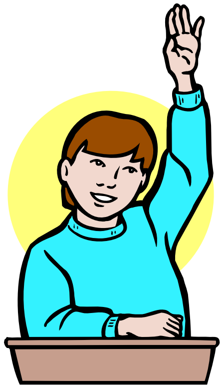
Chapter 2

Processes and Threads

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

Processes

- **A process is basically a running program**
  - **What is the program?**
- **Single-process OS**
  - **MS-DOS**
- **Multi-process OS**
  - **Most OSes**

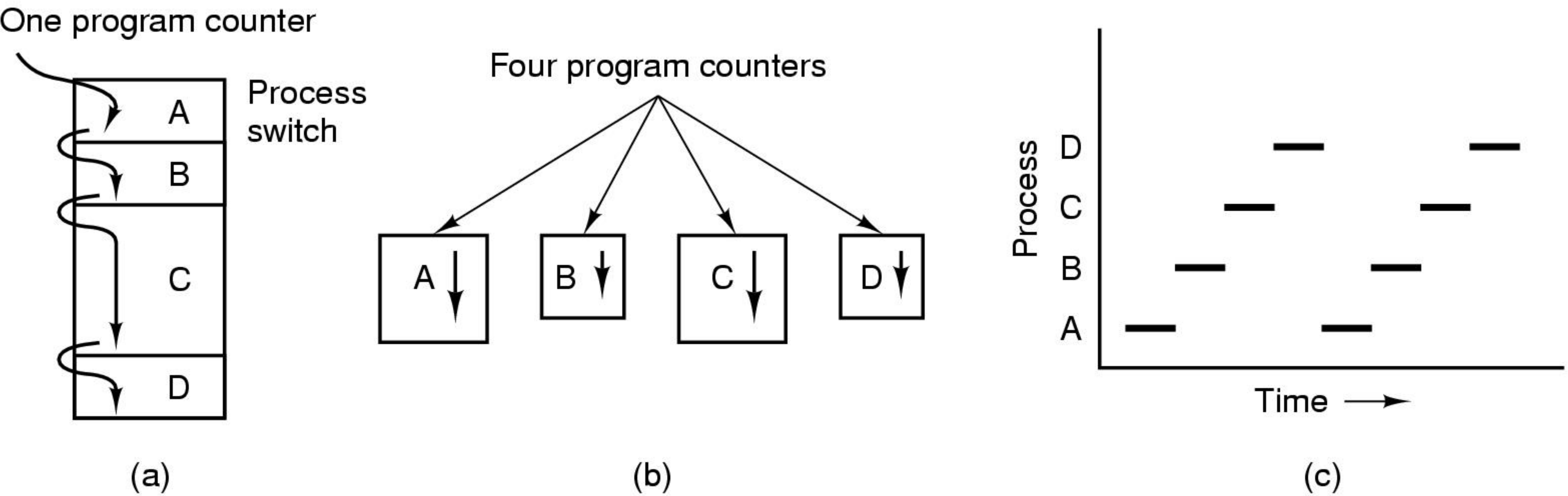


Several processes run together

- **Every process feels that the computer belongs to itself.**
  - **has its own address space**
    - **a list of memory locations from some minimum (usually 0) to some maximum**
  - **can input and output freely**
  - **is executed by CPU continually**
- **General speaking, a process is running on a virtual machine powered by OS.**

CPU switch

- **There is only *ONE* CPU which can run only *ONE* instruction at any instant.**
- **In fact, the CPU switches back and forth from process to process.**



Real-time Problem

- **We can't forecast**
  - **How long will the process run exactly**
  - **When will one instruction run**
- **Thus, there is real-time scheduling**
  - **Run the most important available process**

Process Creation

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system call
  - User request to create a new process
- Technically, a new process is created by having an existing process execute a process creation system call

Process Creation System Call in POSIX

- ```
  #include <sys/types.h>
  #include <unistd.h>
  pid_t fork(void);
  ```

- ```
  #include <unistd.h>
  int execve(const char *filename,
             char *const argv[],
             char *const envp[]
  );
  ```

fork() & execve()

- A stripped down shell:

```
while (TRUE) {                              /* repeat forever */
  type_prompt( );                           /* display prompt */
  read_command(command, parameters)/* input from terminal */

  if (fork() != 0) {                        /* fork off child process */
    /* Parent code */
    waitpid( -1, &status, 0);    /* wait for child to exit */
  } else {
    /* Child code */
    execve (command, parameters, 0);   /* execute command */
  }
}
```

