Processes and Threads

Chapter 2

Seyda Özer

The Process Model (1)

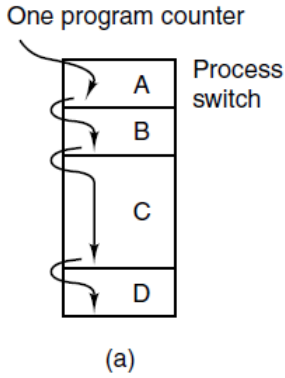


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (2)

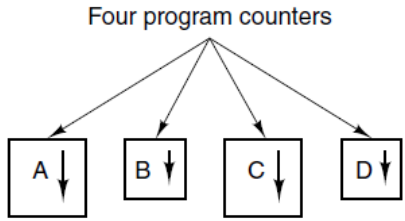


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (3)

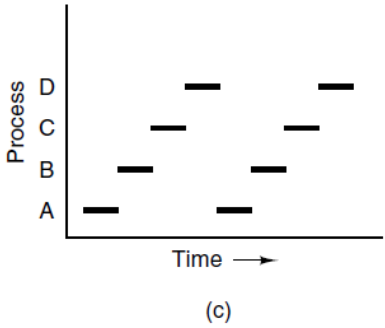


Figure 2-1. (c) Only one program is active at once.

Process Creation

Four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Process Termination

Typical conditions which terminate a process:

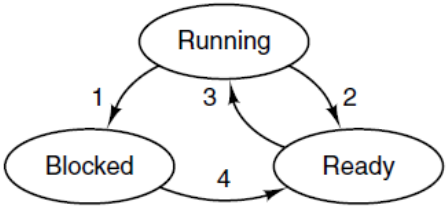
1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

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).

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.

Process States (3)

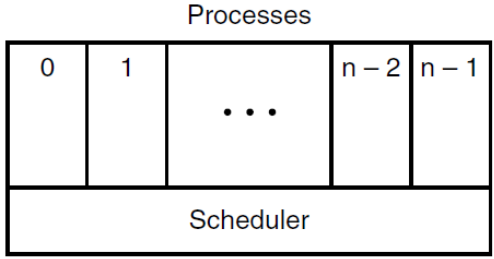


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Process States (3)

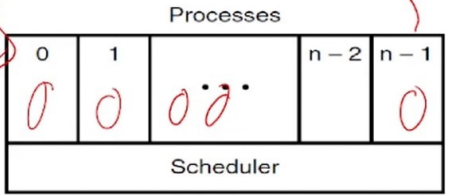


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Here the lowest level of the operating system is the **scheduler**, with a variety of processes on top of it.

All the interrupt handling and details of actually starting and stopping processes are hidden away in what is here called the **scheduler**, which is actually not much code.

Implementation of Processes

- A table (***an array of structures***), called the process table, with one entry per process.
- This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

Implementation of Processes (1)

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.

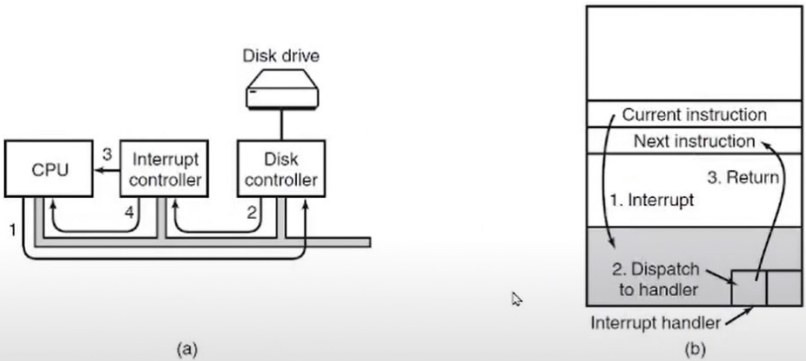


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Implementation of Processes (2)

Suppose that a user process is running when a disk interrupt happens.

1. Hardware stacks program counter, program status word etc. (not all registers, why?)
2. Hardware loads new program counter from interrupt vector. That is all the hardware does.
3. Assembly-language procedure saves registers. All interrupts start by saving the registers, often in the process table entry for the current process. The temporary values are saved
4. Assembly-language procedure sets up new stack. Saving the registers and setting the stack pointer cannot even be expressed in C!
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. Control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running.

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

Implementation of Processes (2)

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

- Suppose that a process spends a fraction p of its time waiting for I/O to complete.
- With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n.

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

Modeling Multiprogramming

if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%.

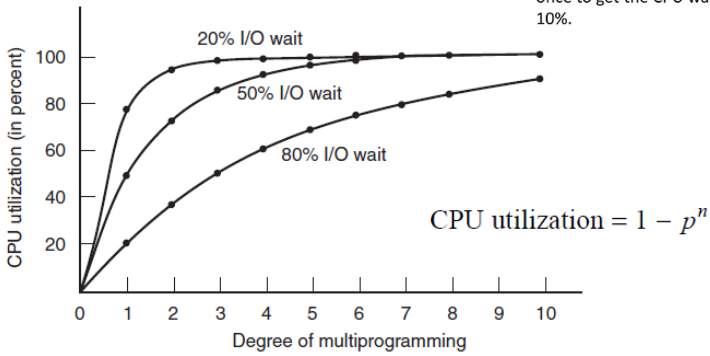


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Modeling Multiprogramming

- A computer has 8 GB of memory,
- OS takes up 2 GB and each user program takes 2 GB.
- 3 programs in memory at once
- With an 80% average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - (0.8)^3$ or about 49%.
- Adding another 8 GB of memory allows seven-way multiprogramming, thus raising the CPU utilization to 79%.
- In other words, the additional 8 GB will raise the throughput by 30%.
- What happens if I add 8 GB more?

Threads

- Like processes: programs in parallel
- Unlike proceses

–Share the same address space and data

–Lighter and faster for use (10 to 100 times faster to create)

–Threads yield no performance gain when all of them are CPU bound

–There is no protection between threads

–Threads co-operate, they do not compete

- The main reason for having threads is that in many applications, multiple activities are going on at once.
- Some of these may block from time to time.
- By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Examples follow

Thread Usage (1)

- One thread interacts with the user
- the other handles reformatting in the background
- The third thread can handle the disk backups without interfering with the other two

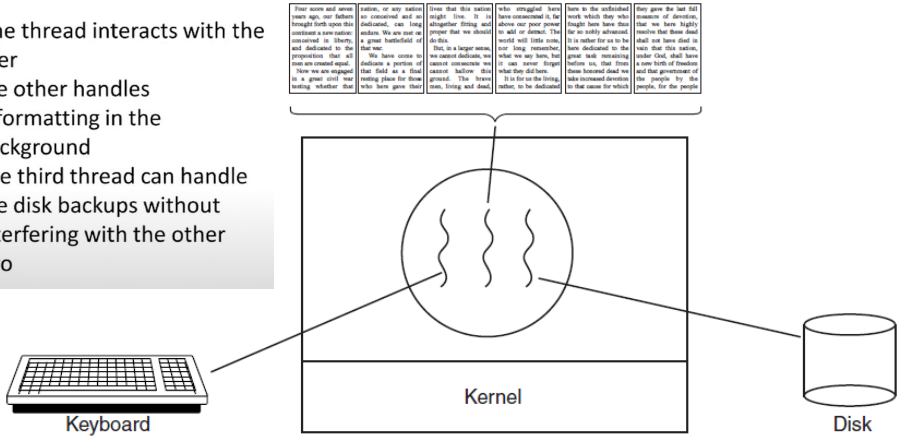


Figure 2-7. A word processor with three threads.

