 - **Lock Variables:**
 - Uses a single shared lock variable that is initially 0
 - if a process wants to enter it has to check the lock first
 - if 0 the process enters and marks the variable as 1 (locked)
 - 0 means no process, 1 means process is in critical region
 - This method has its own race condition and could possibly override data

- **Strict Alternation:**

```

while(TRUE){
    while (turn  $\neq$  0){
        critical_region();
        turn = 1;
        noncritical_region();
    }
}

while(TRUE){
    while(turn  $\neq$  1){
        critical_region();
        turn = 0;
        noncritical_region();
    }
}

```

- This is considered busy waiting because we are constantly testing a variable
 - known as a spin lock
 - Not a good approach because if one process is faster than another it may set the turn to 1 and get stuck

- **Peterson's solution:**

- Consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used

Peterson's solution in code:

```

#define FALSE 0
#define TRUE 1
#define N 2          /*number of processes*/

int turn;             /*who's turn is it?*/
int interested[N];    /*all values initially 0*/

void enter_region(int process){    /*process is 0 or 1*/
    int other;                    /*number of the other process */

    other = 1 - process;          /*the opposite of process */
    interested[process] = TRUE;   /*show that you are interested*/
    turn = process;               /*set flag*/
    /*null statement*/
    while (turn == process && interested[other] == true);
void leave_region(int process){    /*process: who is leaving*/
    /* indicate departure from critical region*/
    interested[process] = FALSE;
}

```

- How does this work?:
 - process 0 calls enter_region -> indicates it is interested and sets turn to 0 -> process 1 not interested, return immediately. IF process one calls enter_region, it will just stay there until interested[0] goes to FALSE

- **TSL Instruction:**

- Test and Set Lock reads contents from the memory word lock into register RX and then stores a nonzero value at the memory address lock (TSL RX, LOCK)
- CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done
- When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory, when done the process sets the lock back to 0, this effectively locks the bus from other processes

Example with TSL in Assembly:

```
enter_region:
    TSL REGISTER, LOCK    |copy lock to register and set lock to 1
    CMP REGISTER, #0      |was lock zero?
    JNE enter_region      |if it was not zero, lock was set, so loop
    RET                   |return to caller; critical region entered
```

```
leave_region:
    MOVE LOCK, #-         |store a 0 in lock
    RET                   |return to caller
```

- XCHG may be used alternatively which exchanges the contents of two locations atomically

Example with XCHG in Assembly:

```
enter_region:
    MOVE REGISTER, #1     |put a 1 in the register
    XCHG REGISTER, LOCK   |swap the contents of the reg and lock variable
    CMP REGISTER, #0      |was lock zero?
    JNE enter_region      |if it was non zero, lock was set, so loop
    RET                   |return to caller; critical region entered
```

```
leave_region:
    MOVE LOCK, #0         |store a 0 in lock
    RET                   |return to caller
```

- **Sleep and Wakeup:**
 - Priority inversion problem: Some low priority process may be in its critical region and one that is high priority may be busy waiting, but since the low priority process is never scheduled while high is running, low never gets the chance to leave its critical region and high loops forever
 - sleep: causes caller to block, until another process wakes it up
 - Producer-Consumer Problem:

Example Code:

```
#define N 100
int count = 0;
```

```
void producer(void){
```



```

typedef int semaphore;                                /*semaphores are a special int*/
semaphore mutex = 1;                                  /*controls access to cr*/
semaphore empty = N;                                  /*counts empty buffer slots*/
semaphore full = 0;                                   /*counts full buffer slots*/

void producer(void){
    int item;

    while(TRUE){                                     /*TRUE is the constant 1*/
        item = produce_item();                       /*generate something for buffer*/
        down(&empty);                                 /*decrement empty count*/
        down(&mutex);                                 /*enter critical region*/
        insert_item(item);                            /*put new item in buffer*/
        up(&mutex);                                   /*leave critical region*/
        up(&full);                                    /*increment count of full slots*/
    }
}

void consumer(void){
    int item;

    while(TRUE){
        down(&full);                                  /*decrement full count*/
        down(&mutex);                                 /*enter critical region*/
        item = remove_item();                         /*take item from buffer*/
        up(&mutex);                                   /*leave critical region*/
        up(&empty);                                   /*increment count of empty slots*/
        consume_item(item);                           /*do something with the item*/
    }
}

```

- Binary semaphores: semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time
- Can hide interrupts by initially setting semaphore to 0 associate with each I/O device
 - after starting, the managing process does a down on the associated semaphore
 - when the interrupt handler comes, up is applied to the associated semaphore
- mutex semaphore is used for mutual exclusion and guarantees one process will be reading or writing the buffer
- Synchronization: producer ensures to stop when buffer is full, consumer when buffer is empty
- **Mutexes:**
 - shared variable that can be in one of two states: locked or unlocked
 - mutex_lock: a thread or process needs to access a critical region

- mutex_unlock: a thread or process leaves a critical region
- when mutex_lock fails to acquire a lock, it calls thread_yield giving up CPU time to another thread, there is no busy waiting
- **Futexes:**
 - “Fast user space mutex”, a feature that implements basic locking but avoid dropping into the kernel unless it really has to
- **Mutexes in Pthreads:**

Thread Call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

- Condition Variables: allow threads to block due to some condition not being met
 - in producer-consumer we use condition variables to let the producer block and be awakened later
 - do not have memory like semaphores do, meaning that if a signal is sent to a condition variable on which no thread is waiting the signal is lost

Thread Call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake them all up

- the statement that puts a thread to sleep should always check the condition to make sure it is satisfied before continuing
- **Monitors:**
 - collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package

Example of threads to solve producer-consumer problem:

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

#define MAX 1000000000 /*how many numbers to produce*/
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /*used for signaling */
int buffer = 0; /*buffer used between p and c*/

void *producer(void *ptr){
    int i;

    for(i = 1; i ≤ MAX; i++){
        pthread_mutex_lock(&the_mutex); /*get exclusive access to
                                         buffer*/
        while(buffer ≠ 0)pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /*put item in buffer*/
        pthread_cond_signal(&condc); /*wake up consumer*/
        pthread_mutex_unlock(&the_mutex); /*release access to buffer*/
    }
    pthread_exit(0);
}

void *consumer(void *ptr){
    int i;

    for(i = 1; i ≤ MAX; i++){
        pthread_mutex_lock(&the_mutex); /*get exclusive access to
                                         buffer*/
        while(buffer == 0)pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /*take item out of buffer*/
        pthread_cond_signal(&condp); /*wake up producer*/
        pthread_mutex_unlock(&the_mutex); /*release access to buffer*/
    }
    pthread_exit(0);
}

int main(int argc, char **argv){
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
```

```

pthread_create(&con, 0, consumer, 0);
pthread_create(&pro, 0, producer, 0);
pthread_join(pro, 0);
pthread_join(con, 0);
pthread_cond_destroy(&condc);
pthread_cond_destroy(&condp);
pthread_mutex_destroy(&the_mutex);
}

```

- Conditional variables can also use wait and signal
 - when the buffer may be full, it does a wait on some condition variable and causes the calling process to block
 - consumer can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on
 - signal must exit the monitor immediately leaving it to be the last statement in a monitor procedure
- **Message Passing:**
 - interprocess communication that uses two primitives, send and receive, which like semaphores and unlike monitors, are system calls rather than language constructs
 - send(destination, &message);
 - receive(source, &message);
 - **The Procedure-Consumer Problem with Message Passing:**
 - Mailbox: place to buffer a certain number of messages, typically specified when the mailbox is created

Example Producer Consumer Problem with Message Passing

```
#define N 100
```

```

void producer(void){
    int item;
    message m;

    while(TRUE){
        item = produce_item();          /*generate something to put in buffer*/
        receive(consumer, &m);          /*wait for an empty to arrive*/
        build_message(&m, item);         /*construct a message to send*/
        send(consumer, &m);              /*send item to consumer*/
    }
}

```

```

void consumer(void){
    int item, i;
    message m;

    for(i = 0; i < N; i++) send(producer, &m); /*send N empties*/
    while(TRUE){

```

```

    receive(producer, &m);           /*get message containing item*/
    item = extract_item(&m);         /*extract item from message*/
    send(producer, &m);              /*send back empty reply*/
    consume_item(item);               /*do something with the item */
}
}

```

- Barriers:

- Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready

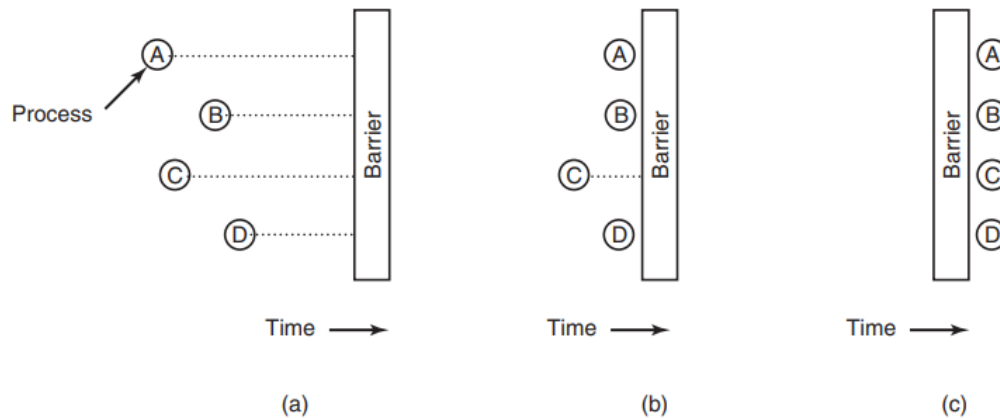
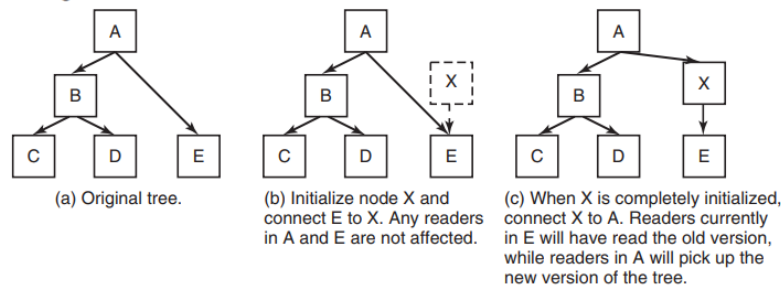


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

- Avoiding Locks: Read-Copy-Update

- We must decide if the reader will read new or old data

Adding a node:



Removing nodes:

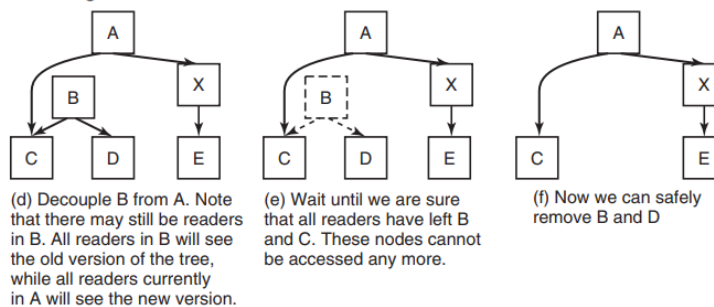


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks.

2.4 Scheduling