Process Creation API inWin32

- ```
  BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
  );
  ```
  ```

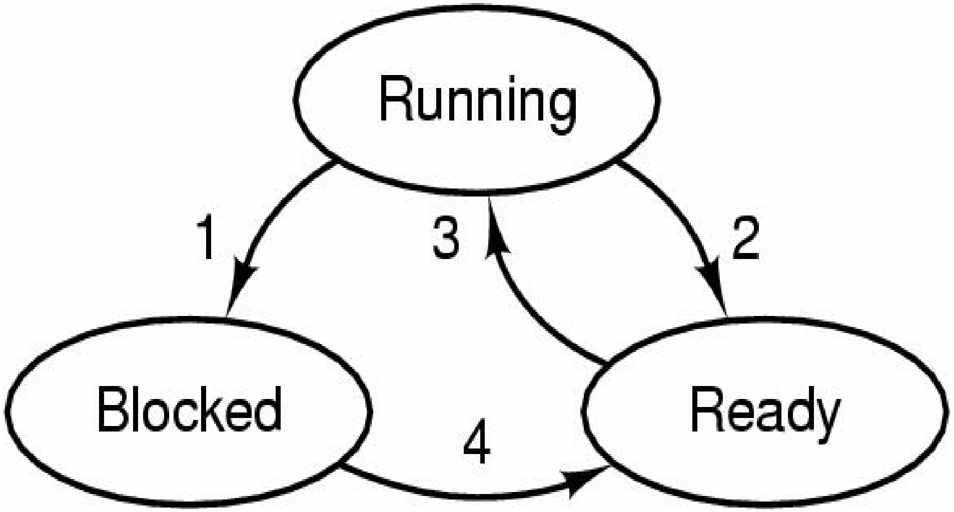
Process Termination

- **Conditions which terminate processes**
  - **Normal exit (voluntary)**
    - `exit()` **and** `ExitProcess()`
  - **Error exit (voluntary)**
  - **Fatal error (involuntary)**
  - **Killed by another process (involuntary)**
    - `kill()` **and** `TerminateProcess()`

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX/Linux calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

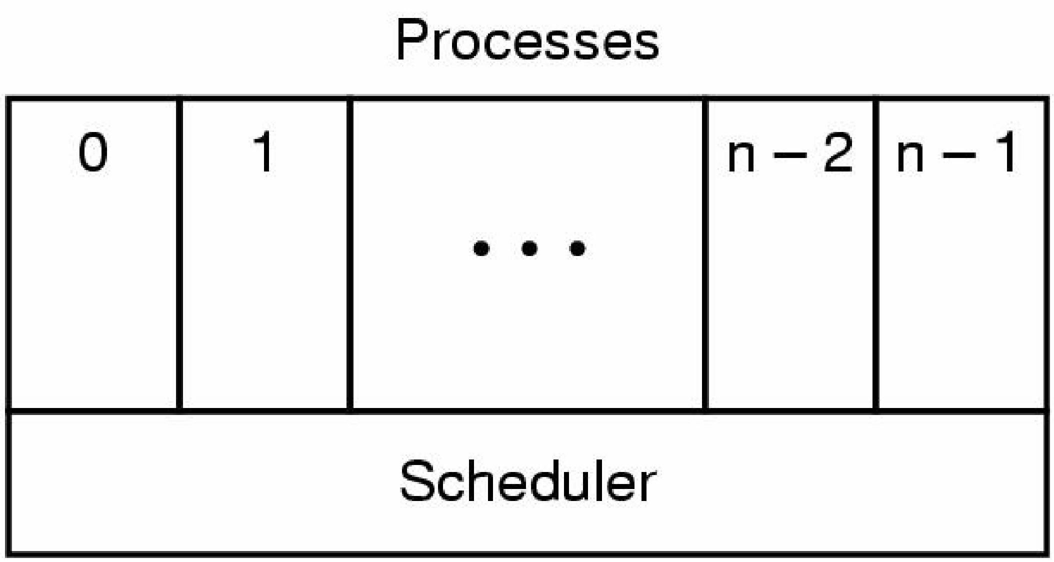
Process States (1)



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

- **Possible process states**
  - **running (运行)**
  - **blocked (阻塞)**
  - **ready (就绪)**
- **Transitions between states are as shown**

Process States (2)



- **Lowest layer of process-structured OS**
  - **handles interrupts, scheduling**
- **Above that layer are sequential processes**

Implementation of Processes (1)

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	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		

Fields of a process table entry

Implementation of Processes (2)

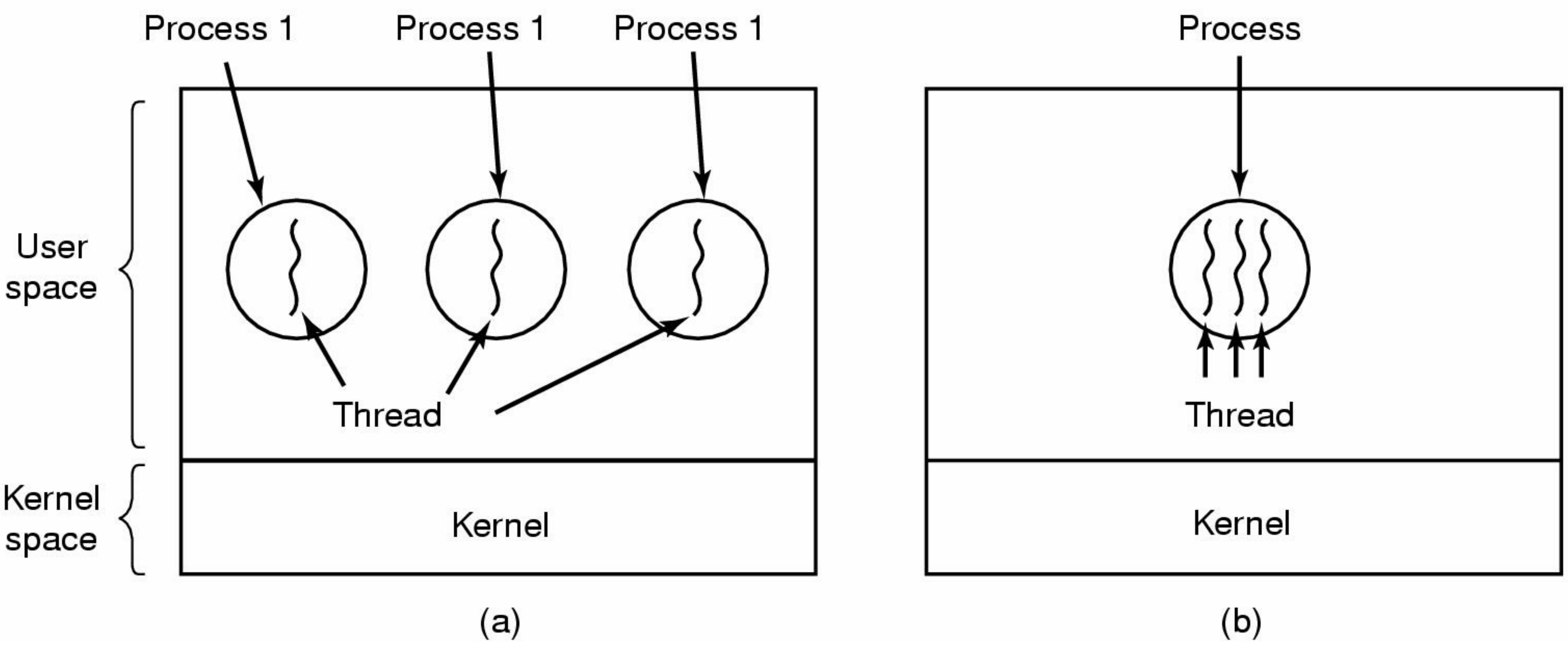
1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is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

Skeleton of what lowest level of OS does when an interrupt occurs

Threads (线程)

- What does a process have?
  - an address space
  - other resources
  - one thread
- Thread has
  - a program counter, registers, a stack
- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU

The Thread Model (1)



(a) Three processes each with one thread

(b) One process with three threads

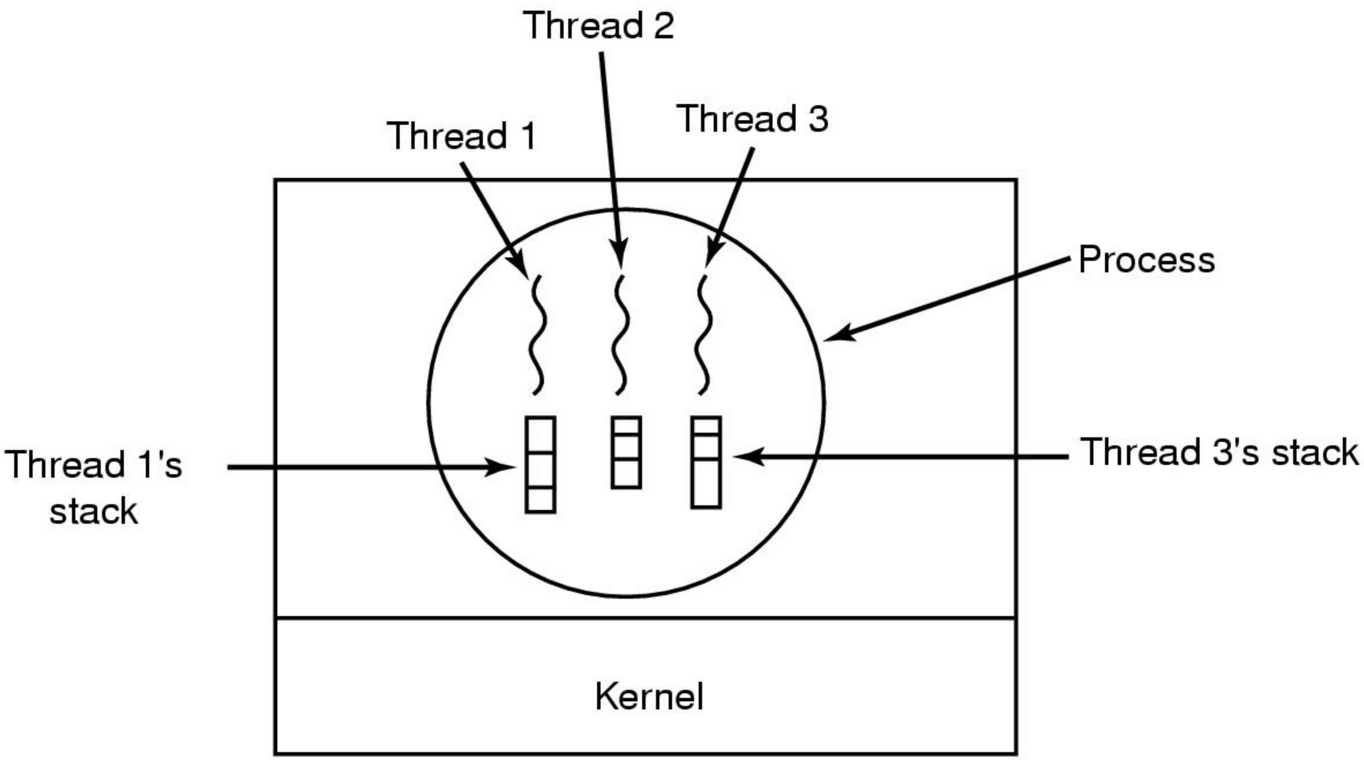
The Thread Model (2)

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	

Items shared by all threads in a process

Items private to each thread

The Thread Model (3)



Each thread has its own stack

Multithreading

- **The threads take turns running**
- **Every thread can access every memory address within the process' address space**
- **Also running, blocked and ready**
- **Threads can work together closely to perform some task**
  - **background work**

System Call About Threads

```c
#include <pthread.h>

int pthread_create(
    pthread_t* thread,
    pthread_attr_t* attr,
    void* (*start_routine)(void*),
    void* arg
);

void pthread_exit(void* retval);

int pthread_cancel(pthread_t thread);

void pthread_testcancel(void);
```

API About Threads

```
#include <Winbase.h>

HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);

