I – M.Sc (IT)- Semester - I

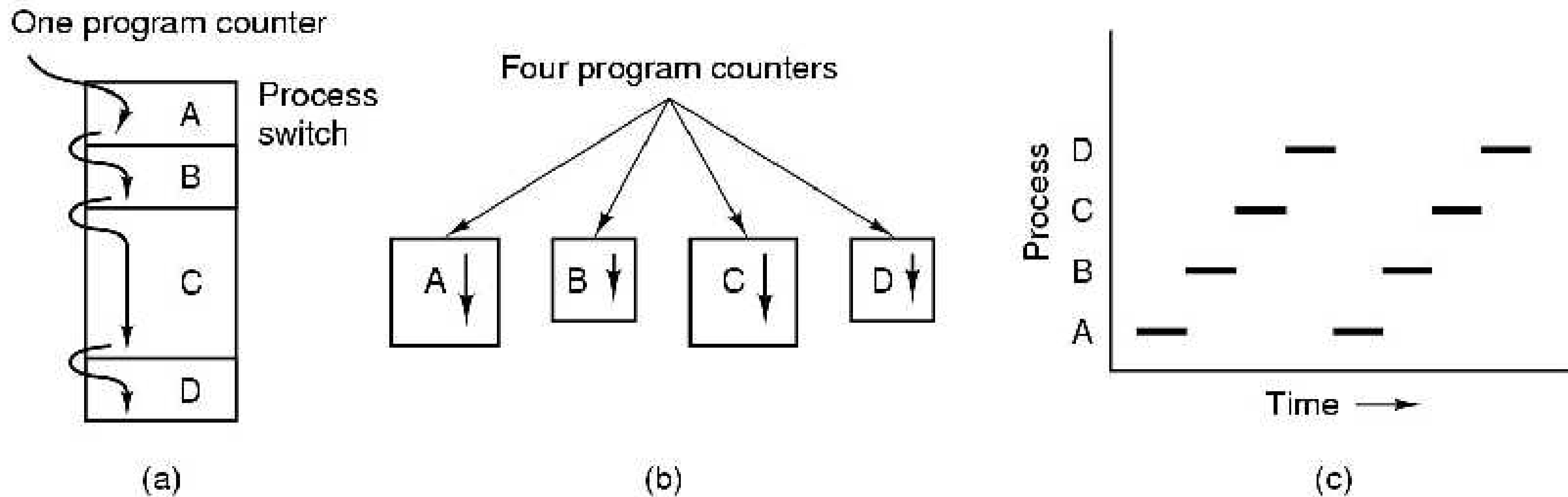
ADVANCED OPERATING SYSTEM
(18MIT12C)

UNIT - II

UNIT–II:

Threads – Thread Usage – Classical Thread model- POSIX threads - Pop-up threads – Inter Process Communication – Race condition – Critical Region – Mutual Exclusion with busy waiting – Sleep and wakeup – Semaphores – Mutexes – Monitors - Message Passing - Classical IPC Problems:
The Dining Philosophers Problem – The Readers and Writers Problem- Memory management: virtual memory – Paging- Paging tables- Speeding up paging – Page tables for large memories.

Multiprogramming



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

Reasons to use threads

- Enables parallelism (web server) with blocking system calls
- Threads are faster to create and destroy then processes
- Natural for multiple cores
- Easy programming model

Threads are lightweight

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	

Threads are like processes

- Have same states
  - Running
  - Ready
  - Blocked
- Have their own stacks –same as processes
- Stacks contain frames for (un-returned) procedure calls
  - Local variables
  - Return address to use when procedure comes back

How do threads work?

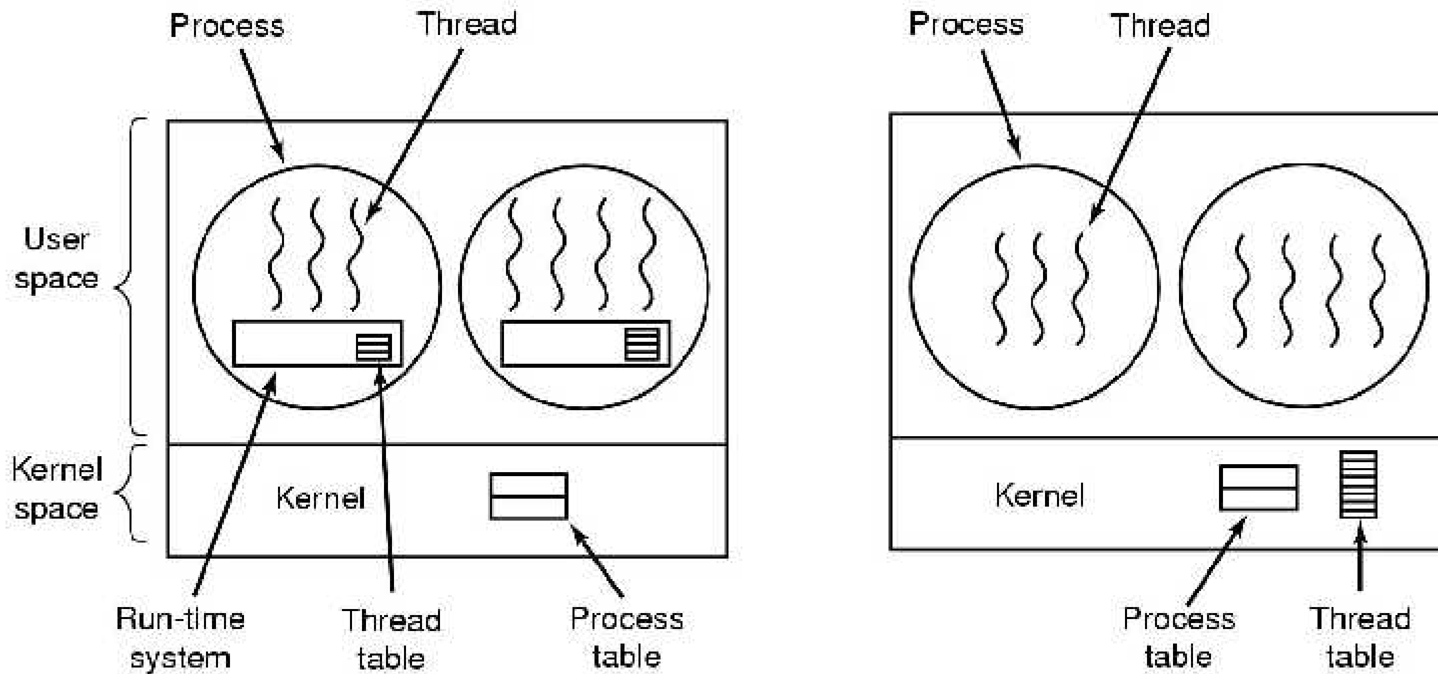
- Start with one thread in a process
- Thread contains (id, registers, attributes)
- Use library call to create new threads and to use threads
  - Thread_create includes parameter indicating what procedure to run
  - Thread_exit causes thread to exit and disappear (can’t schedule it)
  - Thread_join Thread blocks until another thread finishes its work
  - Thread_yield

POSIX Threads (Pthreads)

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

Pthreads are IEEE Unix standard library calls

Implementing Threads in User Space



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

Threads in user space-the good

- Thread table contains info about threads (program counter, stack pointer...) so that run time system can manage them
- If thread blocks, run time system stores thread info in table and finds new thread to run.
- State save and scheduling are invoked faster then kernel call (no trap, no cache flush)

Threads in user space-the bad

- Can’t let thread execute system call which blocks because it will block all of the other threads
- No elegant solution
  - Hack system library to avoid blocking calls
  - Could use select system calls-in some versions of Unix which do same thing
- Threads don’t voluntarily give up CPU
  - Could interrupt periodically to give control to run time system
  - Overhead of this solution is a problem…..