Thread Usage (2)

Here one thread, the dispatcher, reads incoming requests for work from the network.

After examining the request, it chooses an idle (i.e., blocked) worker thread and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread.

The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

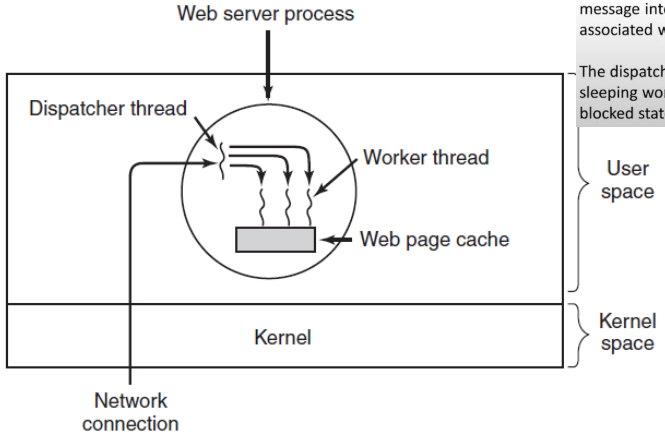


Figure 2-8. A multithreaded Web server.

Thread Usage (3)

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

(a)

```
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);
}
```

(b)

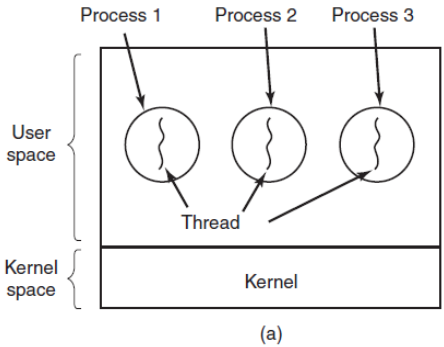
Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

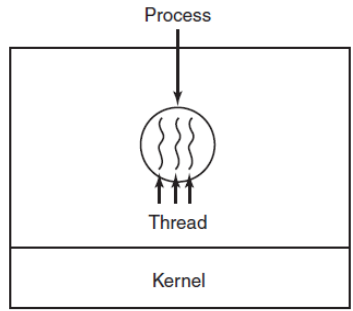
Thread Usage (4)

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-10. Three ways to construct a server.

The Classical Thread Model (1)





The process model is based on two independent concepts: **resource grouping and execution**. We can separate them!
Because threads have some of the properties of processes, they are sometimes called **lightweight processes.**

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

The Classical Thread Model (1.5)

- There is no protection between threads because (1) it is impossible, and (2) it should not be necessary
- All the threads can share the same set of open files, child processes, alarms, and signals, an so on, as shown in Fig. 2-12.
- If one thread opens a file, that file is visible to the other threads in the process and they can read and write it.

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)

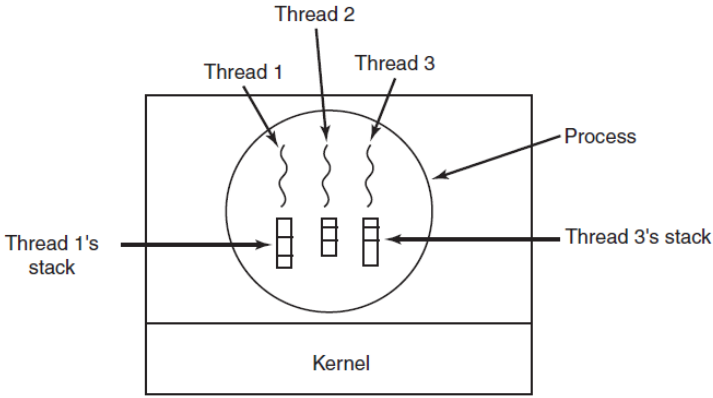


Figure 2-13. Each thread has its own stack.

POSIX Threads (1)

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

- With create_thread It is not necessary (or even possible) to specify anything about the new thread's address space
- thread_yield is important because there is no clock interrupt to actually enforce multiprogramming with the threads as there is with processes.

Figure 2-14. Some of the Pthreads function calls.

POSIX Threads (2)

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the 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);
```

Figure 2-15. An example program using threads.

POSIX Threads (3)