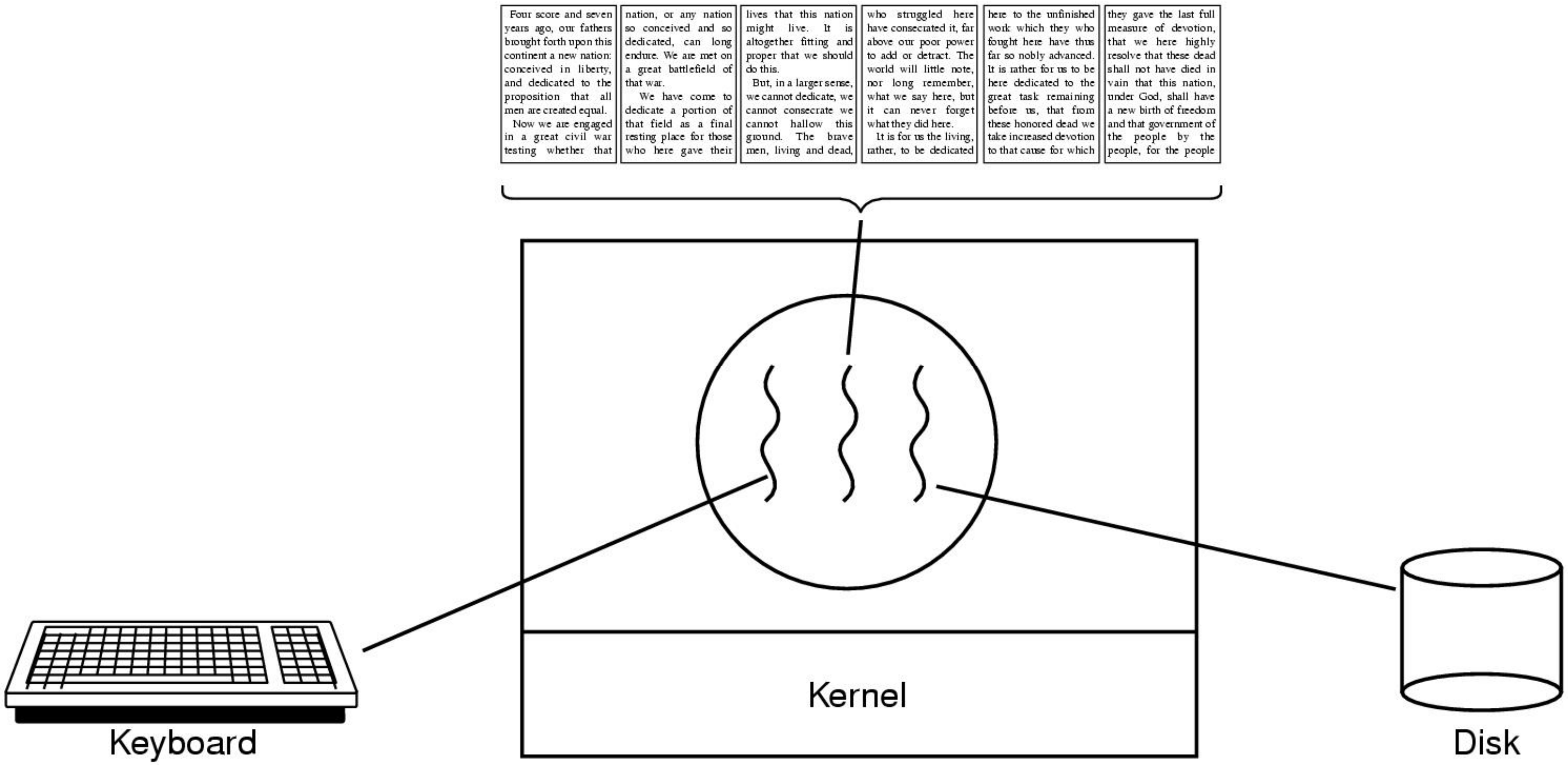
void ExitThread( DWORD dwExitCode );