Threads in kernel space-the good

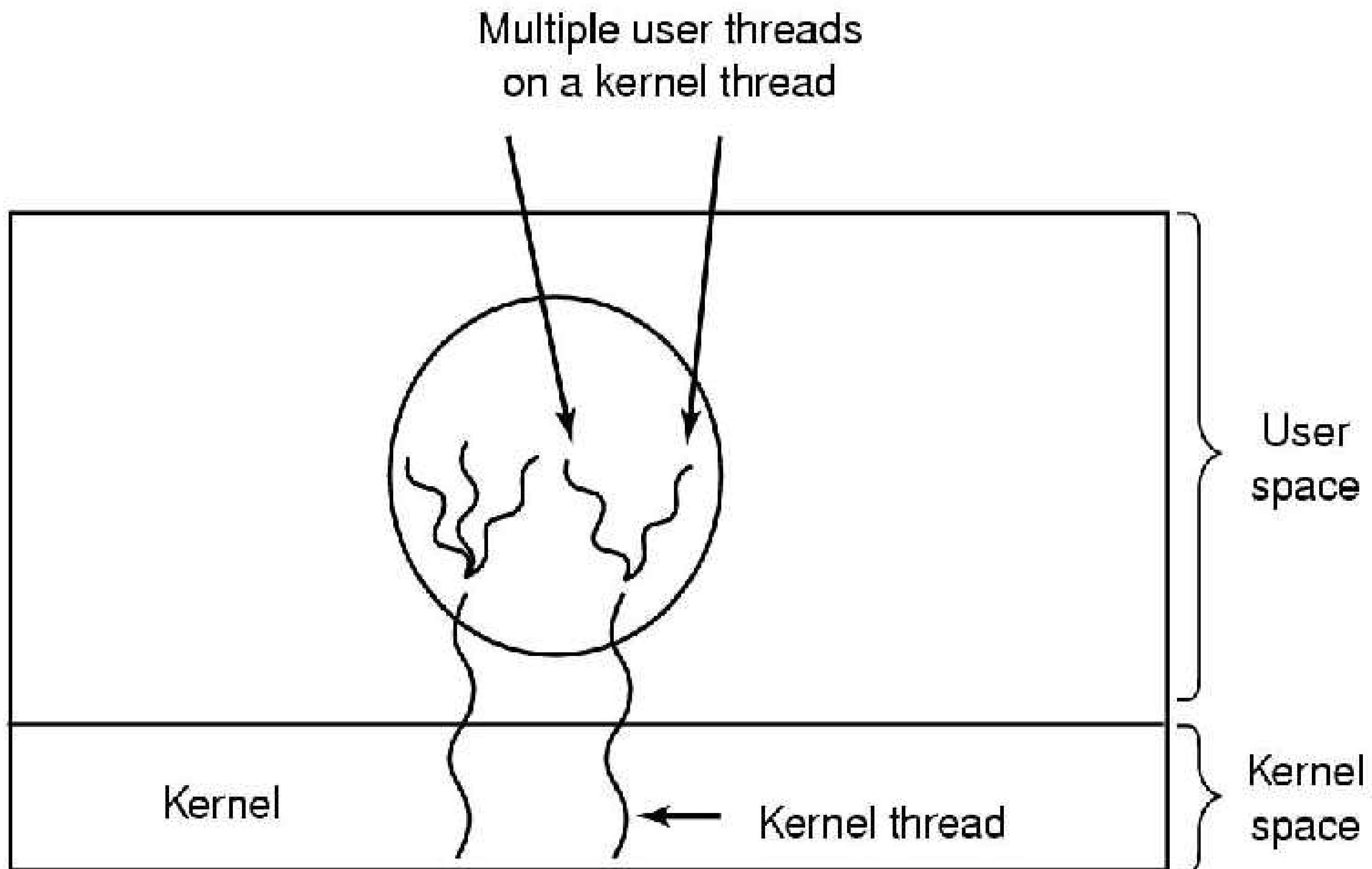
- Kernel keeps same thread table as user table
- If thread blocks, kernel just picks another one
  - Not necessarily from same process!
- The bad-expensive to manage the threads in the kernel and takes valuable kernel space

Threads in kernel space-the bad

- Expensive to manage the threads in the kernel and takes valuable kernel space
- How do we get the advantages of both approaches, without the disadvantages?

Hybrid approach

Multiplex user-level threads onto kernel level threads

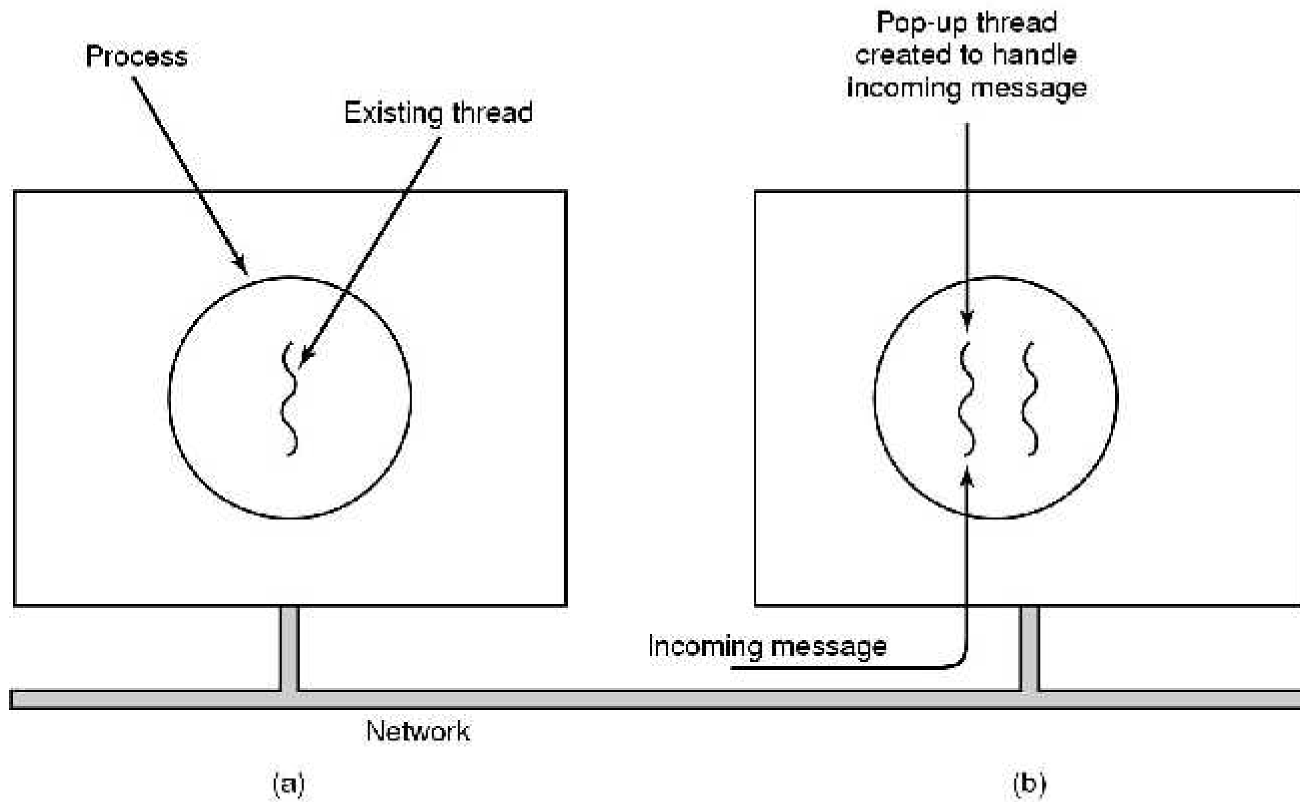


Hybrid

- Kernel is aware of kernel threads only
- User level threads are scheduled, created destroyed independently of kernel thread
- Programmer determines how many user level and how many kernel level threads to use

Pop-Up Threads

(How to handle message arrivals in distributed systems)