```
    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);
}
```

Figure 2-15. An example program using threads.

Implementing Threads

Three basic strategies

- User-level: possible on most operating systems, even ancient ones
- Kernel-level: looks almost like a process (i.e., Linux, Soloris)
- Scheduler activations (Digital Tru Unix 64, NetBSD, some Mach)

Implementing Threads in User Space

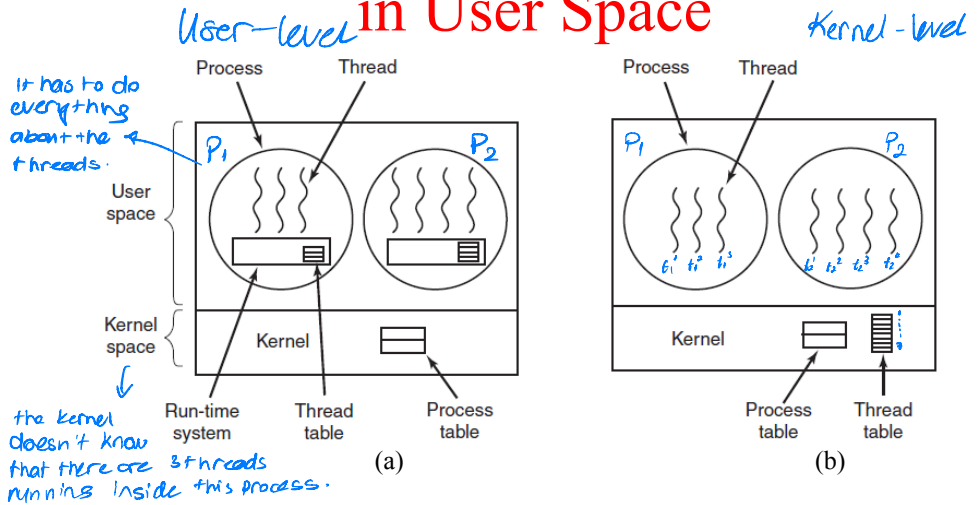


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

User-Level Threads

- Thread creation, destruction, and scheduling done at user level
- To create a thread, must allocate storage for stack, program counter, registers, state, etc. in a thread table
- When a thread yields the CPU or blocks, the thread library saves everything in the thread table
- The thread scheduler finds the highest-priority thread that's ready to run
- Its registers, program counter, etc., are restored from its thread table

Efficiency Gains (User-level)

- No system calls needed
- No context switch
- No copying data safely across a protection boundary
- No need to flush the hardware memory cache

Other Advantages (User-Level)

- Allows each process to have its own customized scheduling algorithm
  - Eg., a garbage collector thread
- Scales better than kernel-level threads
  - Kernel-level threads may require table & stack space in the kernel-level
  - → problematic if when large number of threads

Disadvantages of User-Level Threads

- Blocking system calls
- Page faults
- Inability to benefit from multiprocessor CPUs
- Kernel can't automatically grow per-thread stacks

Blocking System Calls

- What if a thread issues, say, a **read()** call and there's no data available?
- The kernel will make the whole process block because it's unaware of the thread structure
- This is contrary to the purpose of having threads
  - Difficult issue in user level threads → need modifications to OS, e.g. making all system calls to be nonblocking…
  - Another alternative in two slides…

Page Faults

- Part of the virtual memory system; won't say much now
- Briefly, though — a trap occurs because referenced memory isn't available
- No ability to suspend just one thread

Kernel-level Threads

- Kernel manages the thread
  - To create/destroy a thread, the originating thread makes a kernel call
- Similar thread table, but in the kernel
  - Often integrated with process table
- No problem with blocking system calls — ordinary process scheduling does the right thing
- Handles multi-CPU case easily
- But — thread-switching is expensive, since it needs a context switch
- Still cheaper than process switches, since you can use the same virtual memory map

Other Schemes

- Hybrid: some kernel-level threads, with each kernel-level thread supporting several user-level threads
  - –User-level threads are multiplexed to the kernel-level threads
  - –Kernel is aware of only the kernel-level threads and schedules those
- Scheduler activations
- Pop-up threads

Scheduler Activations

- Get the best of both worlds — the efficiency of user-level threads and the non-blocking ability of kernel-level threads
- Relies on upcalls

What's an Upcall?

- Normally, user programs call functions in the kernel
- Sometimes, though, the kernel calls a user-level process to report an event
- Sometimes considered unclean — violates usual layering

Scheduler Activations

- Thread creation and scheduling is done at user level
- When a system call from a thread blocks, the kernel does an upcall to the thread manager
- The thread manager marks that thread as blocked, and starts running another thread
- When a kernel interrupt occurs that's relevant to the thread, another upcall is done to unblock it

Hybrid Implementations

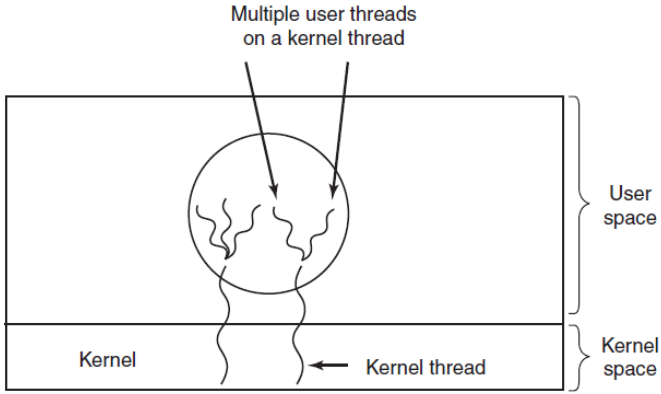


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

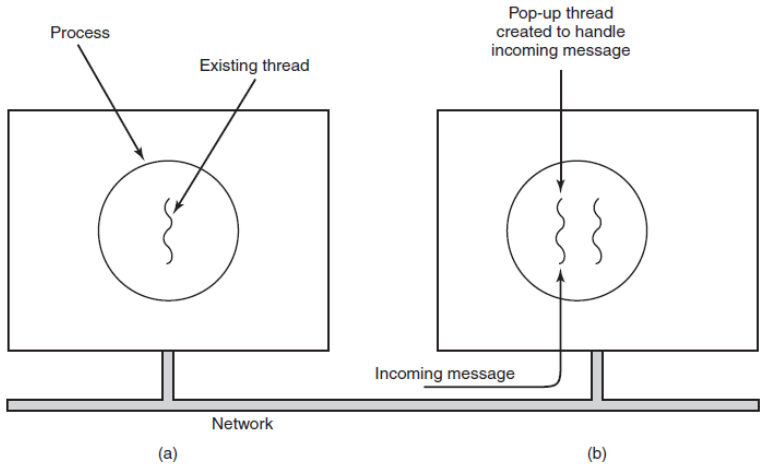


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)

- Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program are a problem.
- One solution is to prohibit global variables altogether.
- Another solution: it is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread as an extra parameter.
- create global("bufptr"); set global("bufptr", &buf); bufptr = read global("bufptr");

Threads share address space so the same memory address space. It is very easy to communicate between the threads.

Making Single-Threaded Code Multithreaded (1)

global int
int errno;

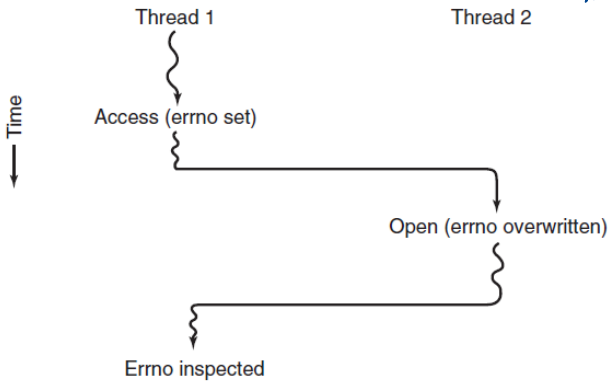


Figure 2-19. Conflicts between threads over the use of a global variable.

Using Threads

- Library routines may be used by more than one thread simultaneously
- Not all routines are prepared for this
- Static variables can be overwritten by another thread
- Must use reentrant routines

What is a Reentrant Routine?

- A routine that can be invoked more than once simultaneously
- Can only use local variables or dynamically allocated variables
- Must not use static variables
- Many C library routines do use static buffers!

Normal and Reentrant Versions

Normal struct tm *localtime(const time t *timep);

Reentrant struct tm *localtime r(const time t *timep, struct tm *result);

The normal version stores the result in a static buffer and passes a pointer back; the reentrant version requires the caller to supply a buffer.

Making Single-Threaded Code Multithreaded (2)

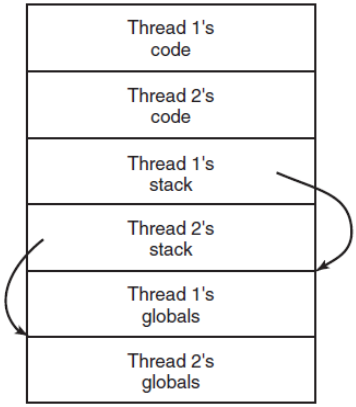


Figure 2-20. Threads can have private global variables.

Interprocess Communication

this is easy for threads why because threads share memory space

There are three issues here.

- The first: how one process can pass information to another. you just modify a global variable.
- The second has to do with making sure two or more processes do not get in each other's way, for example, two processes in an airline reservation system each trying to grab the last seat on a plane for a different customer.
- The third concerns proper sequencing when dependencies are present: if process A produces data and process B prints them, B has to wait until A has produced some data before starting to print. race condition

Second and third ones apply to threads!

Race Conditions

•Where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

•Example spooler

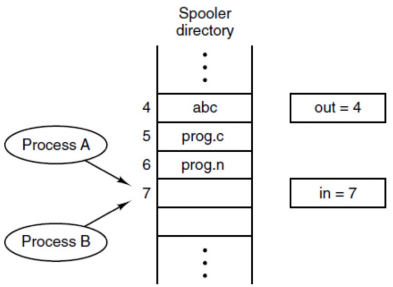


Figure 2-21. Two processes want to access shared memory at the same time.

- Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions
- what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

Race Conditions

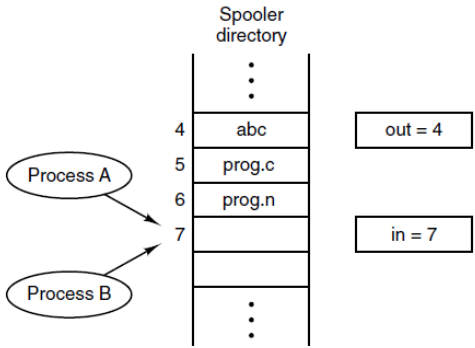
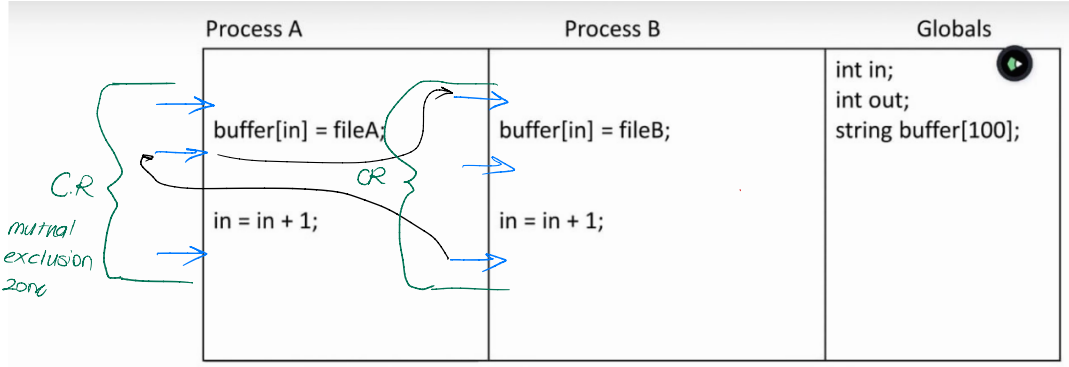
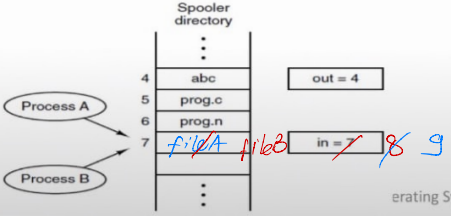


Figure 2-21. Two processes want to access shared memory at the same time.





if I am inside this region while I am executing this no body can use these 3 global variables. Only I can do it.

Critical Regions (1)

- **Mutual exclusion:** some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
- Part of the program where the shared memory is accessed is called the **critical region or critical section.**

Critical Regions (1)

Requirements to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Critical Regions (2)

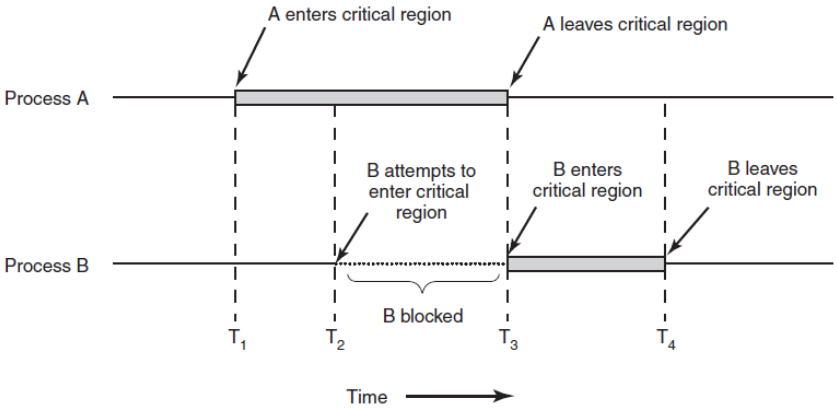


Figure 2-22. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting:

- **Busy waiting**: Continuously testing a variable until some value appears
- Proposals for achieving mutual exclusion:
  - Disabling interrupts
  - Lock variables
  - Strict alternation
  - Peterson's solution
  - The TSL instruction

Disabling Interrupts

- Available only to the kernel
- Better not be available to user processes!
- Doesn't work well on multiprocessors
- A special-case solution for the kernel only

Lock Variables

- How about this?

P1