BOOL TerminateThread(HANDLE hThread,
                     DWORD dwExitCode);
```

Why Thread?

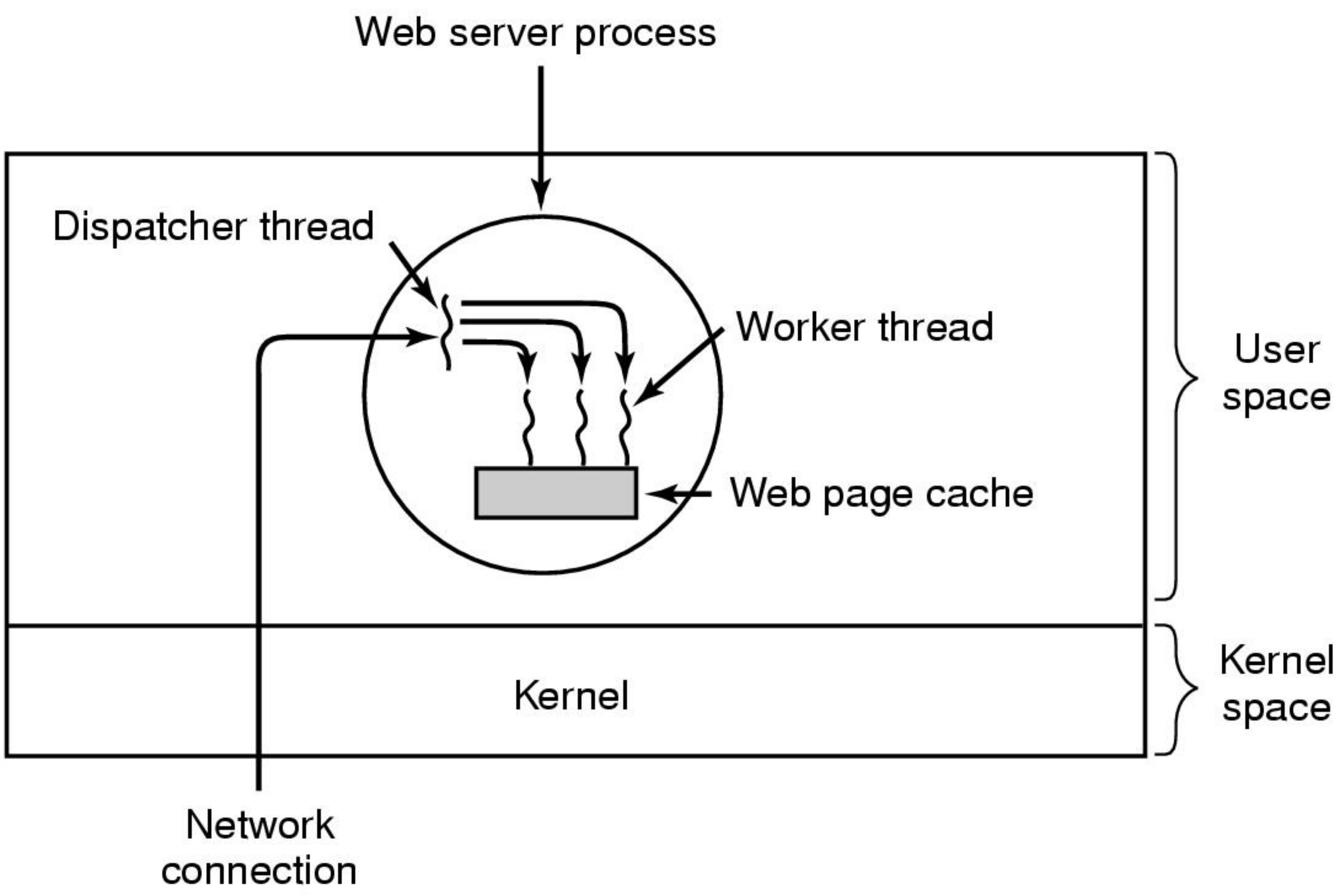
- Processes can’t share many resources
- Easier to create and destroy
- Speed up the application
  - More threads, more performance?

Thread Usage (1)



A word processor with three threads

Thread Usage (2)



A multithreaded Web server

Thread Usage (3)

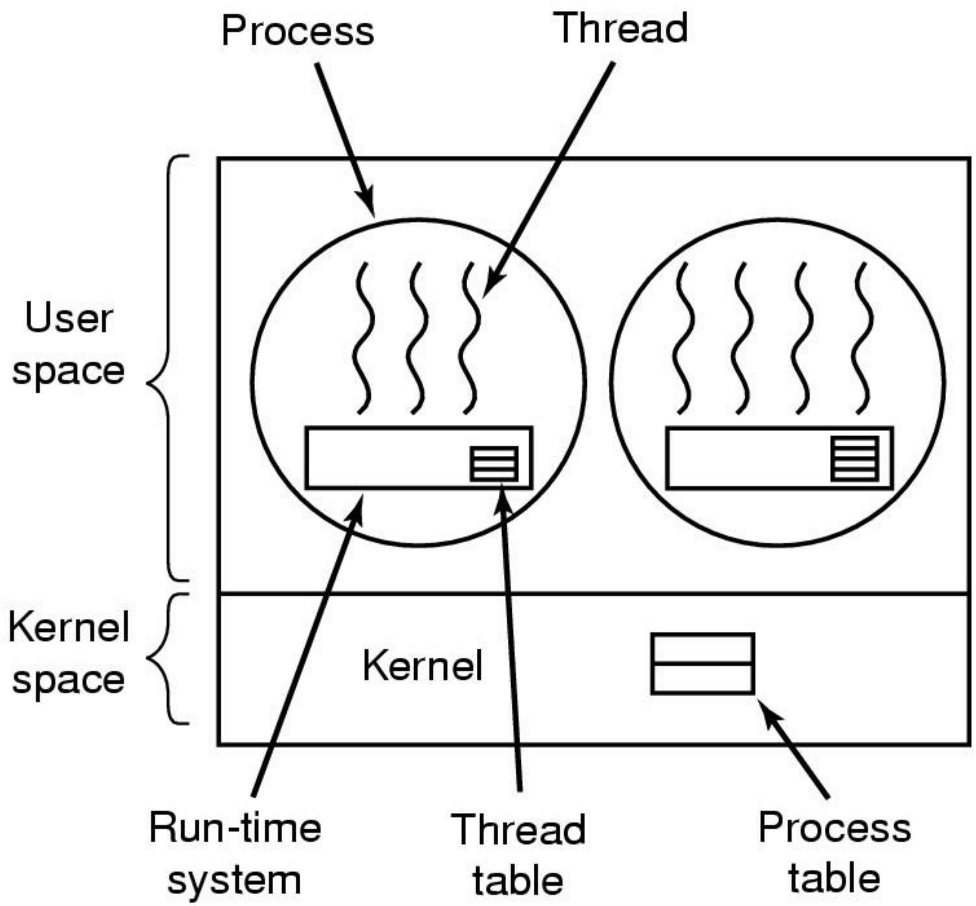
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

Implementing Threads

- In user space
- In kernel

Implementing Threads in User Space

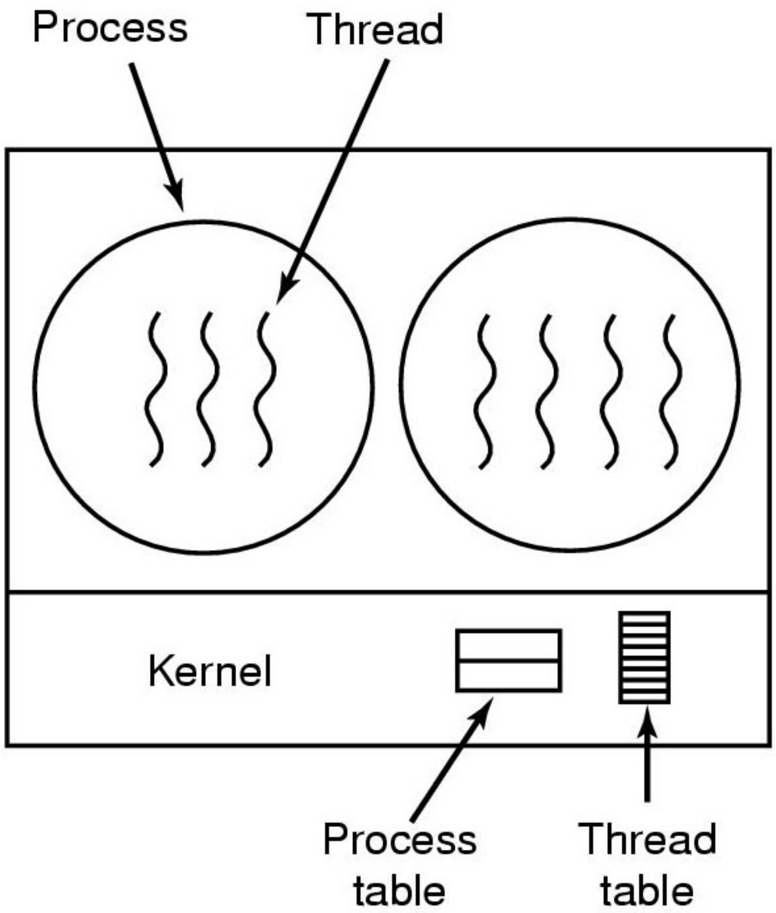


A user-level threads package

Implementing Threads in User Space

- **Advantages**
  - Can be implemented on an OS that doesn't support thread
  - No trap and context switch is needed for scheduler
  - Each process can have its own customized scheduling algorithm
- **Problems**
  - One thread is blocked, others are blocked too
  - How to switch context?

Implementing Threads in the Kernel

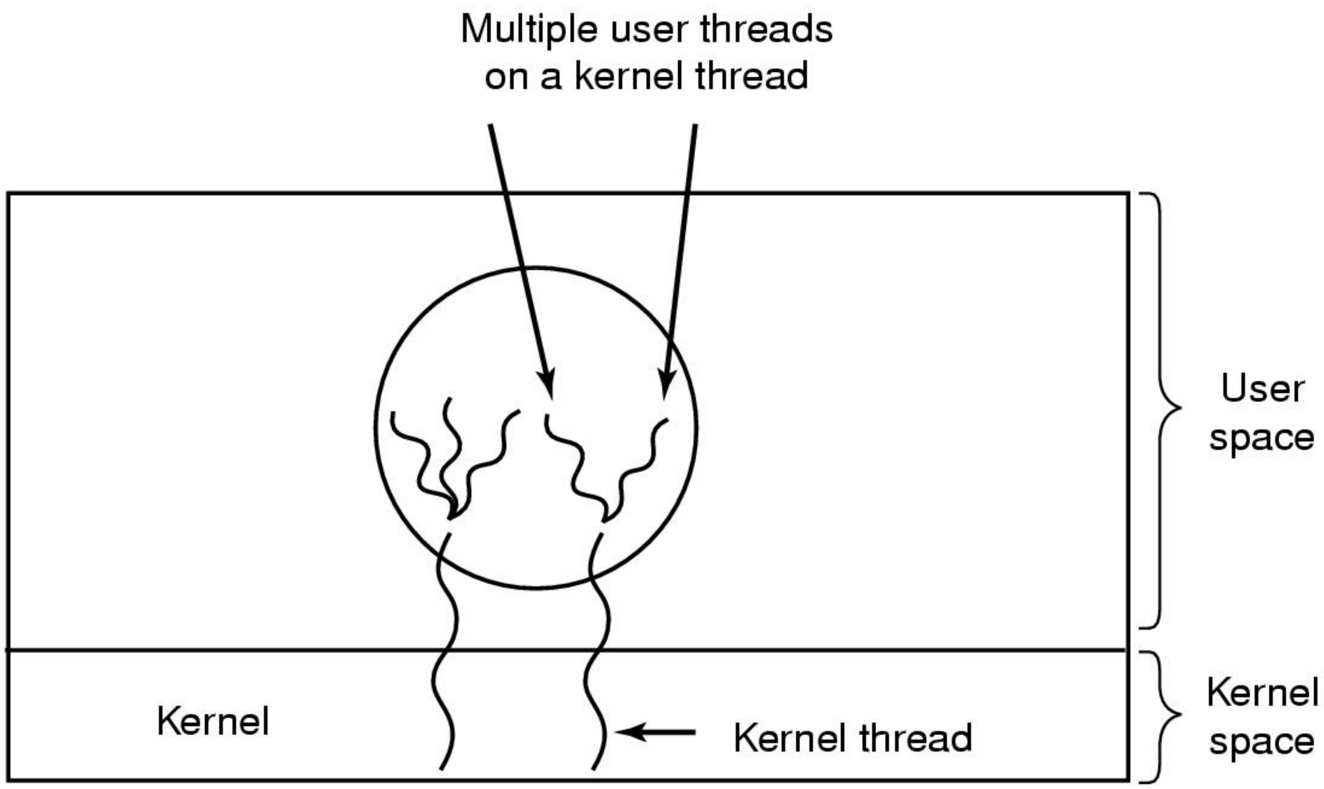


A threads package managed by the kernel

Implementing Threads in the Kernel

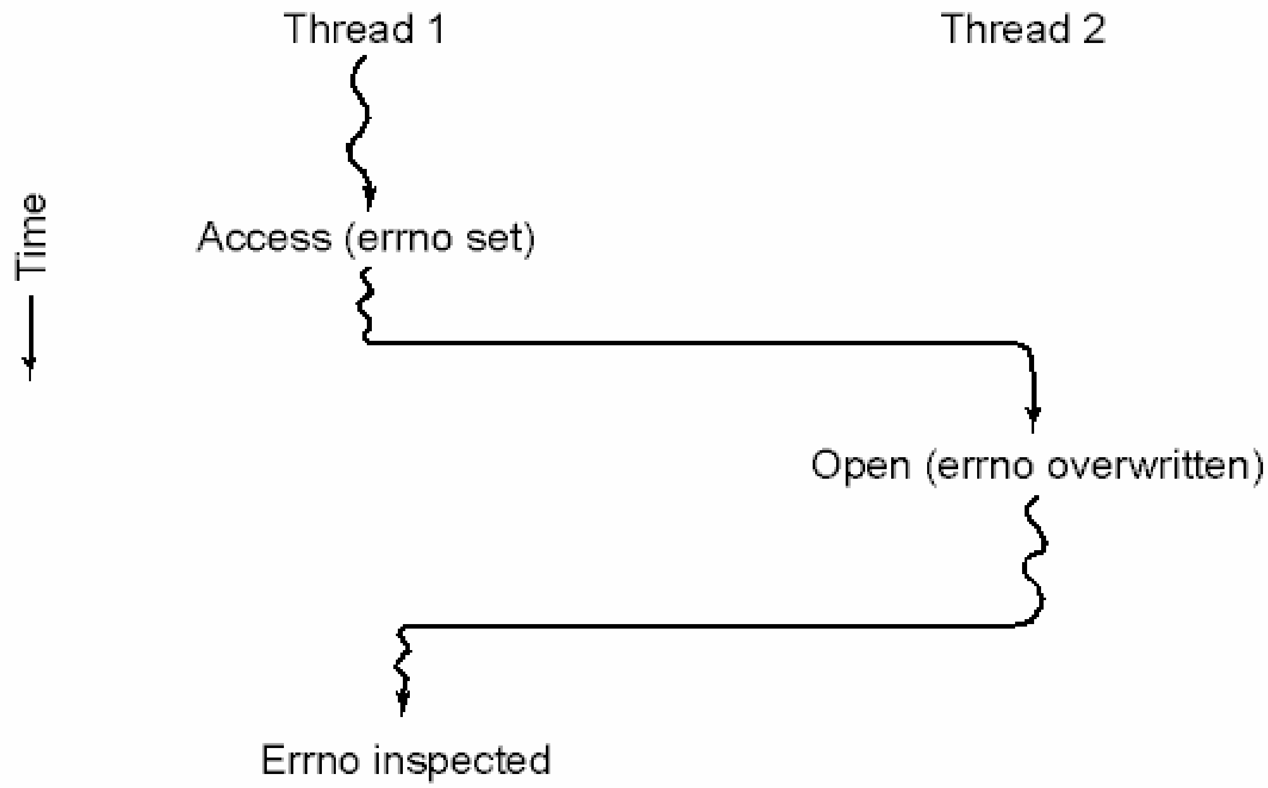
- **Blocked thread doesn't affect others**