Create a new thread when a message arrives

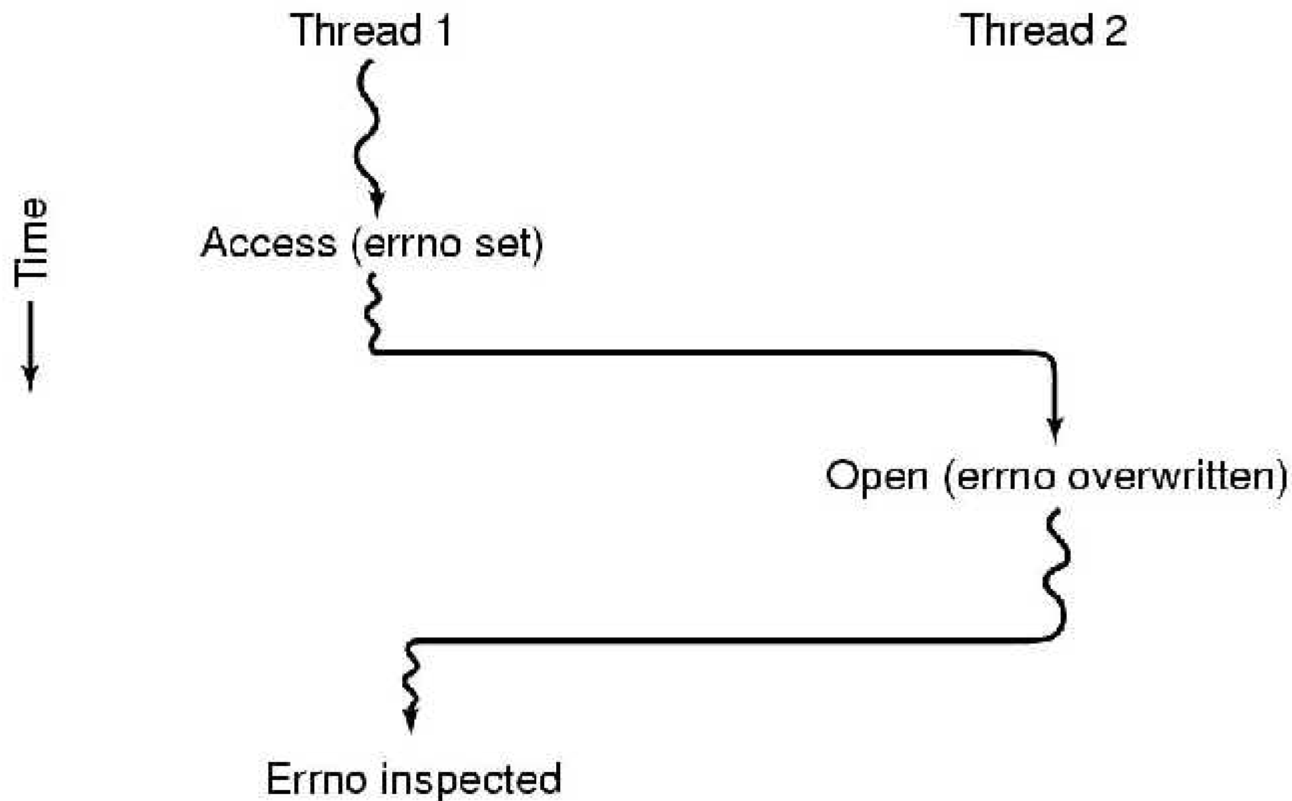
Why pop ups?

- Could use thread which blocks on a receive system call and processes messages when they arrive
- Means that you have to restore the history of the thread each time a message arrives
- Pop ups are entirely new-nothing to restore
- They are faster

Adding threads to an OS-problems

- How do we implement variables which should be global to a thread but not to the entire program?
- Example: Thread wants access to a file, Unix grants access via global errno
- Race ensues-thread 1 can get the wrong permission because thread 2 over-writes it

Thread 1 gets the wrong permission

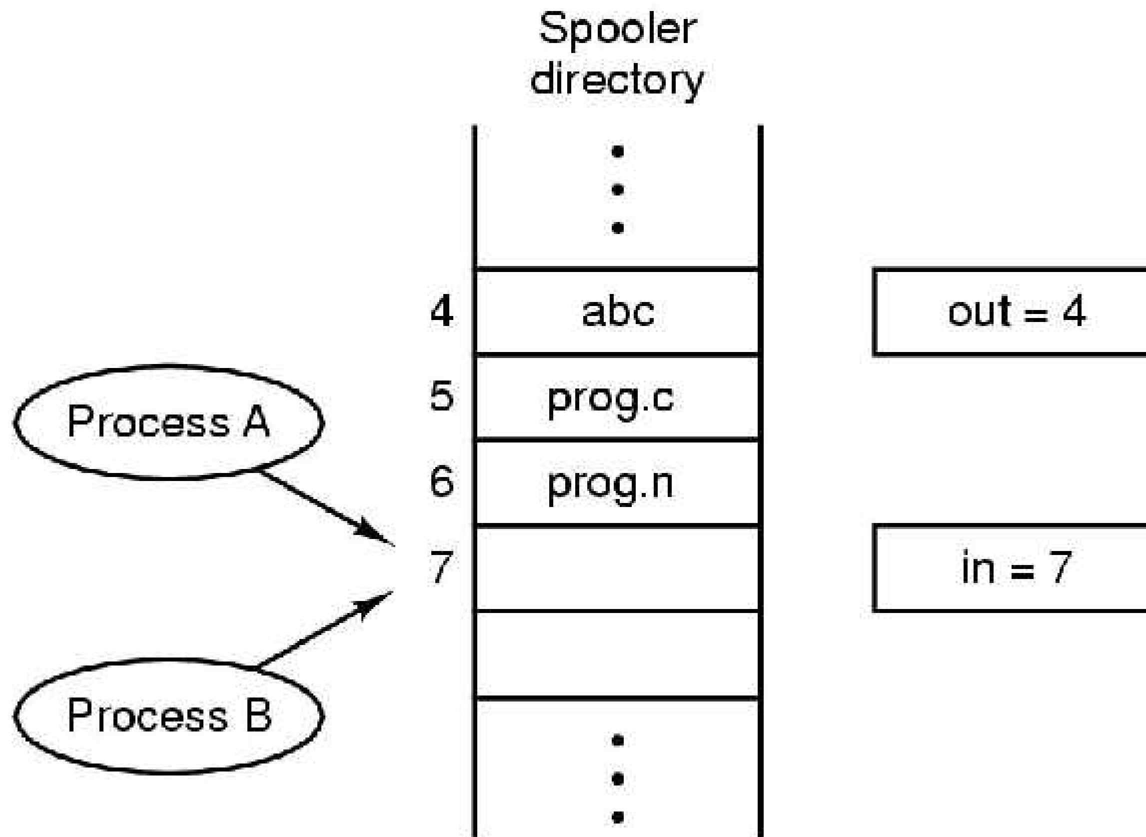


Interprocess Communication

- Three problems
  - How to actually do it
  - How to deal with process conflicts (2 airline reservations for same seat)
  - How to do correct sequencing when dependencies are present-aim the gun before firing it
- SAME ISSUES FOR THREADS AS FOR PROCESSES-SAME SOLUTIONS AS WELL
- Proceed to discuss these problems

.