```
while (lock != 0)
        ;
/* Start critical region */
lock = 1;
...
/* End critical region */
lock = 0;
```

P2

```
while (lock != 0)
        ;
lock = 1;
arr[in] = fileN;
in++;
lock = 0;
```

- Does not work — there is a window between testing for zero and setting it to 1

Let's Use a C Feature: ++

```
while(lock++ != 0)
lock--;
/* critical region ...*/
```

- Is lock++ atomic?
- No — the language makes no guarantees about that!
- IBM mainframe sequence:

```
L R0,lock        Fetch variable into Register 0
LR R1,R0
A R1,=F'1'  Increment it
ST R1,lock  Store new value in 'lock'
C R0,=F'0'  Compare original value to 0
BZ CRITREG       If so, go on...
```

- No atomic "increment" instruction!

Mutual Exclusion with Busy Waiting:

turn

P_0 Strict Alternation P_1

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

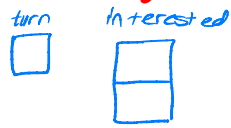
- Continuously testing a variable until some value appears is called busy waiting.
- A lock that uses busy waiting is called a spin lock.

Figure 2-23. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the *while* statements.

Mutual Exclusion with Busy Waiting: Strict Alternation

- Suppose that Process 1's non-critical region code takes a long time to execute
- It won't re-enter its critical region
- Taking turns is not a good idea when one of the processes is much slower than the other.
- This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region.

Mutual Exclusion with Busy Waiting:




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

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

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 */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Mutual Exclusion with Busy Waiting: The TSL Instruction

TSL RX,LOCK (Test and Set Lock) .

- It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.
- The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

Mutual Exclusion with Busy Waiting:
The TSL Instruction (1)

Test and Set Lock


```
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 nonzero, 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
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

Mutual Exclusion with Busy Waiting:
The TSL Instruction (2)

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register 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
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction

Spin Locks

- two correct solutions — Test and Set Lock, Peterson's Algorithm — involve busy waiting
- Also known as a spin lock
- Acceptable for short waits, especially on multiprocessors or when dealing with interrupt contexts
- For general-purpose use, need a solution that permits sleeping

Spin Locks and Priority Inversion

- Suppose we have two processes with different priorities, H and L
- L never runs if H is runnable
- While H is sleeping, the low priority process L grabs the spin lock
- H wakes up and tries to get the lock
- It spins, waiting for L to free it
- But L can't get the CPU, so it doesn't progress
- One example of a deadlock

Blocking instead of Busy Waiting

- Prevent wasting time busy waiting
- Simplest way is sleep+wakeup…
  - System call sleep causes caller to block (suspend until another process wakes it up)
  - Alternatively, sleep+wakeup both have a memory address to match them…
- Let's use this in an example

Classic Problem: Producer-Consumer

- Producer wants to add elements to bounded-size buffer
- If buffer is full, producer must sleep
- Consumer wants to remove items from buffer
- If buffer is empty, consumer must sleep
- Example: print spoolers

Sleep and Wakeup

The Producer-Consumer Problem (1)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Sleep and Wakeup
The Producer-Consumer Problem (2)

```
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                                  /* repeat forever */
                if (count == 0) sleep( );               /* if buffer is empty, got to sleep */
                item = remove_item( );                  /* take item out of buffer */
                count = count – 1;                      /* decrement count of items in buffer */
                if (count == N – 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);                     /* print item */
        }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Semaphores

≡ int

just like an actual integer
So think of a semaphore like an integer

- Invented by Dijkstra in 1965
- Two operations: down (sometimes known as P) and up (also known as V)
- **down**: decrements semaphore variable if greater than zero; if 0, process sleeps before doing the decrement (**sleep**)
- Note: check, decrement, and sleep are atomic
- **up**: increments semaphore; if a process was sleeping on it, wake it and let it do its decrement (**wakeup**)
- Generalizes well to multiple users of the semaphore

Using Semaphores

```
semaphore mex = 1;
down(&mex);
/* critical region */
up(&mex);
```

Implementing Semaphores

- Typically done in the kernel
- Mask interrupts while manipulating semaphore
- On multiprocessors, use Test and Set Lock protection as well
- Both of these are for only a few microseconds — not a serious problem

Semaphores (1)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
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 to put in 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)
```

Figure 2-28. The producer-consumer problem using semaphores.

Semaphores (2)

```
            up(&full);                          /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
            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 */
        }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Two Different Uses of Semaphores

- Counting semaphores — atomic way to manipulate a counter with blocking if 0
  - In the example, empty counts how many slots are free and full is the number of slots that are full
  - Used for sleep/wake when buffer is in the wrong state: **synchronization**
- Mutual exclusion semaphores — make sure only one process is in critical region
  - In this case, it protects access to the buffer
  - (Probably, that code should be in the insert item()/ remove item() routines)

Mutexes

- Special case of semaphore: useful when no counting is needed: only a bit is enough
- If Test and Set Lock instruction is available, easy to implement at user level for thread package
- Try to grab lock; if unsuccessful, let another thread run
- If failed to acquire a lock, it calls thread yield to give up the CPU to another thread. Consequently there is no busy waiting

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

Different Processes?

- If processes have disjoint address spaces, how can they share semaphores or a common buffer?
- one: shared vars are in kernel
- Two: Many OSes offer a way for processes to share some portion of their address space with other processes

Futex (Fast User Space Mutex)

- Busy Wait or block: which is faster?
- Block only if necessary and switch to kernel space.
- The kernel service provides a ‘‘wait queue’’ that allows multiple processes to wait on a lock.
- They will not run, unless the kernel explicitly unblocks them.
- Read the details from the book

Mutexes in Pthreads (1)

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

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Condition Variables

- Pthreads offer a second synchronization mechanism: condition variables.
- Mutexes are good for allowing or blocking access to a critical region.
- Condition variables allow threads to block due to some condition not being met.
- Almost always the two methods are used together.

Mutexes in Pthreads (2)

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 all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#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 producer and consumer */

void *producer(void *ptr)                       /* produce data */
{     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);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (4)

```
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       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)
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (5)

```
    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);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

The Risks of Semaphores and Mutexes

- Using semaphores and mutexes correctly is difficult
- If you get it wrong, you can deadlock
- Testing is difficult, because it's all timing-dependent
- We need a higher-level construct

Monitors

- Invented by Hoare (1974) and Brinch Hansen (1975)
- A programming language construct
- Java has it; C and C++ do not
- A monitor is like a class, but only one thread can be executing in it at a time
- In other words, they're a language implementation of mutexes

Monitors

- Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.
- Introduction of condition variables, along with two operations on them, wait and signal.
- When a monitor procedure discovers that it cannot continue, it does a wait on some condition variable, say, full.
- This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Figure 2-33. A monitor.

Monitors and Application Blocking

- We still need a way to block if, say, the buffer is full
- Two new operations: wait and signal
- Subtle semantics about who gets to run within the monitor after a signal call
  - Hoare: signaler blocks; waiter runs
  - Brinch Hansen: signal can only be done when exiting the monitor
  - Or: signaler keeps running; waiter resumes when signaler exits monitor

Monitors (2)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Monitors (3)

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Monitors (4)

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );       // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );       // start the producer thread
           c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
           public void run( )  { // run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }      // actually produce
        }

        static class consumer extends Thread {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitors (5)

```
    private int produce_item( ) { ... }      // actually produce
}

static class consumer extends Thread {
    public void run( ) { run method contains the thread code
        int item;
        while (true) {      // consumer loop
            item = mon.remove( );
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... } // actually consume
}

static class our_monitor {  // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;  // counters and indices

    public synchronized void insert(int val) {
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitors (6)

```
        if (count == N) go_to_sleep( );    // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;        // slot to place next item in
        count = count + 1;        // one more item in the buffer now
        if (count == 1) notify( );        // if consumer was sleeping, wake it up
    }

    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );    // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;        // slot to fetch next item from
        count = count – 1;        // one few items in the buffer
        if (count == N – 1) notify( ); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

The Disadvantages of Monitors

- Monitors are nice, but they're only available in a very few languages
- Monitors, semaphores, and mutexes are fine on a single machine, even a multiprocessor
- They don't work well without some shared memory
- What about a distributed system connected via a LAN?

Message Passing

- Model based on network operations:
  - `send(dest, &mesg);`
  - `recv(src, &mesg);`
- Receiver blocks if there's no data available
- Sender blocks if the receiver's buffers are full
- Complex network questions, including acknowledgment, flow control, error detection and correction, message ordering, and authentication
  - all out of scope for this course

Using Message Passing

- Many different paradigms of how to use it
- Rendezvous — one message at a time; sender and receiver operate in lock-step
- Mailboxes — fixed-size buffers on channel; sender blocks if they're full
- Explicit requests — empty messages from the consumer to the producer

The Producer-Consumer Problem with Message Passing (1)

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     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)
```

Figure 2-36. The producer-consumer problem with *N* messages.

The Producer-Consumer Problem with Message Passing (2)

```
        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 */
    }
}
```

Figure 2-36. The producer-consumer problem with *N* messages.

Barriers

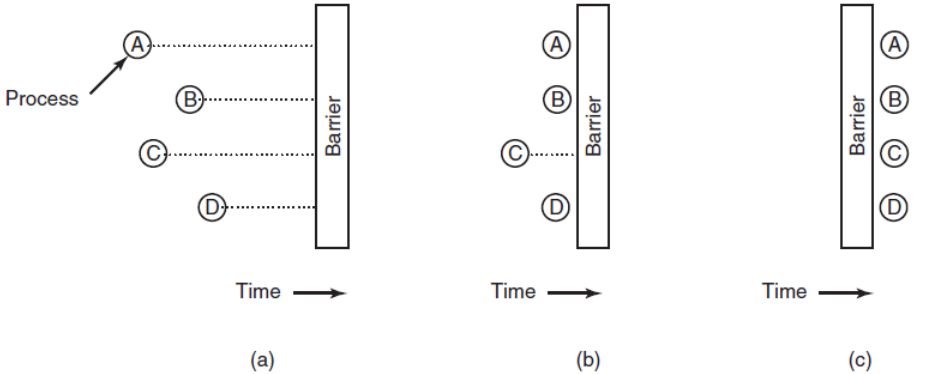


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Avoiding Locks: Read-Copy-Update (1)

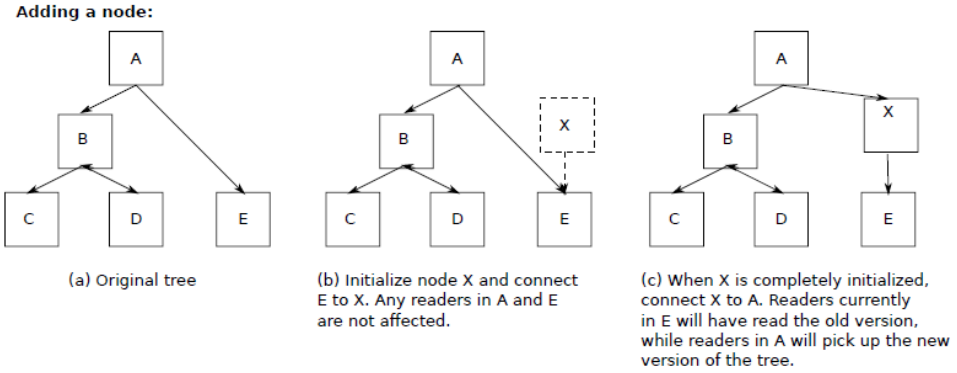


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

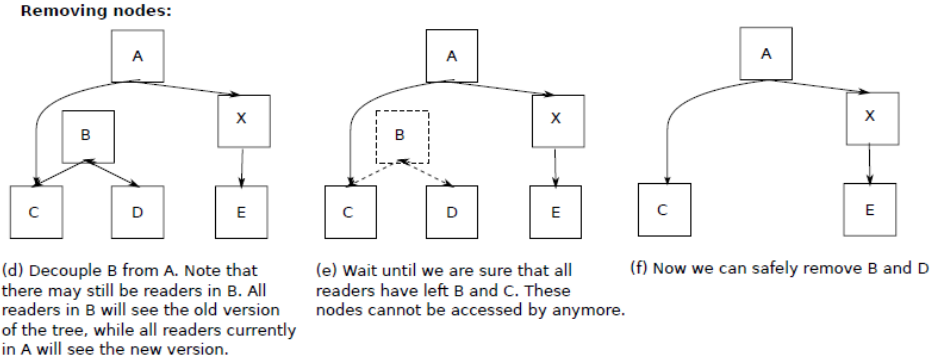


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Scheduling

- Suppose several processes are runnable?
- Which one is run next?
- Many different ways to make this decision

Environments

- Old batch systems didn't have a scheduler; they just read whatever was next on the input tape
- Actually, they did have a scheduler: the person who loaded the card decks onto the tape
- Hybrid batch/time-sharing systems tend to give priority to short timesharing requests
- Still a policy today: must give priority to interactive requests

Introduction to Scheduling
Process Behavior

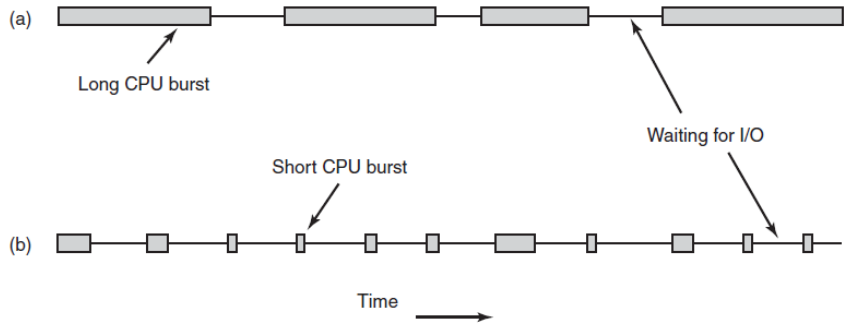


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Process Behavior

- Processes alternate CPU use with I/O requests
- I/O requests frequently block, either waiting for input or when too much has been written and no buffer space is available
- CPU-bound processes think more than they read or write
- CPUs have been getting much faster relative to disks
- It is better to schedule the I/O-bound processes earlier so that they issue their I/O earlier

When to Make Scheduling Decisions

- After a fork — run the parent or child?
- On process exit
- When a process blocks
- When I/O completes
- Sometimes, after timer interrupts

Preemptive vs. Nonpreemptive Schedulers

- Nonpreemptive scheduler: lets a process run as long as it wants, Only switches when it blocks
- Preemptive: switches after a time quantum.
- Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler.
- If no clock is available, nonpreemptive scheduling is the only option

Categories of Scheduling Algorithms

1. Batch. – responsiveness isn't important; preemption moderately important.
2. Interactive. – must satisfy a human; preemption important
3. Real time. – often nonpreemptive!!

Scheduling Algorithm Goals

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Figure 2-40. Some goals of the scheduling algorithm under different circumstances.

Goals- All Systems

- Fairness — give each process its share of the CPU
- Policy and enforcement — give preference to work that is administratively favored; prevent subversion of OS scheduling policy
- Balance — keep all parts of the system busy

Goals: Batch Systems

- Throughput — maximize jobs/hour
- Turnaround time — return jobs quickly. Often want to finish short jobs very quickly
- CPU utilization

Interactive Systems

- Response time — respond quickly to user requests
- Meet user expectations — psychological
  - Users have a sense of "cheap" and "expensive" requests
  - Users are happier if "cheap" requests finish quickly
  - "Cheap" and "expensive" don't always correspond to reality!

Real-Time Systems

- Meet deadlines — avoid losing data (or worse!)
- Predictability — users must know when their requests will finish
- Requires careful engineering to match priorities to actual completion times and available resources

Scheduling in Batch Systems

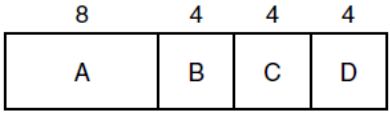
Batch Schedulers

- First-Come, First-Served
- Shortest Job First (Shortest first)
- Shortest Remaining Time Next
  (Shortest remaining time first)

First-Come, First-Served

- Run the first process on the run queue
- When the running process blocks, the first process on the queue is run next.
- Never preempt based on timer
- Seems simple; just like waiting in line
- Not very fair (no preemption means CPU bound processes takes more CPU priority)

Shortest Job First



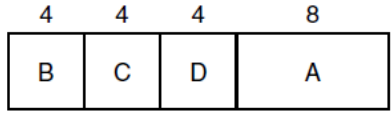
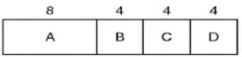
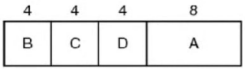


Figure 2-41. An example of shortest job first scheduling.
(a) Running four jobs in the original order.
(b) Running them in shortest job first order.

Shortest First

- Suppose you know the time requirements of each job
- A needs 8 seconds, B needs 4, C needs 4, D needs 4
- Run B, C, D, A
- Nonpreemptive
- Provably optimal turnaround time:
  - Suppose four jobs have runtimes of a, b, c, and d
  - First finishes at time a, second at a + b, etc
  - Mean turnaround is (4a + 3b + 2c + d)/4
  - d contributes less to the mean



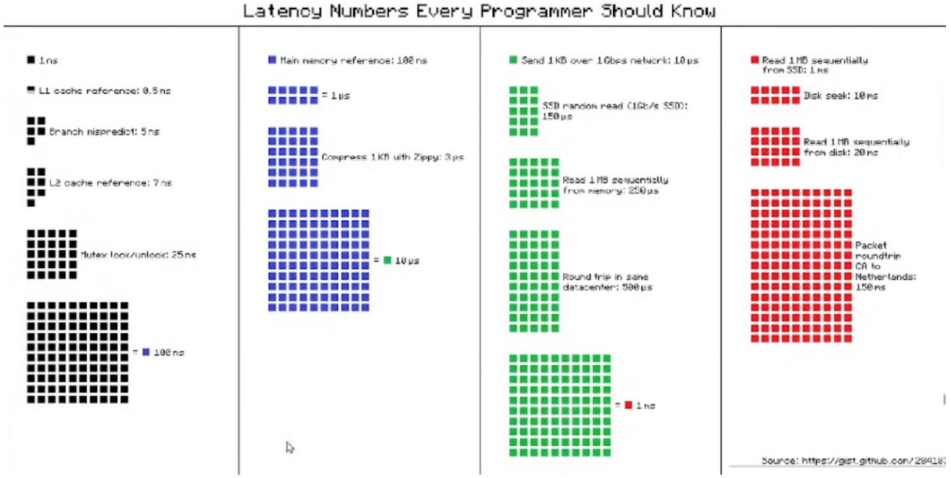


Optimality Requires Simultaneous Availability

- Jobs don't all arrive at the same time
- Can't make optimal scheduling decision without complete knowledge
- Example: jobs with times of 2,4,1,1,1 that arrive at times 0,0,3,3,3
- Shortest-first runs A, B, C, D, E; average wait is 4.6 secs
- If we run B, C, D, E, A, average wait is 4.4 secs
- While B is running, more jobs arrive, allowing a better decision for the total load
- We need somwthing else!

Shortest Remaining Time Next

- Preemptive variant of FCFS
- Still need to know run-times in advance
- When a new job arrives, its total time is compared to the current process' remaining time
- Helps short jobs get good service
- May have a problem with indefinite overtaking



Scheduling in Interactive Systems

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

Round-Robin Scheduling

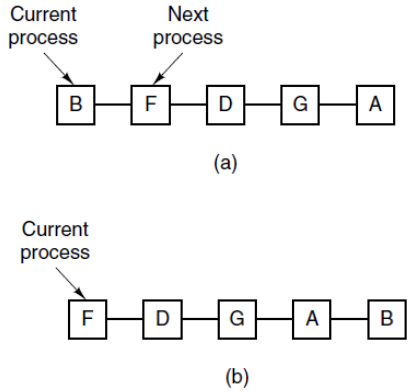


Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Round-Robin Scheduling

- One of the oldest, simplest, fairest, and most widely used algorithms is round robin.
- Each process is assigned a time interval, called its quantum, during which it is allowed to run.
- If the process is still running at the end of the quantum, the CPU is preempted and given to another process.
- If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course.

Quantum Length

- The shorter the quantum, the more responsive the system
- However, process switching is expensive (saving and reloading registers, switching virtual memory maps, flushing the cache, bookkeeping, etc.)
- Suppose the computer quantum is 4 msec and process switches take 1 msec.
- That's 20% overhead — too much
- Suppose we have 100 msec quanta
- If the run queue ever gets long, even cheap requests will take too long
- Need a reasonable compromise: 20-50 msec?

Priority Scheduling

- Not all processes are equally important
- Assign them priority levels
- Simplest version: always run the highest-priority process
- Not a good idea — what if it's CPU bound?

Priority Inversion Revisited

- Strict priority scheduling can lead to a phenomenon called "priority inversion"
- Consider the following example where p(H) > p(M) > p(L)
  - H needs a lock currently held by L, so H blocks
  - M that was already on the ready list gets the processor before L
  - H indirectly waits for M, (no deadlock but priority inversion)
- On Path Finder, a watchdog timer noticed that H failed to run for some time, and continuously reset the system

Priority Adjustments

- Periodically reduce the priority of the running process
- Eventually, it falls below the priority of the next process
- Alternative: increase the priority of non-running processes
- Called process aging
- Or do both

Dynamic Priority Adjustment

- Adjust process priority according to its recent history
  - Example: increase priority of non-running processes; decrease priority of running processes
- Boost priority of I/O-bound processes (why??)
- If process used 1/f of its last quantum, boost its priority proportional to f. Set the priority to 25, where 1/25 is the fraction of the last quantum that a process used.
- Use priority classes: have separate queues for each priority level, and run each queue round-robin; switch to lower-priority queue when this one is empty

Run Queues

- Each run queue is a linked list
- To raise or lower a process' priority, move it to a different list
- Two schemes for priority aging:
  - Not many processes: have a fine-grained counter for each process incremented at a clock interrupt; at some limit, increase priority
  - Lots of processes and queues: periodically, move each list to the next level up

Priority Scheduling

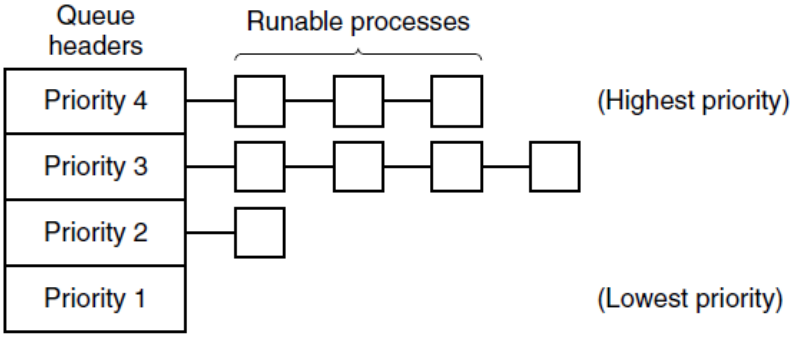


Figure 2-43. A scheduling algorithm with four priority classes.

Varying Quanta

- Many processes need just a little bit of CPU time
- Since process switches can be expensive, don't do them too often
- Solution to both problems: give lower priority queues longer quanta
- Top priority queue: one quantum
- Second queue: two quanta CPU allocation
- Third queue: four quanta, etc.
- Alternate solution: "short" (initial) quantum at high priority and "long" quanta at low priority thereafter

Helping Interactive Processes

- Look for signs of user input
- When they occur, give the process a very high priority
- Example: on CTSS, when a user typed a carriage return, the process got top priority
- Harder to do today — what's "interactive" on a networked process? For a mouse movement, which process is credited?

Process Priorities

- What processes should have higher priorities?
- Administrative issues
- System performance
  - Kernel processes (up to a point)
  - Interactive services processes (i.e., X server)
- Users can lower priority of their own processes, sometimes to avoid competing with themselves

Unix Priorities

- Tradition: lower numbers indicate higher priorities
- "Nice" value is a user-specified modifier
- A nice value of +20 specifies a very low priority process
- Only root can set negative niceness
- Default is 0
- Note: this is an API; internal metric can be different

Some Linux Priorities

UID	PRI	NI	SZ	CMD
root	76	0	606	init
root	94	19	0	[ksoftirqd/0]
root	75	0	0	[khubd]
root	83	0	0	[scsi_eh_1]
root	79	0	6285	ypbind
root	75	0	1242	/usr/sbin/sshd
smb	76	0	1007	ps

The PRI value factors in the niceness and the process' dynamic priority

Shortest Process Next

- Can we emulate batch systems' shortest first algorithm?
- It's hard, because we don't have good estimates
- Instead, use historical data for a moving, decaying, average
- Let first time = T_0; second is T_1

$$T_2 = \alpha T_0 + (1-\alpha)T_1$$

$$T_3 = \alpha^2 T_0 + \alpha(1-\alpha)T_1 + (1-\alpha)T_2$$

$$T_4 = \alpha^3 T_0 + \alpha^2(1-\alpha)T_1 + \alpha(1-\alpha)T_2 + (1-\alpha)T_3$$

...

Guaranteed Scheduling

- For n users or process on a system, give each 1/n of the CPU
- Measure actual CPU usage
- Calculate process' CPU time entitlement
- Look at the ratio of the two: 0.5 means it's had only half the CPU it's entitled to, so it gets priority over a process with a ratio of 2.0

Lottery Scheduling

- Give each process lottery tickets
- Higher priority processes get more tickets; lower priority process get fewer
- At scheduling time, pick a random ticket
- The process holding that ticket gets to run
- Note: tickets can be exchanged between processes, such as between client and server

Fair-Share Scheduling

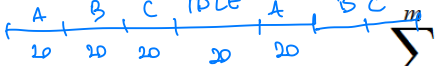
- In some systems, processes don't compete, users do
- We don't want to encourage forking just to get a larger share of the CPU
- Solution: make decisions based on user CPU consumption instead of process CPU consumption
- Priorities, etc., can still apply

Scheduling in Real-Time Systems

- Time plays an essential role
- Categories
  - Hard real time
  - Soft real time
  - Periodic or aperiodic
- Schedulable satisfies

Real-Time Systems

- Meet deadlines — avoid losing data (or worse!)
- Predictability — users must know when their requests will finish
- Requires careful engineering to match priorities to actual completion times and available resources



$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A CPU'yu 20'den fazla kullanırsa preemption

Real-Time Systems

- **hard real time**, meaning there are absolute deadlines that must be met—or else!
- **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable

approximate

Video player
50 ms – 1 frame
20 frame

Periodic Events

- Many real-time events occur with a regular frequency
- Suppose there are m events, withe event i needing Ci seconds of CPU and occurring every Pi seconds
- System works if and only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Watch out for process switch overhead. . .

Schedulable real time systems

A soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively.

periyodik calisiyorlar
öyle tasarlanmişlar, özeller

If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because 0.5 + 0. 15 + 0. 2 < 1.

	A	B	C			
Period	100	200	500	$\frac{50}{100}$ +	$\frac{30}{200}$ +	$\frac{100}{500}$
CPU Usage	50	30	100	.50	.15	.20

= .85

Aperiodic Events

- Other events don't happen on a regular schedule
- The basic constraint is the same: total load must be less than total capacity
- Engineering such systems is harder

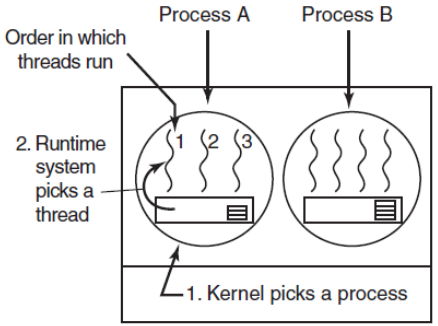
Different address spaces is more expensive.

A1 → A2

A1 → B1 → more expensive

Thread Scheduling (1)

cache



bunu tercih eder

ama ihtiyac varse bunu da yapar.

Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (2)

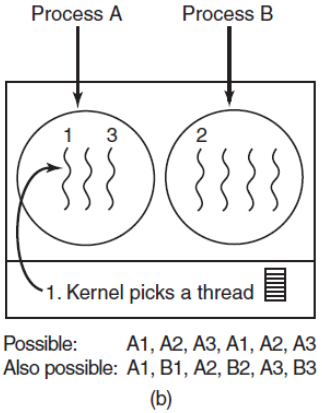


Figure 2-44. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Thread Scheduling (3)

- Since the kernel knows that switching from a thread in process A to a thread in process B is more expensive than running a second thread in process A, it can take this information into account when making a decision.
- User-level threads can employ an application-specific thread scheduler: for example, the Web server.
- Suppose that a worker thread has just blocked and the dispatcher thread and two worker threads are ready. Who should run next?

The Dining Philosophers Problem (1)

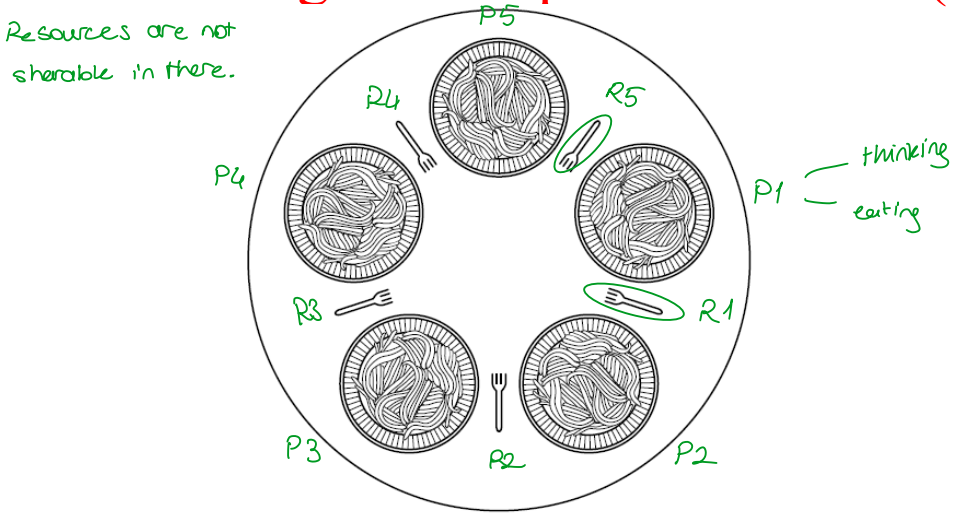


Figure 2-45. Lunch time in the Philosophy Department.

The Dining Philosophers Problem (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                                /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat();                                  /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

down(mutex)

up(mutex)

Figure 2-46. A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem (3)

```
#define N           5                       /* number of philosophers */
#define LEFT        (i+N–1)%N               /* number of i's left neighbor */
#define RIGHT       (i+1)%N                 /* number of i's right neighbor */
#define THINKING    0                       /* philosopher is thinking */
#define HUNGRY      1                       /* philosopher is trying to get forks */
#define EATING      2                       /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

Figure 2-47. A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
        put_forks(i);                       /* put both forks back on table */
    }
}

void take_forks(int i)                      /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
    test(i);                                /* try to acquire 2 forks */
    up(&mutex);                             /* exit critical region */
    down(&s[i]);                            /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
```

Figure 2-47. A solution to the dining philosophers problem.

The Dining Philosophers Problem (5)

```
}

void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-47. A solution to the dining philosophers problem.

The Readers and Writers Problem (1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
```

Figure 2-48. A solution to the readers and writers problem.

The Readers and Writers Problem (2)

```
        use_data_read();            /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data();            /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base();          /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

Figure 2-48. A solution to the readers and writers problem.

End

Chapter 2