- **More overhead**

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

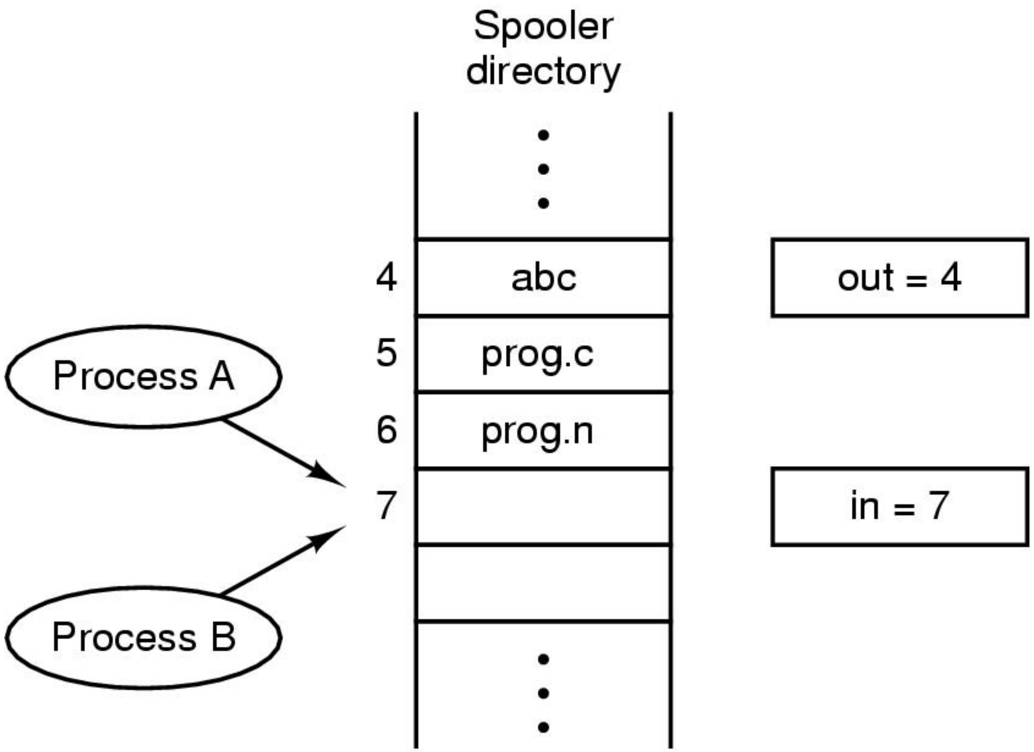
Making Single-Threaded Code Multithreaded



Conflicts between threads over the use of a global variable

Interprocess Communication

Race Conditions

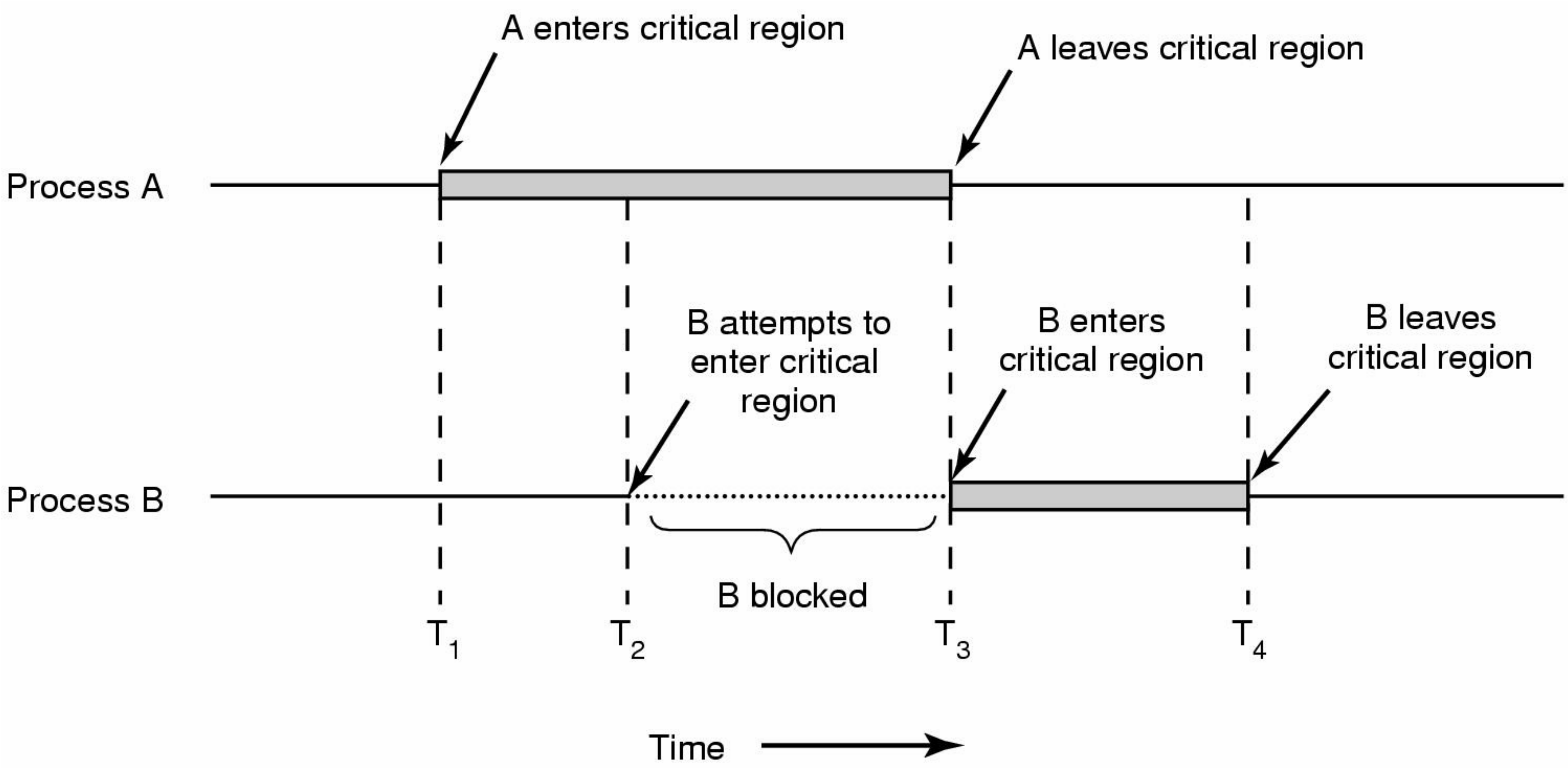


Two processes want to access shared memory at same time

Critical Regions (临界区)

- **Mutual exclusion (互斥现象)**
  - **Some way of making sure that if one process is using a shared variable or file, the other processes mustn’t use it**
- **Critical Region**
  - **That part of the program where the shared memory or file is accessed**
  - **To avoid race, no two processes are ever in their critical regions at the same time**

Critical Regions



Mutual exclusion using critical regions

Four conditions

- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region

Disabling Interrupts

- Each process disable all interrupts just after entering its critical region and re-enable them just before leaving it
- With interrupts disabled,
  - CPU will not be switched
  - application can affect the whole system
- Only kernel can disable interrupts

Lock Variables

- **Basic idea**
- `int lock_variable = 0;`