Race Conditions



In is local variable containing pointer to next free slot

Out is local variable pointing to next file to be printed

How to avoid races

- Mutual exclusion–only one process at a time can use a shared variable/file
- Critical regions-shared memory which leads to races
- Solution- Ensure that two processes can't be in the critical region at the same time

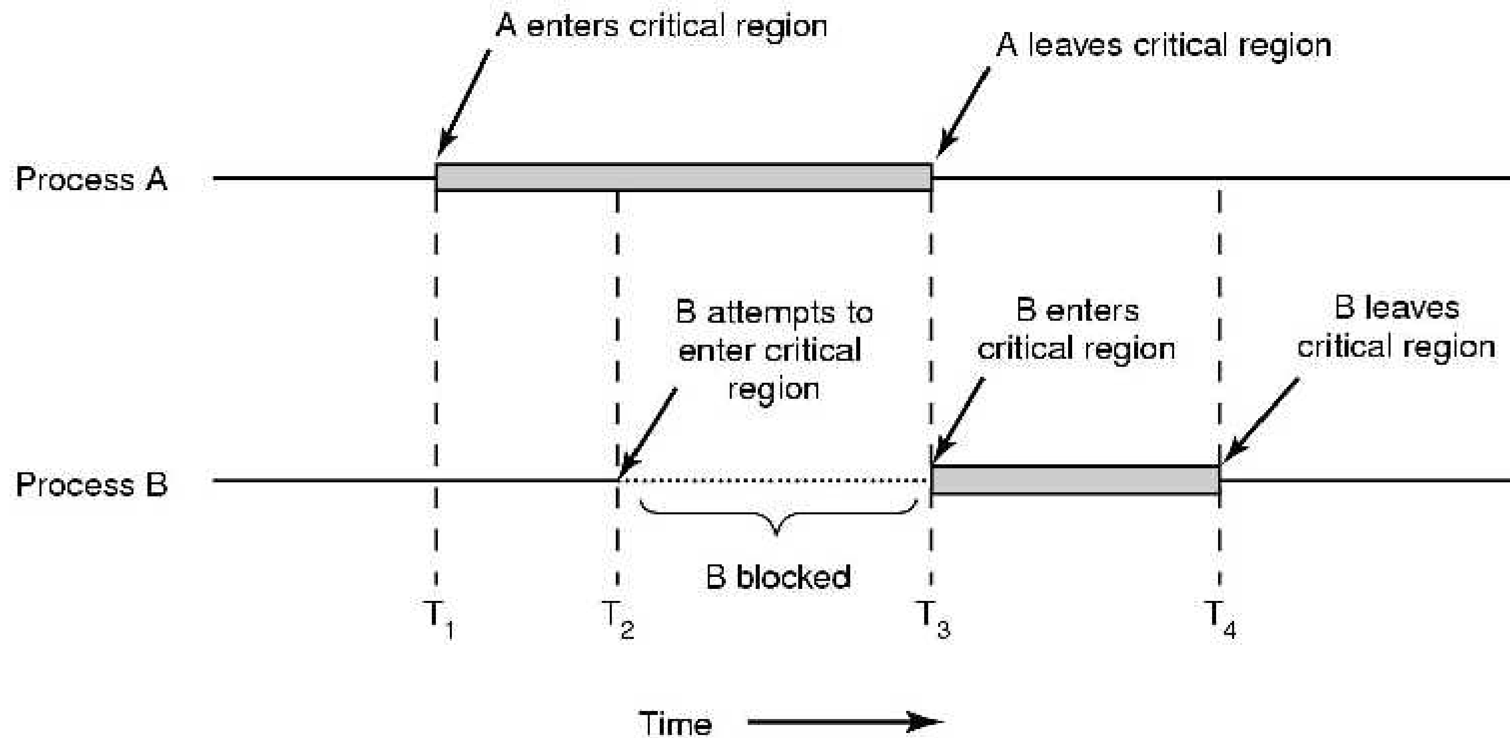
.

Properties of a good solution

- Mutual exclusion
- No assumptions about speeds or number of CPU's
- No process outside critical region can block other processes
- No starvation-no process waits forever to enter critical region

.

What we are trying to do



First attempts-Busy Waiting

A list of proposals to achieve mutual exclusion

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling Interrupts

most obvious way of achieving mutual exclusion is to allow a process to **disable interrupts** before it enters its critical section and then enable **interrupts** after it leaves its critical section. By **disabling interrupts** the CPU will be unable to switch processes.

- Idea: process disables interrupts, enters critical region , enables interrupts when it leaves critical region
- Problems
  - Process might never enable interrupts, crashing system
  - Won't work on multi-core chips as disabling interrupts only effects one CPU at a time

Lock variables

- A software solution-everyone shares a lock
  - When lock is 0, process turns it to 1 and enters critical region
  - When exit critical region, turn lock to 0
- Problem-Race condition

Strict Alternation

Turn Variable or **Strict Alternation** Approach.

Turn Variable or **Strict Alternation** Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.

```
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)

First me, then you

Problems with strict alternation

- Employs busy waiting-while waiting for the critical region, a process spins
- If one process is outside the critical region and it is its turn, then other process has to wait until outside guy finishes both outside AND inside (critical region) work

Peterson’s Solution

Peterson's algorithm (or **Peterson's solution**) is a concurrent programming **algorithm** for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication

The **critical section** is a code segment where the shared variables can be accessed. An atomic action is required in a **critical section** i.e. only one process can execute in **its critical section** at a time

Peterson’s Solution

```
#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 */
}
```

Peterson

- Process 0 & 1 try to get in simultaneously
- Last one in sets turn: say it is process 1
- Process 0 enters (turn= = process is False)

TSL

operating system provides a special instruction called Test Set Lock (**TSL**) instruction which simply loads the value of lock variable into the local register R0 and sets it to 1 simultaneously.

- TSL reads lock into register and stores NON ZERO VALUE in lock (e.g. process number)
- Instruction is atomic: done by freezing access to bus line (bus disable)

Using TSL

test-and-set instruction is an **instruction** used to write 1 (**set**) to a memory location and return its old value as a single atomic (i.e., non-interruptible) **operation**. ...

```
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
```

TSL is atomic. Memory bus is locked until it is finished executing.

What’s wrong with Peterson, TSL ?

- Busy waiting-waste of CPU time!
- Idea: Replace busy waiting by blocking calls
  - Sleep blocks process
  - Wakeup unblocks process

The Producer-Consumer Problem (Bounded Buffer Problem)

```
#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)
{
     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 */
     }
}
```

The problem with sleep and wake-up calls

- Empty buffer,count==0
- Consumer gets replaced by producer before it goes to sleep
- Produces something, count++, sends wakeup to consumer
- Consumer not asleep, ignores wakeup, thinks count= = 0, goes to sleep
- Producer fills buffer, goes to sleep
- P and C sleep forever
- So the problem is lost wake-up calls

Semaphores

a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent **system** such as a multitasking **operating system**

3-**types of semaphores** namely Binary, Counting and Mutex **semaphore**.

Wait and signal are the atomic **operation** possible on **semaphore**

Semaphores

- Semaphore is an integer variable
- Used to sleeping processes/wakeups

- Two operations, down and up

- Down checks semaphore. If not zero, decrements semaphore. If zero, process goes to sleep
- Up increments semaphore. If more then one process asleep, one is chosen randomly and enters critical region (first does a down)

- ATOMIC IMPLEMENTATION-interrupts disabled

Producer Consumer with Semaphores

- 3 semaphores: full, empty and mutex
- Full counts full slots (initially 0)
- Empty counts empty slots (initially N)
- Mutex protects variable which contains the items produced and consumed

Producer Consumer with semaphores

The **producer consumer** problem is a synchronization problem. There is a fixed size buffer and the **producer** produces items and enters them into the buffer. The **consumer** removes the items from the buffer and consumes them.

A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A **semaphore** is simply a variable

Example of a multi-process synchronization **problem**. The **problem** describes two processes, the **producer** and the **consumer** , who share a common, fixed-size buffer used as a queue

Producer Consumer with semaphores

```
#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)
{
        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 */
        }
}
```

Mutexes

Mutex is a **mutual exclusion** object that synchronizes access to a resource. It is created with a unique name at the start of a program. The **Mutex** is a locking mechanism that makes sure only one thread can acquire the **Mutex** at a time and enter the critical section

- Mutex: variable which can be in one of two states-locked (0), unlocked(1 or other value)
  - Easy to implement
- Good for using with thread packages in user space
  - Thread (process) wants access to cr, calls mutex_lock.
  - If mutex is unlocked, call succeeds. Otherwise, thread blocks until thread in the cr does a mutex_unlock.

Mutexes

Mutex is only modified by the process that may request or release a resource.

Mutex operations are locked or unlocked.