```
……
thread()
{
    ……
    while (lock_variable != 0)
        ;
    lock_variable = 1;
    critical_region();
    lock_variable = 0;
    ……
}
```

Mutual Exclusion with Busy Waiting

```
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 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
```

```
enter_region();
critical_region();
leave_region();
```

Entering and leaving a critical region using the TSL instruction

Sleep and Wakeup

```
#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 */
     }
}
```

Producer-consumer (生产者—消费者) problem with fatal race condition

Producer

```
#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? */
    }
}
```

Consumer

```
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 */
        }
}
```

Producer and Consumer

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

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

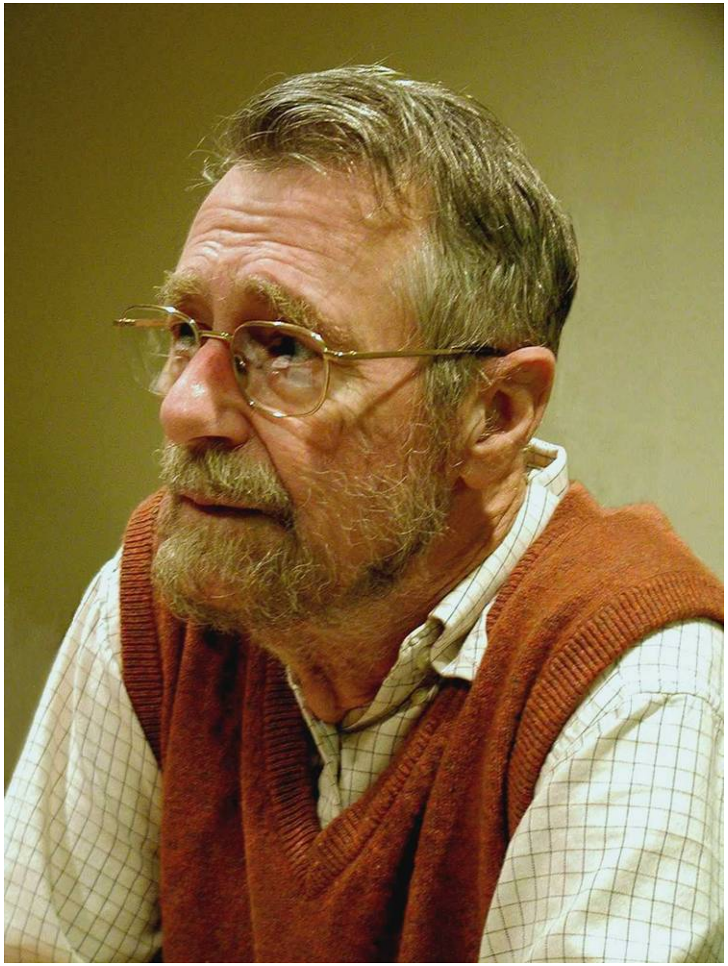
```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count − 1;
                if (count == N − 1) wakeup(producer);
                consume_item(item);
        }
}
```

PV Atomic Action and Semaphore (PV原语操作和信号量)

```
• down(int *sem) {
      if (*sem == 0)
          sleep();
      (*sem)--;
  }

• up(int *sem) {
      (*sem)++;
      wakeup();
  }
```



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 */
     }
}
```

The producer-consumer problem using semaphores

Semaphores

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

mutex is used to ensure only one process can enter the critical region at the same time.

It is called binary semaphore (二元信号量)

Mutexes (互斥锁)

```
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 later
ok:     RET | return to caller; critical region entered


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

Implementation of *mutex_lock* and *mutex_unlock*

API About Critical Region

- **VOID InitializeCriticalSection(**
  LPCRITICAL_SECTION *lpCriticalSection*);
- **VOID EnterCriticalSection(**
  LPCRITICAL_SECTION *lpCriticalSection*);
- **BOOL TryEnterCriticalSection(**
  LPCRITICAL_SECTION *lpCriticalSection*);
- **VOID LeaveCriticalSection(**
  LPCRITICAL_SECTION *lpCriticalSection*);
- **VOID DeleteCriticalSection(**
  LPCRITICAL_SECTION *lpCriticalSection*);

API About Critical Region

- **HANDLE CreateMutex(**
  LPSECURITY_ATTRIBUTES ***lpMutexAttrs*****,**
  BOOL ***bInitialOwner*****,**
  LPCTSTR ***lpName*****);**
- **HANDLE OpenMutex(**
  DWORD ***dwDesiredAccess*****,**
  BOOL ***bInheritHandle*****,**
  LPCTSTR ***lpName*****);**
- **DWORD WaitForSingleObject(**
  HANDLE ***hHandle*****,**
  DWORD ***dwMilliseconds*****);**
- **BOOL ReleaseMutex(HANDLE** ***hMutex*** **);**
- **BOOL CloseHandle(HANDLE** ***hObject*****);**

API About Critical Region

- ```
  HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTSTR lpName);
  ```
- ```
  HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName);
  ```
- ```
  DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds);
  ```
- ```
  BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount);
  ```
- ```
  BOOL CloseHandle( HANDLE hObject );
  ```
  ```

System Call About Critical Region

- **pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;**
- **int pthread_mutex_init(pthread_mutex_t *_mutex_, const pthread_mutexattr_t *_mutexattr_);**
- **int pthread_mutex_lock(pthread_mutex_t *_mutex_);**
- **int pthread_mutex_trylock(pthread_mutex_t *_mutex_);**
- **int pthread_mutex_unlock(pthread_mutex_t *_mutex_);**
- **int pthread_mutex_destroy(pthread_mutex_t *_mutex_);**

System Call About Critical Region

```
• pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
• int pthread_cond_init(
    pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);
• int pthread_cond_signal(
    pthread_cond_t *cond);
• int pthread_cond_broadcast(
    pthread_cond_t *cond);
• int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
• int pthread_cond_timedwait(
    pthread_cond_t   *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
• int pthread_cond_destroy(
    pthread_cond_t *cond);
```

System Call About Critical Region

- ```
  int semget(
    key_t key,
    int nsems,
    int semflg
  );
  ```
- ```
  int semop(
    int semid,
    struct sembuf *sops,
    unsigned nsops
  );
  ```
- ```
  int semctl(
    int semid,
    int semnum,
    int cmd,
    union arg
  );
  ```
- ```
  int fcntl(
    int fd,
    int cmd,
    struct flock *lock
  );
  ```

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a 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;
```

```
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;
```

- **Outline of producer-consumer problem with monitors**
  - **only one monitor procedure active at one time**
  - **buffer has *N* slots**

Monitors (3)

```
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 {
           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
        }
```

Solution to producer-consumer problem in Java (part 1)

Monitors (4)

```
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) {
        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) {};}
  }
}
```

Solution to producer-consumer problem in Java (part 2)

Message Passing

```
#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 */
    }
}

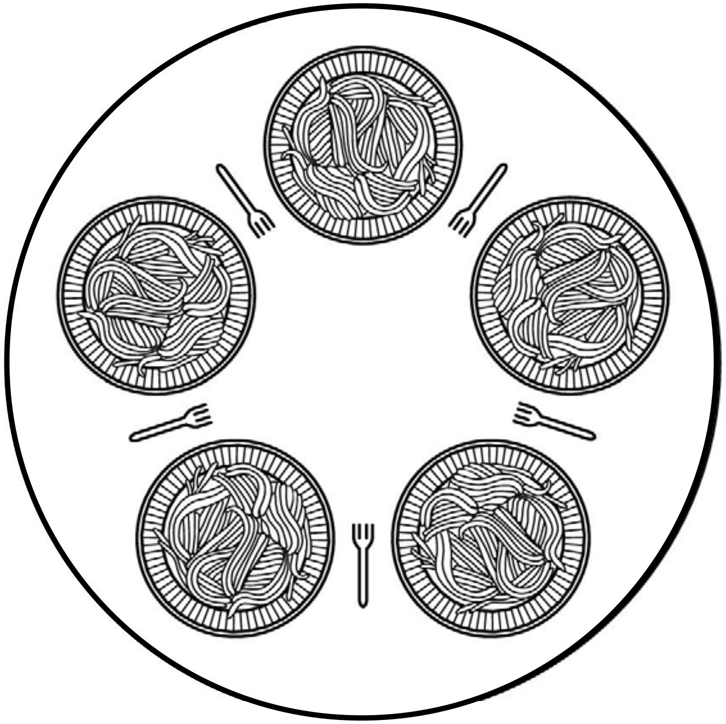
void consumer(void)
{
    int item, i;
    message m;

    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 */
    }
}
```

The producer-consumer problem with N messages

Dining Philosophers (1)

- **Philosophers eat/think**
- **Eating needs 2 forks**
- **Pick one fork at a time**
- **How to prevent deadlock**



Dining Philosophers (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 */
    }
}
```

A <u>non</u>solution to the dining philosophers problem

Dining Philosophers (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 */
     }
}
```

Solution to dining philosophers problem (part 1)

Dining Philosophers (4)

```
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 */
{
      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 */
}
```

Solution to dining philosophers problem (part 2)

Dining Philosophers (5)

```
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]);
    }
}
```

Solution to dining philosophers problem (part 3)

The Readers and Writers Problem

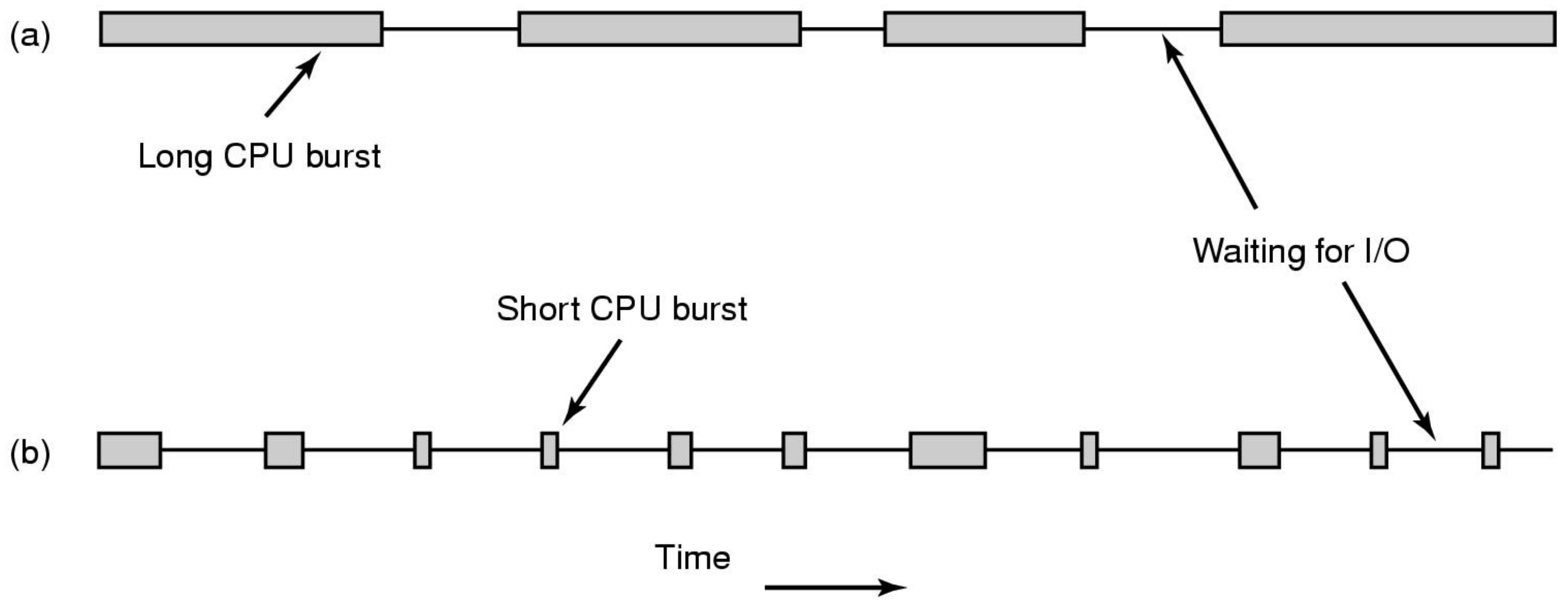
```
void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base( );
        down(&mutex);
        rc = rc – 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read( );
    }
}
```