```
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
```

Pthread calls for mutexes

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

Pthread_mutex-trylock tries to lock mutex. If it fails it returns an error code, and can do something else.

Condition Variables

- Allows a thread to block if a condition is not met, e.g. Producer-Consumer. Producer needs to block if the buffer is full.
- Mutex make it possible to check if buffer is full
- Condition variable makes it possible to put producer to sleep if buffer is full
- Both are present in pthreads and are used together

Pthread Condition Variable calls

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

Producer Consumer with condition variables and mutexes

- Producer produces one item and blocks waiting for consumer to use the item
- Signals consumer that the item has been produced
- Consumer blocks waiting for producer to signal that item is in buffer
- Consumer consumes item, signals producer to produce new item

Producer Consumer with condition variables

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
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);
}

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)
{
        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);
}
```

Monitors

- Easy to make a mess of things using mutexes and condition variables. Little errors cause disasters.
  - Producer consumer with semaphores-interchange two downs in producer code causes deadlock
- Monitor is a language construct which enforces mutual exclusion and blocking mechanism
- C does not have monitor

Monitors

- Monitor consists of {procedures, data structures, and variables} grouped together in a “module”
- A process can call procedures inside the monitor, but cannot directly access the stuff inside the monitor
- C does not have monitors

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Monitor-a picture

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Onwards

- In a monitor it is the job of the compiler, not the programmer to enforce mutual exclusion.
- Only one process at a time can be in the monitor

  When a process calls a monitor, the first thing done is to check if another process is in the monitor. If so, calling process is suspended.
- Need to enforce blocking as well –
  - use condition variables
  - Use wait , signal ops on cv's

Condition Variables

- Monitor discovers that it can’t continue (e.g. buffer is full), issues a signal on a condition variable (e.g. full) causing process (e.g. producer) to block
- Another process is allowed to enter the monitor (e.g. consumer).This process can can issue a signal, causing blocked process (producer) to wake up
- Process issuing signal leaves monitor

Producer Consumer Monitor

```
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;

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;
```

Monitors:Good vs Bad

- The good-No messy direct programmer control of semaphores
- The bad- You need a language which supports monitors (Java).
- OS's are written in C

Semaphores:Good vs Bad

- The good-Easy to implement
- The bad- Easy to mess up

Reality

- Monitors and semaphores only work for shared memory
- Don’t work for multiple CPU’s which have their own private memory, e.g. workstations on an Ethernet

Message Passing

- Information exchange between machines
- Two primitives
  - Send(destination, &message)
  - Receive(source,&message)
- Lots of design issues
  - Message loss
    - acknowledgements, time outs deal with loss
  - Authentication-how does a process know the identity of the sender? For sure, that is

Producer Consumer Using Message Passing

- Consumer sends N empty messages to producer
- Producer fills message with data and sends to consumer

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 */
    }
}
```

.

Producer-Consumer Problem with Message Passing (2)

. . .

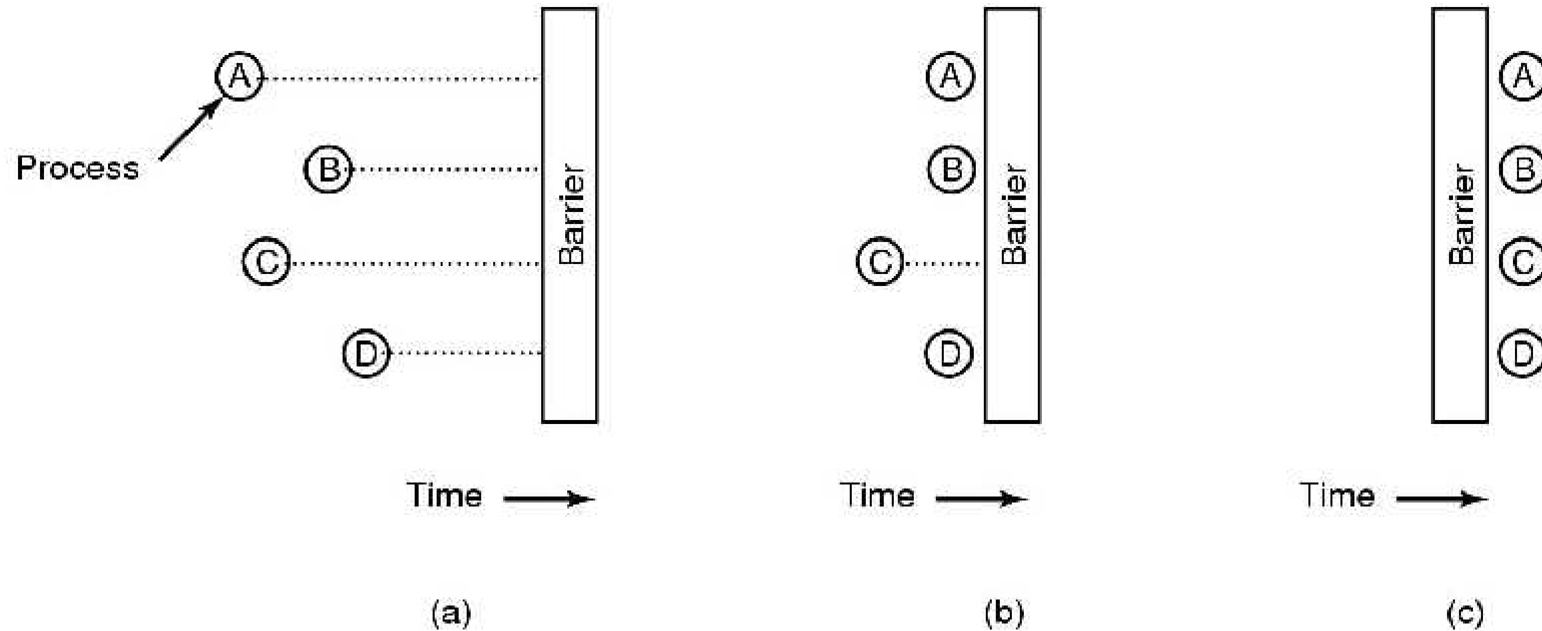
```
void consumer(void)
{
        int item, i;
        message m;

        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 */
        }
}
```

Message Passing Approaches

- Have unique ID for address of recipient process
- Mailbox
  - In producer consumer, have one for the producer and one for the consumer
- No buffering-sending process blocks until the receive happens. Receiver blocks until send occurs (Rendezvous)
- MPI

Barriers



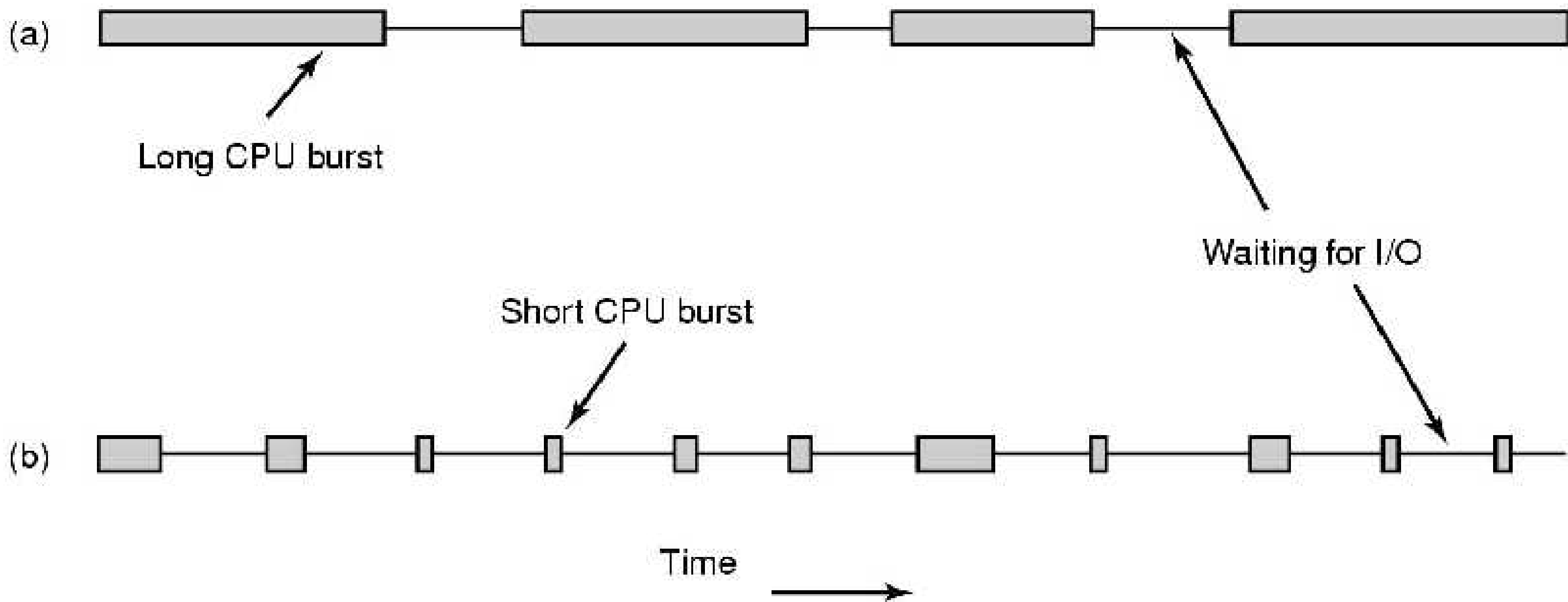
. Barriers are intended for synchronizing groups of processes

Often used in scientific computations.

Who doesn’t care about scheduling algorithms?

- PC’s
  - One user who only competes with himself for the CPU

Scheduling – Process Behavior



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

Categories of Scheduling Algorithms

- Batch (accounts receivable, payroll…..)
- Interactive
- Real time (deadlines)
- Depends on the use to which the CPU is being put

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

First come first serve

- Easy to implement
- Won’t work for a varied workload
  - I/O process (long execution time) runs in front of interactive process (short execution time)

Shortest Job First

- Need to know run times in advance
- Non pre-emptive algorithm
- Provably optimal

Eg 4 jobs with runs times of a,b,c,d

First finishes at a, second at a+b,third at a+b+c, last at a+b+c+d

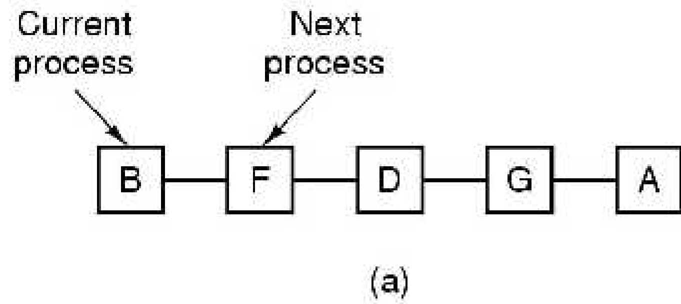
Mean turnaround time is (4a+3b+2c+d)/4

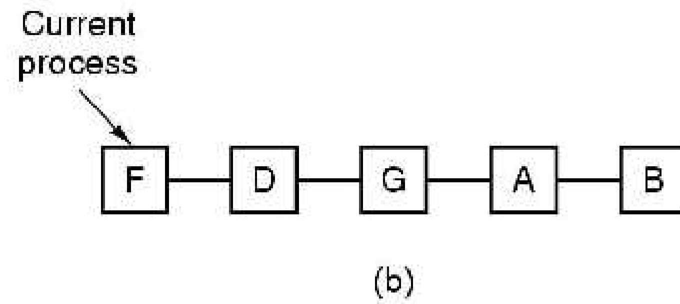
=> smallest time has to come first to minimize the mean turnaround time

Scheduling in Interactive Systems

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

Round-Robin Scheduling



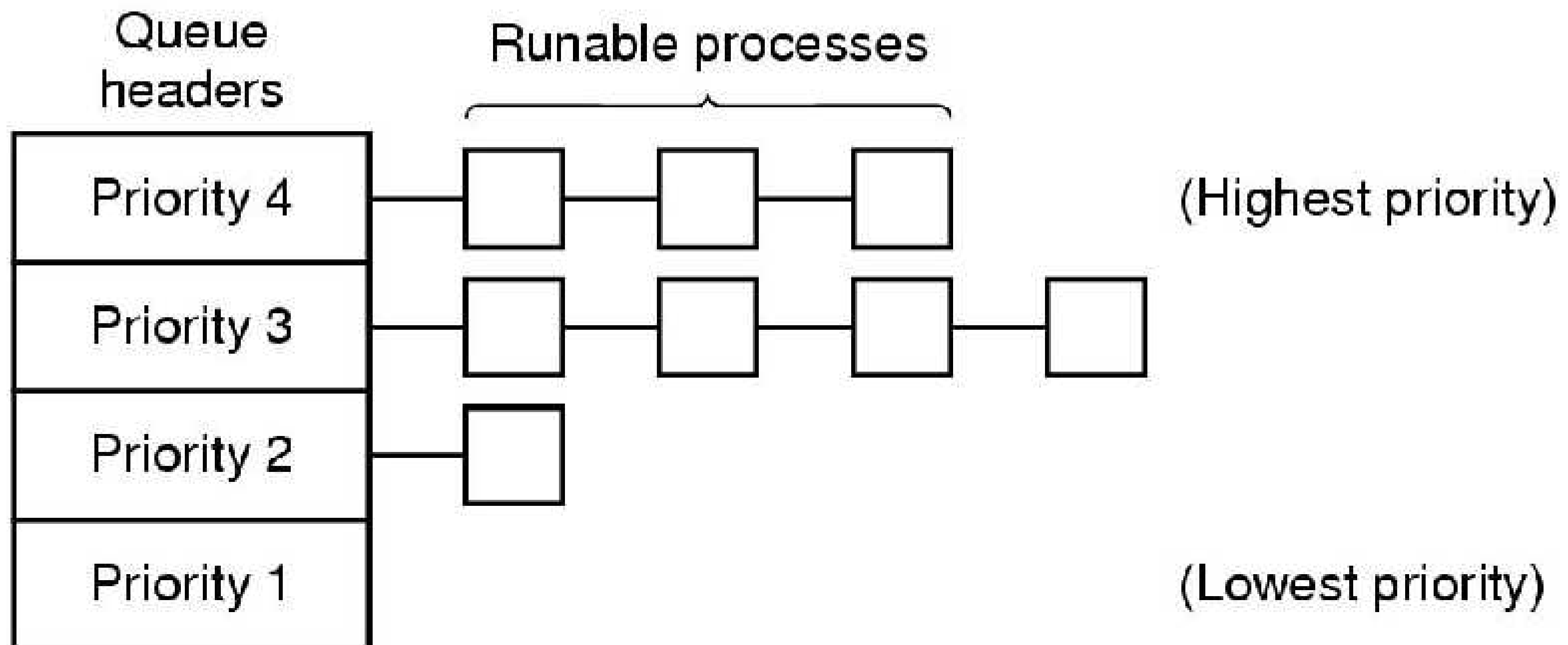


Process list-before and after

Priority Scheduling

- Run jobs according to their priority
- Can be static or can do it dynamically
- Typically combine RR with priority. Each priority class uses RR inside

Priority Scheduling



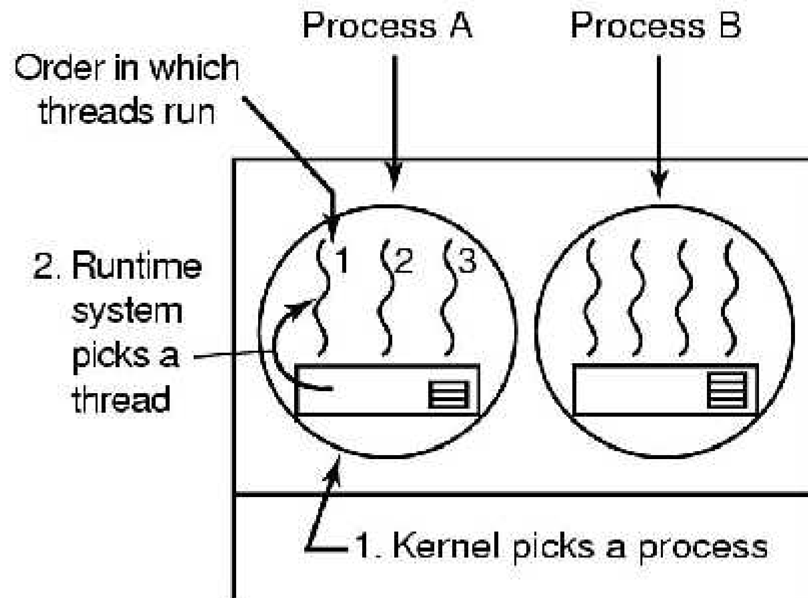
Shortest Process Next

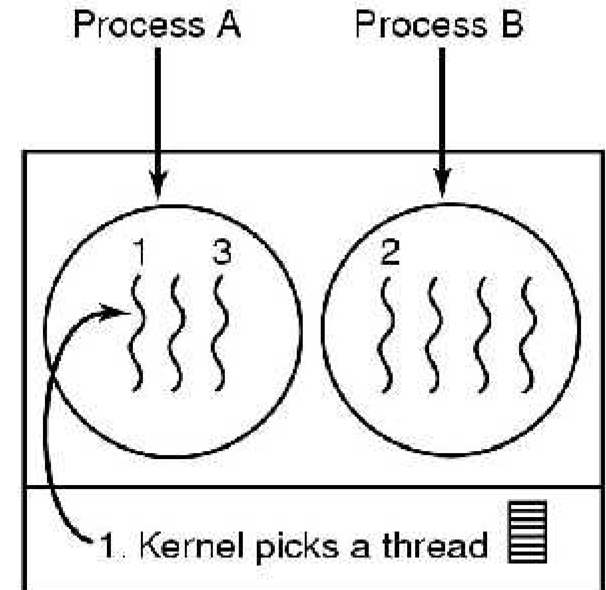
- Cool idea if you know the remaining times
- exponential smoothing can be used to estimate a jobs' run time
- $aT0 + (1-a)T1$ where T0 and T1 are successive runs of the same job

Lottery Scheduling

- Hold lottery for cpu time several times a second
- Can enforce priorities by allowing more tickets for “more important” processes

Thread Scheduling

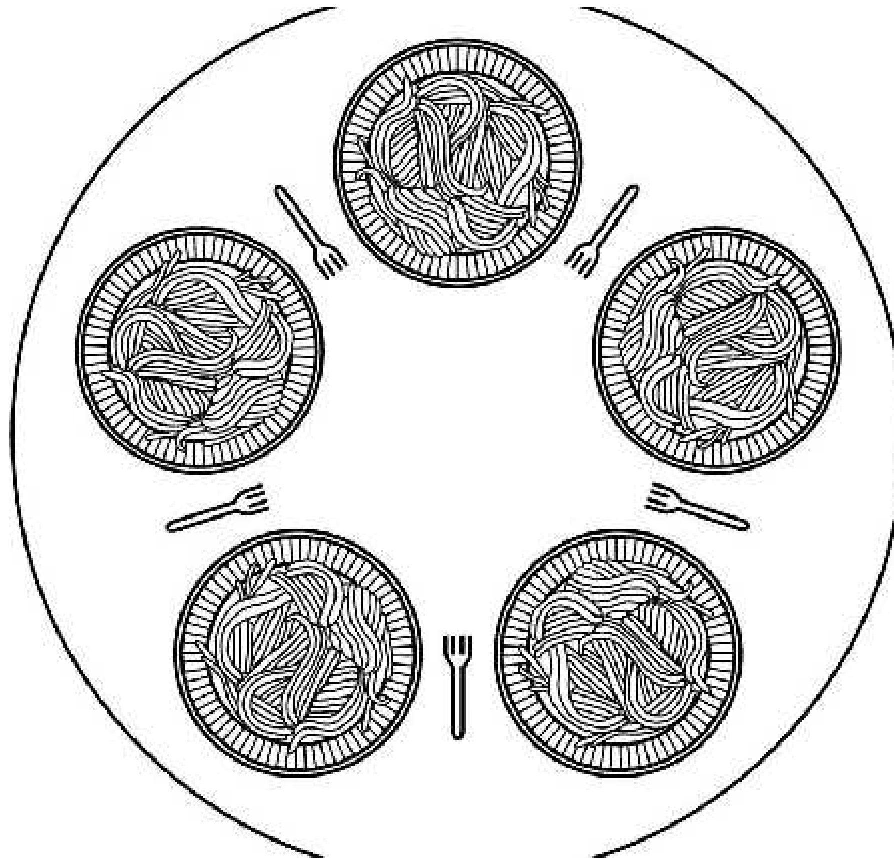




Kernel picks process (left)

Kernel picks thread (right)

Dining Philosophers Problem



. Lunch time in the Philosophy Department.

Dining Philosophers Problem

```
#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 for;% is modulo operator*/
        eat():                    /*self-explanatory*/
        put_fork(i);              /*put left fork back on table*/
        put_fork(i+1)%N;          /*put right fork back on table*/
        }
}
```

```
#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 */
     }
}
```

Dining Philosophers Problem

```
#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 */
    }
}
```

A solution to the dining philosophers problem. N=5.

```
#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 */
     }
}
```

Dining Philosophers Problem

. . .

```
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 */
}
```

. . .

. A solution to the dining philosophers problem.

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]);
        }
}
```

A solution to the dining philosophers problem.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- **Problem** – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore **mutex** initialized to 1.
  - Semaphore **wrt** initialized to 1.
  - Integer **readcount** initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
    wait (wrt) ;

//    writing is performed

    signal (wrt) ;
} while (true)
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)  wait (wrt) ;
    signal (mutex)

    // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
} while (true)
```

Readers-Writers Locks

Generalized to provide reader-writer locks on some systems.

Most useful in following situations:

1. In apps where it is easy to identify which processes only read shared data and which only write shared data.
2. In apps with more readers than writers. More overhead to create reader-writer lock than plain semaphores.

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 */
     }
}
```

. . .

A solution to the readers and writers problem.

The Readers and Writers Problem (2)

. . .

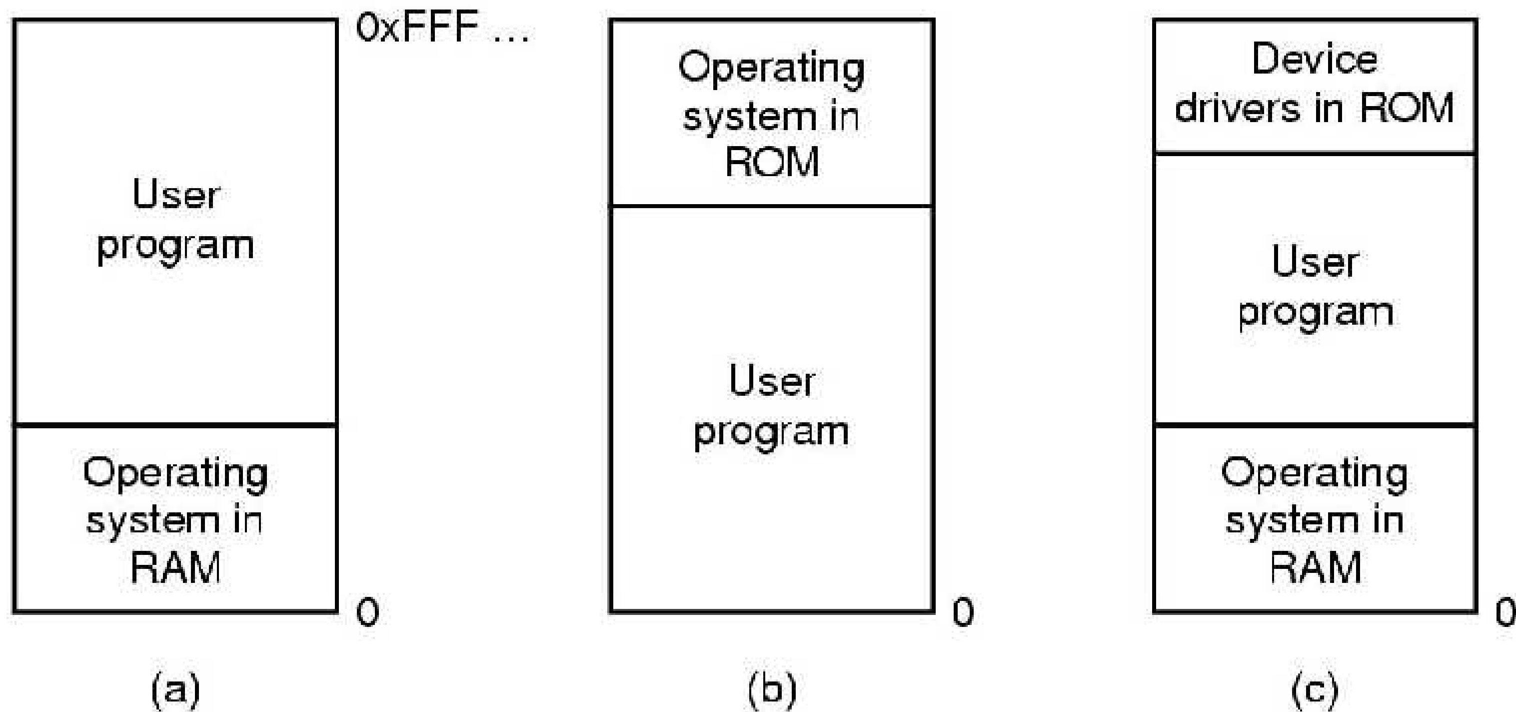
```
void writer(void)
{
        while (TRUE) {                      /* repeat forever */
                think_up_data( );           /* noncritical region */
                down(&db);                  /* get exclusive access */
                write_data_base( );         /* update the data */
     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

. A solution to the readers and writers problem.

Memory Management Basics

- Don’t have infinite RAM
- Do have a memory hierarchy-
  - Cache (fast)
  - Main(medium)
  - Disk(slow)
- Memory manager has the job of using this hierarchy to create an abstraction (illusion) of easily accessible memory

One program at a time in memory



OS reads program in from disk and it is executed

One program at a time

- Can only have one program in memory at a time.
- Bug in user program can trash the OS (a and c)
- Second on some embedded systems
- Third on MS-DOS (early PCs) -part in ROM called BIOS

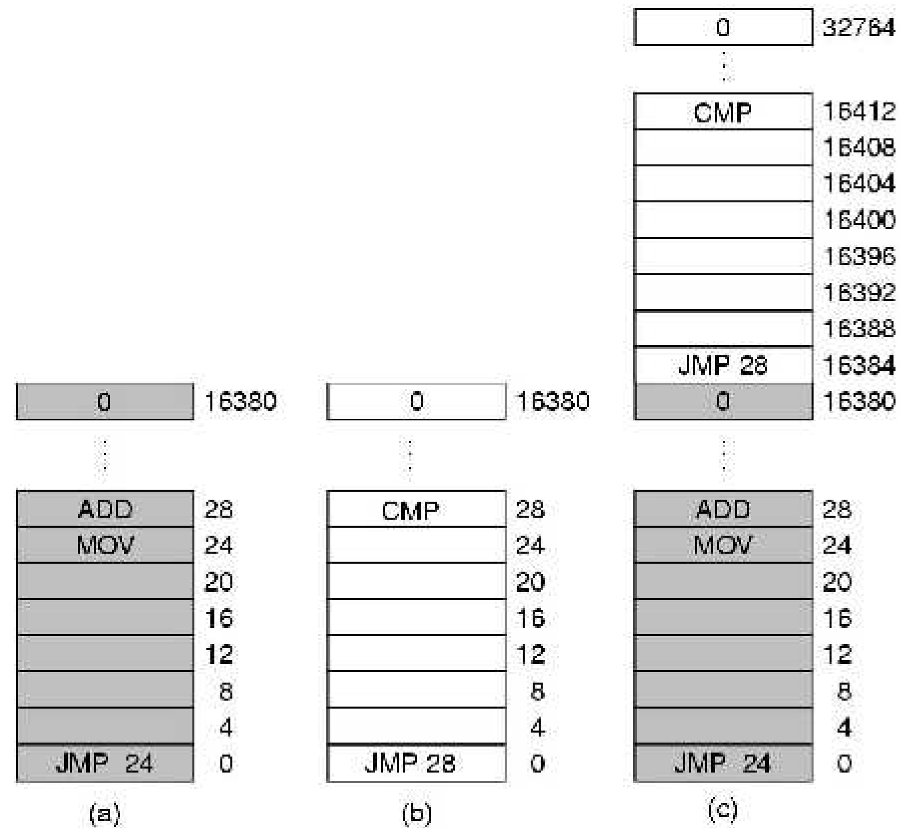
Really want to run more than one program

- Could swap new program into memory from disk and send old one out to disk
- Not really concurrent

IBM static relocation idea

- IBM 360 -divide memory into 2 KB blocks, and associate a 4 bit protection key with chunk. Keep keys in registers.
- Put key into PSW for program
- Hardware prevents program from accessing block with another protection key

Problem with relocation



JMP 28 in program (b) trashes ADD instruction in location 28

Program crashes

Static relocation

- Problem is that both programs reference absolute physical memory.
- Static relocation- load first instruction of program at address x, and add x to every subsequent address during loading
  - This is too slow and
  - Not all addresses can be modified
    - Mov register 1,28 can’t be modified

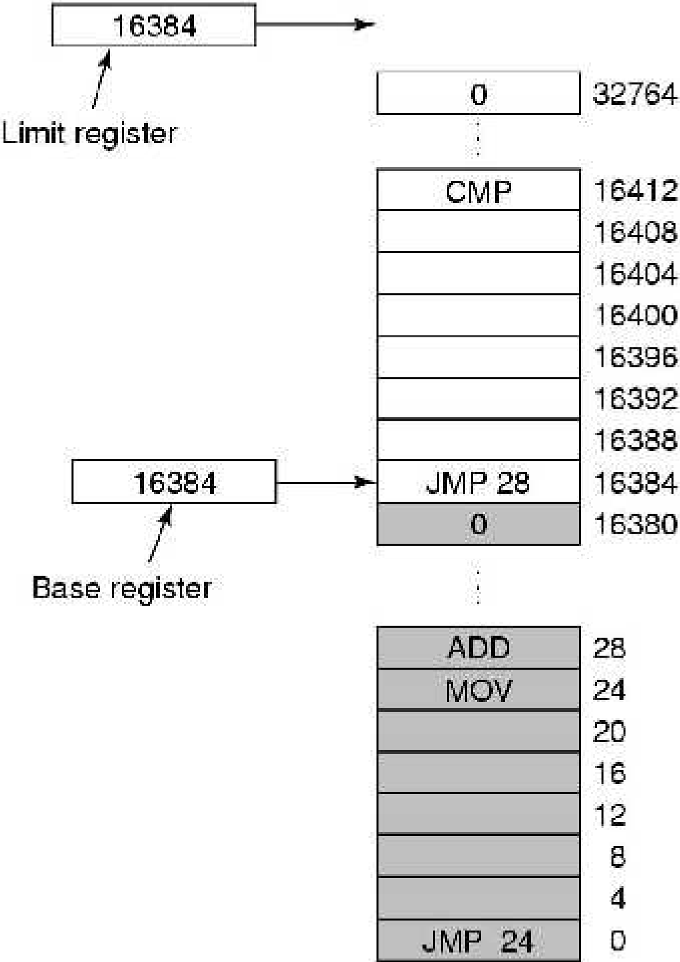
Address Space

- Create abstract memory space for program to exist in
  - Each program has its own set of addresses
  - The addresses are different for each program
  - Call it the address space of the program

Base and Limit Registers

- A form of dynamic relocation
- Base contains beginning address of program
- Limit contains length of program
- Program references memory, adds base address to address generated by process. Checks to see if address is larger then limit. If so, generates fault
- Disadvantage-addition and comparison have to be done on every instruction
- Used in the CDC 6600 and the Intel 8088

Base and Limit Registers



(c)

Add 16384 to JMP 28. Hardware adds 16384 to 28 resulting in JMP 16412

How to run more programs then fit in main memory at once

- Can’t keep all processes in main memory
  - Too many (hundreds)
  - Too big (e.g. 200 MB program)
- Two approaches
  - Swap-bring program in and run it for awhile
  - Virtual memory-allow program to run even if only part of it is in main memory