```
void writer(void)
{
    while (TRUE) {
        think_up_data( );
        down(&db);
        write_data_base( );
        up(&db);
    }
}
```

Scheduling (调度)

- Scheduler
  - makes the choice about which process to run next
- Scheduling algorithm
  - is the algorithm used by scheduler

Process Behavior



- **Bursts of CPU usage alternate with periods of I/O wait**
  - **a CPU-bound process**
  - **an I/O-bound process**

When to Schedule

- **When a new process is created**
- **When a process exits**
- **When a process is blocked**
- **When an I/O interrupt occurs**
- **When a hardware clock interrupt occurs**
  - **Nonpreemptive**
  - **Preemptive**

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

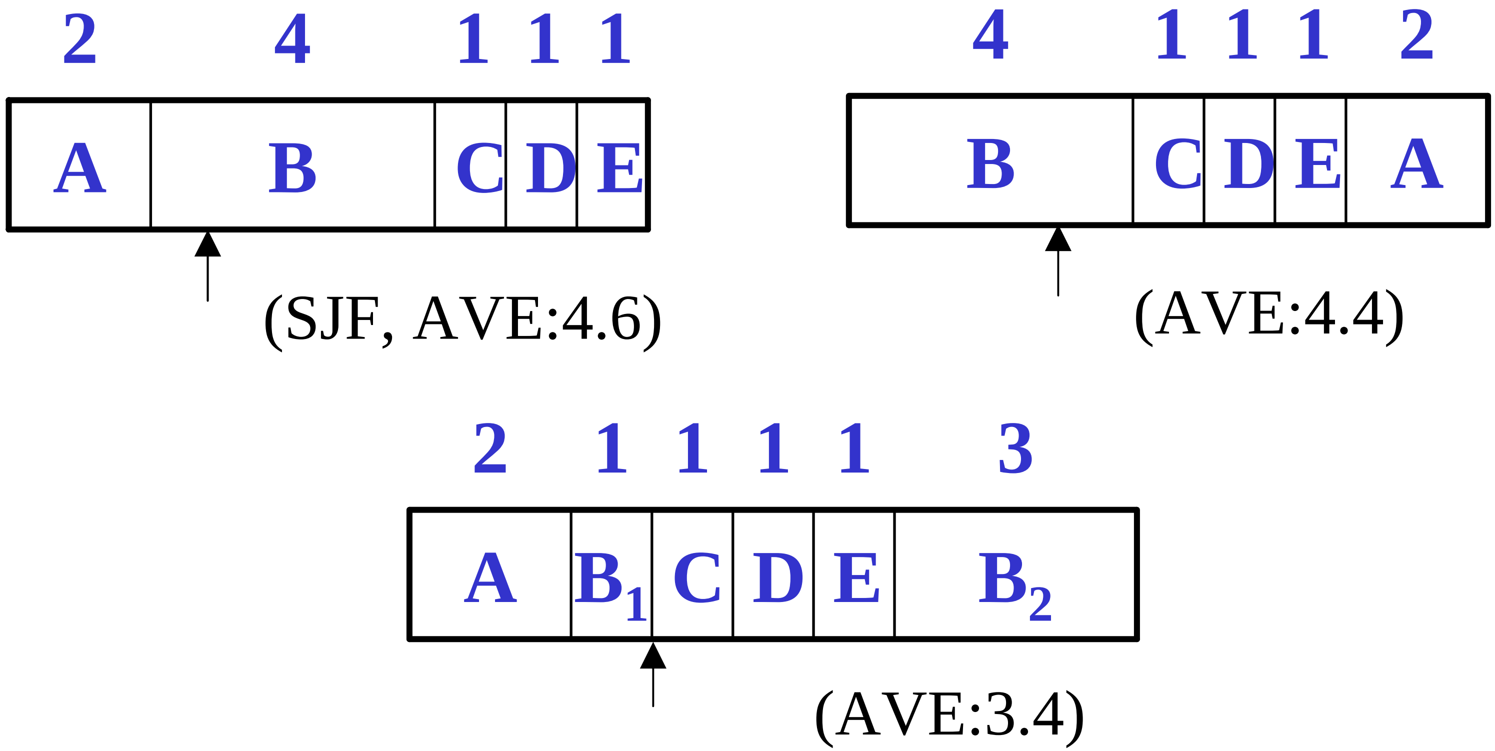
Scheduling in Batch Systems

An example of first-come first-served

Scheduling in Batch Systems

An example of shortest job first scheduling

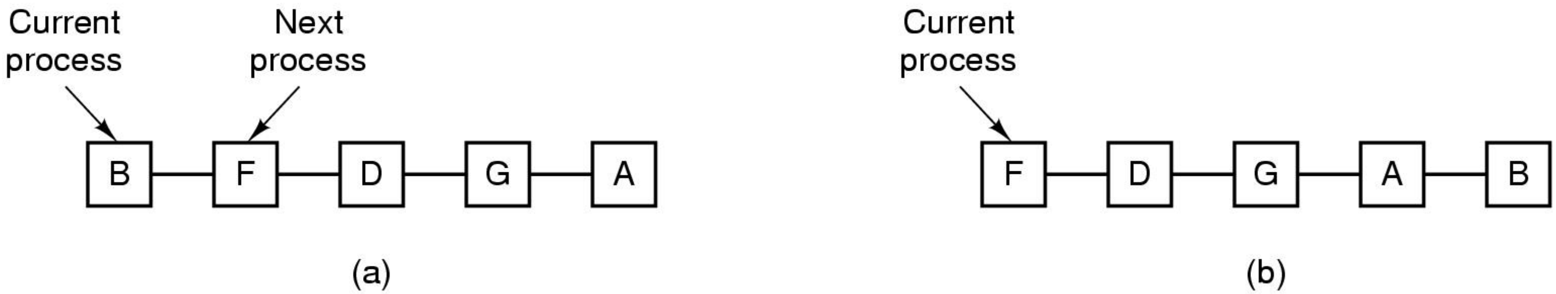
Scheduling in Batch Systems



C, D and E arrive at ↑

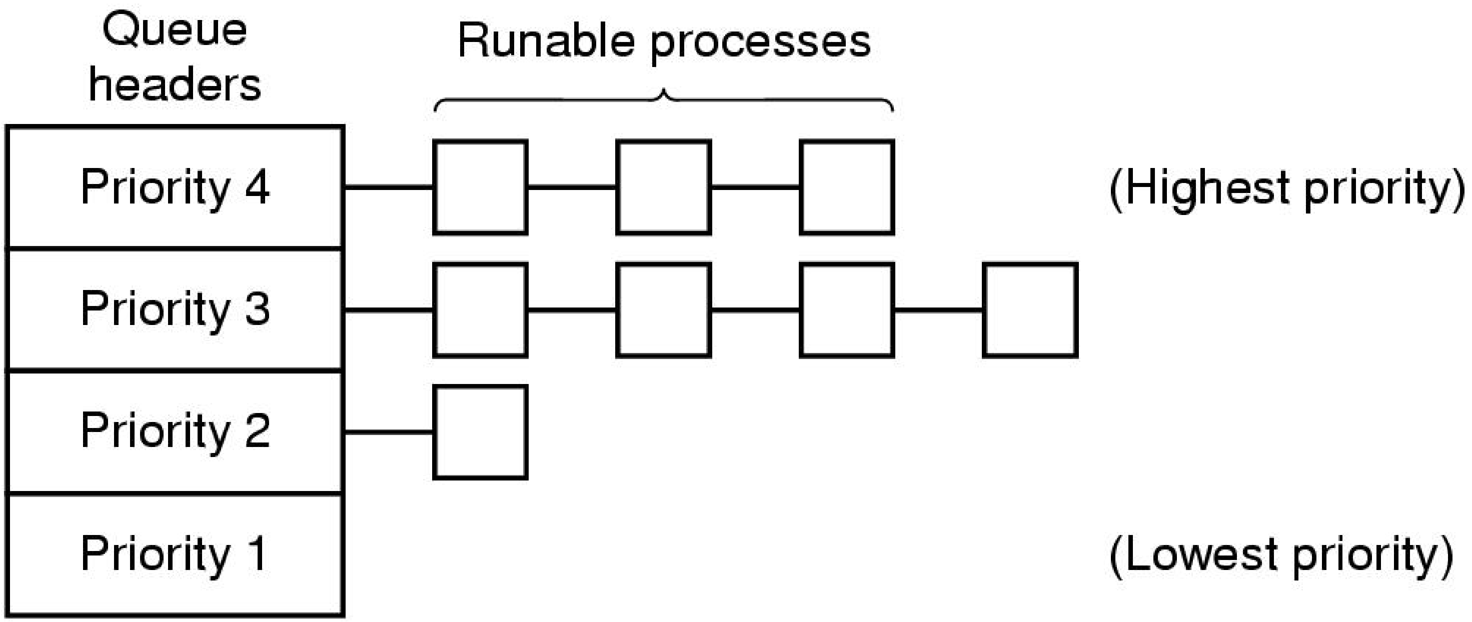
An example of shortest remaining time next

Scheduling in Interactive Systems



- **Round Robin Scheduling**
  - **list of runnable processes**
  - **list of runnable processes after B uses up its quantum**

Scheduling in Interactive Systems



A scheduling algorithm with four priority classes

Scheduling in Real-Time Systems

- The system must react to external events within a fixed amount of time
  - Hard real time
  - Soft real time
- Events
  - Periodic
  - Aperiodic

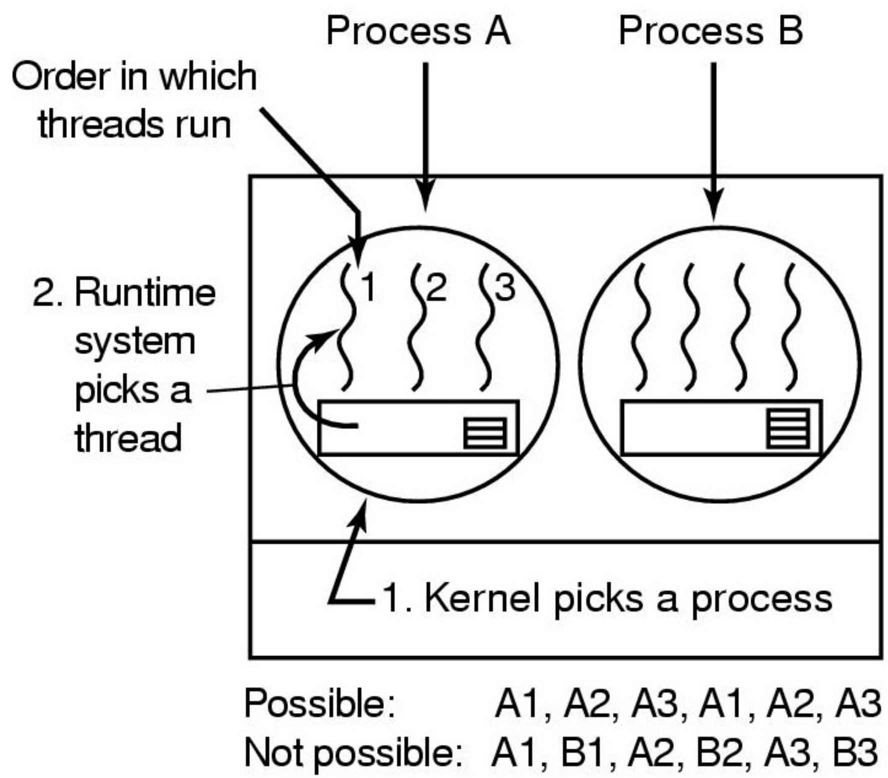
Scheduling in Real-Time Systems

Schedulable real-time system

- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

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

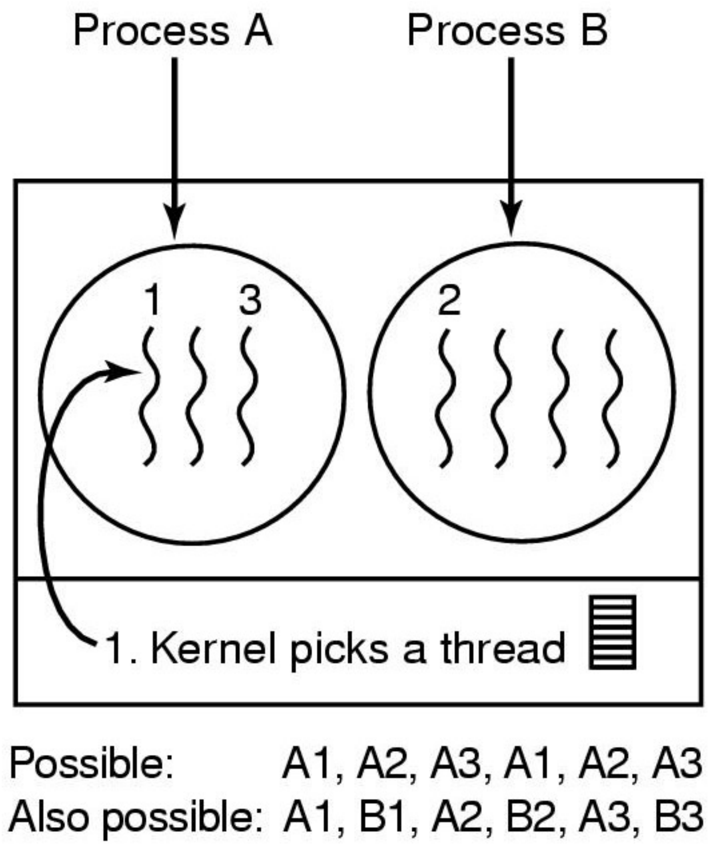
Thread Scheduling (1)



Possible scheduling of user-level threads

- **50-msec process quantum**
- **threads run 5 msec/CPU burst**

Thread Scheduling (2)



Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst