ENCE360
Operating Systems

Processes and Threads

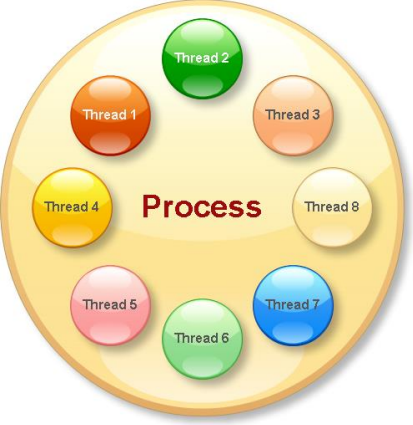
Lab Test and Assignment Dates

- Lab test 5:00-6:30pm Wed 21 Aug
- Assignment due 11pm Thu 17 Oct

Please let me know if you have any clashes with other assessment items on those dates.

UCSA – become a class rep

- http://ucsa.org.nz/student-support/class-reps/signup



Processes and Threads

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

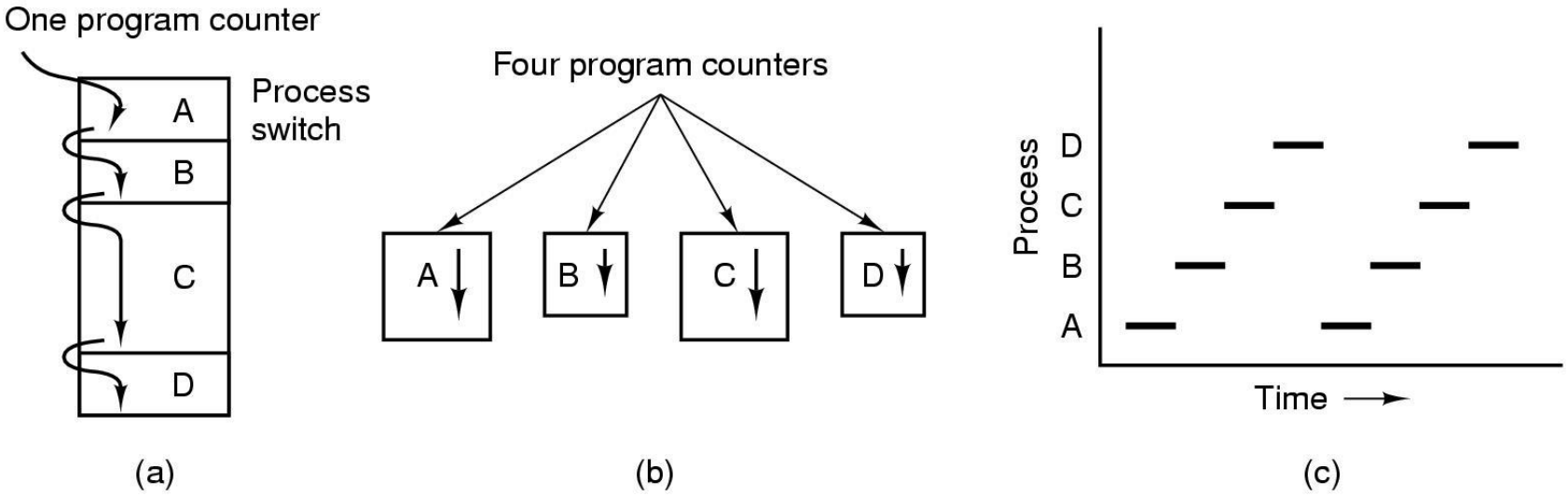
Chapter 2

MODERN OPERATING SYSTEMS (MOS)

By Andrew Tanenbaum

Processes

The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Process Creation

Principal events that cause process creation:

- System initialization
- Execution of a process creation system
- User request to create a new process
- Initiation of a batch job

Process Termination

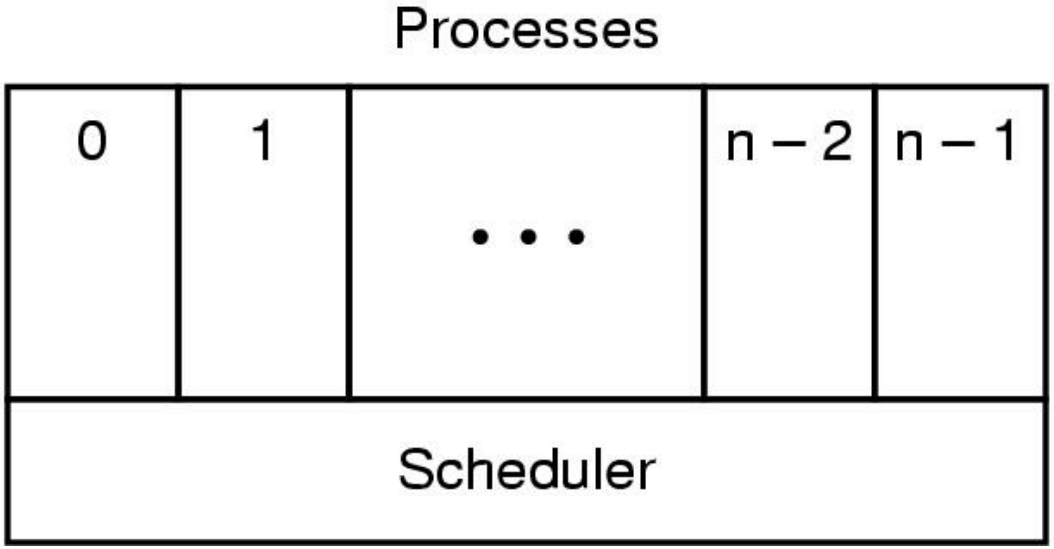
Conditions which terminate processes:

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

Process Hierarchies

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

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 the lowest level of OS does when an interrupt occurs

Processes

- A process is one of the fundamental units within an operating system.
- A process is an activity on a computer which involves a program, input, output and a state

Definition:

A process is a program in execution and comprises:

- the program instructions
- data
- program counter and status word,
- stack pointer, stack, and
- other registers
- Other administrative information describing the process
- Associated with process is the address space

Every process has a unique

process identifier (PID)

Process Control Blocks

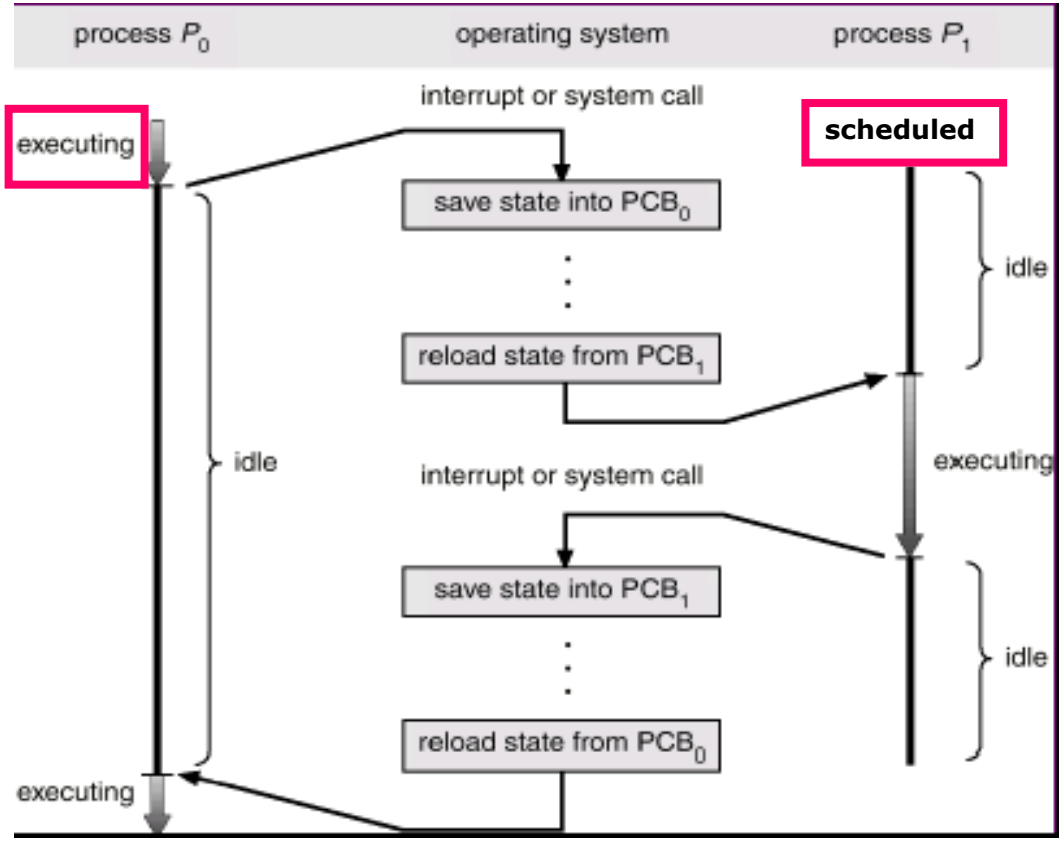
- Elements of a process are stored in a Process Control Block (PCB):
  - Process Control Blocks and the values within these blocks change over time as the process is executed and managed by the OS

Process table: 1 PCB for each process

- Two parts to a process
  - **Sequential execution**
    - No concurrency inside a process
  - **Process state**
    - Everything that interacts with process
      - » Registers
      - » Memory
      - » Files in UNIX

Model for multiprogramming

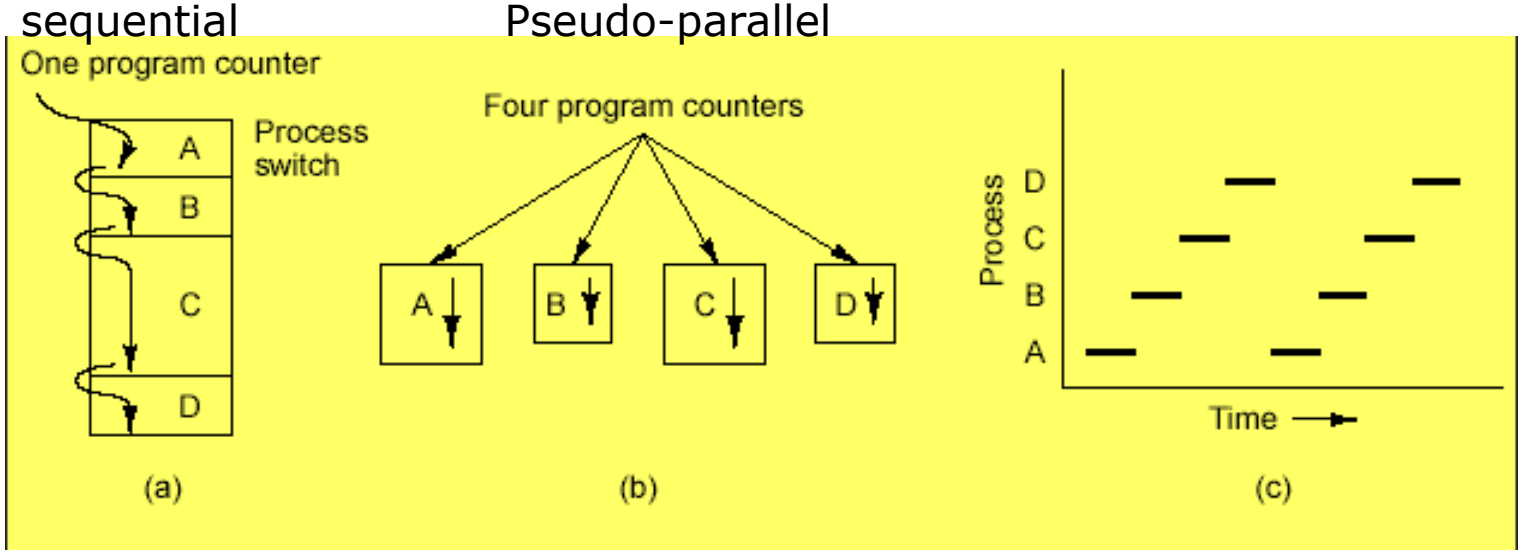
- Multiprogramming systems support:
  - a CPU switches from process to process
  - Each process is executing for tens or hundreds of milliseconds usually until timeslice is up or some other interrupt arrives
    - E.g I/O



Silberschatz & Galvin, 5th Ed, Wiley, Fig 4.3

Operating System Concepts

Pseudo parallelism



Where the CPU contains multiple cores, there will be both parallelism, when two processes are concurrently executing on different cores, and pseudo parallelism as each core switches among different processes.

Suspension of process execution can be for a variety of reasons and the duration of the suspension may also vary and in general is beyond the control
of the process.

Process Lifecycle

Process creation is one result of 4 principle events:

1. **System initialization:**
   - When the operating system boots a variety of processes are created:
     - background processes are active without the need to interact directly with the user
       - Some background processes are also known as daemons.
     - foreground processes require input from the user

2. **Initiation by user:**

   Each time a user enters a shell command or starts an application by, for example, double clicking on an icon.

3. **Initiation of a batch job:**

   Programs to be executed may be queued awaiting the necessary resources. When available, the operating system creates the process(es) for each program.

4. **Execution of a system call:**

   A process may itself initiate other processes to accomplish the assigned activity.

Linux

- **Process hierarchies:**
  - The parent process clones itself to form a child process.
  - Children may themselves act as parents and in doing so they form **Process Hierarchies**

A parent and child share environment strings and open files, thereby allowing parent and child processes to communicate, but they have separate address spaces with the child's initially having the same values as the parent's

- **Items Per Process**
  - Address space
  - Global variables
  - Open files
  - Child processes
  - Pending alarms
  - Signals and signal handlers
  - Accounting information

fork()

The parent uses the *fork()* system call to create a child:

- **The return value from the fork()** system call to the parent process is different to that returned to the child process:
  - **The value returned to the parent is the process identifier (PID) for the child created by the fork() system call**
  - **the value returned to the child is 0**
- **Both the parent and child process execute another program using the *execl()* system call**

Example:

```
main()
{
  int return_value;

  printf("Forking process \n`");
  fork();

  printf("The process id is %d and
                return value is %d\n",
                getpid(), return_value);

  execl("/bin/ls/","-l",0);

  printf("End\n");
}
```

Termination

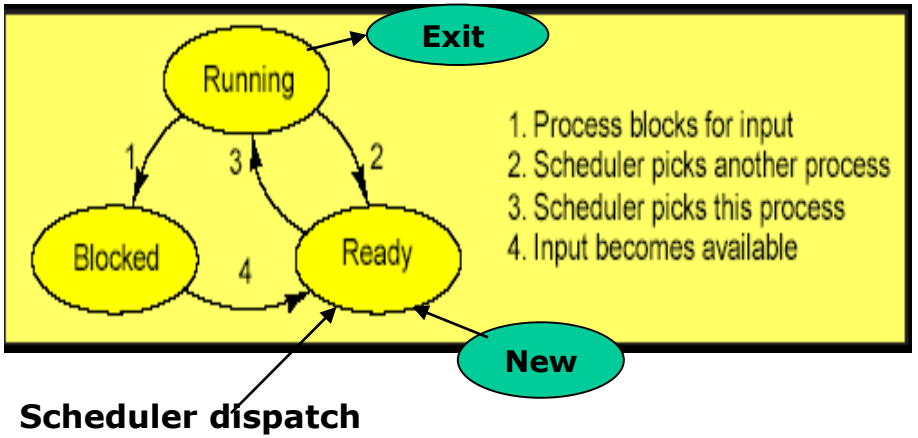
Processes terminate:

- voluntarily by calling a system call:
  - on Linux this is exit()
- involuntarily either:
  - by the operating system because of "misbehaving", or
  - by another process

Process states:

A process may be in one of three states:

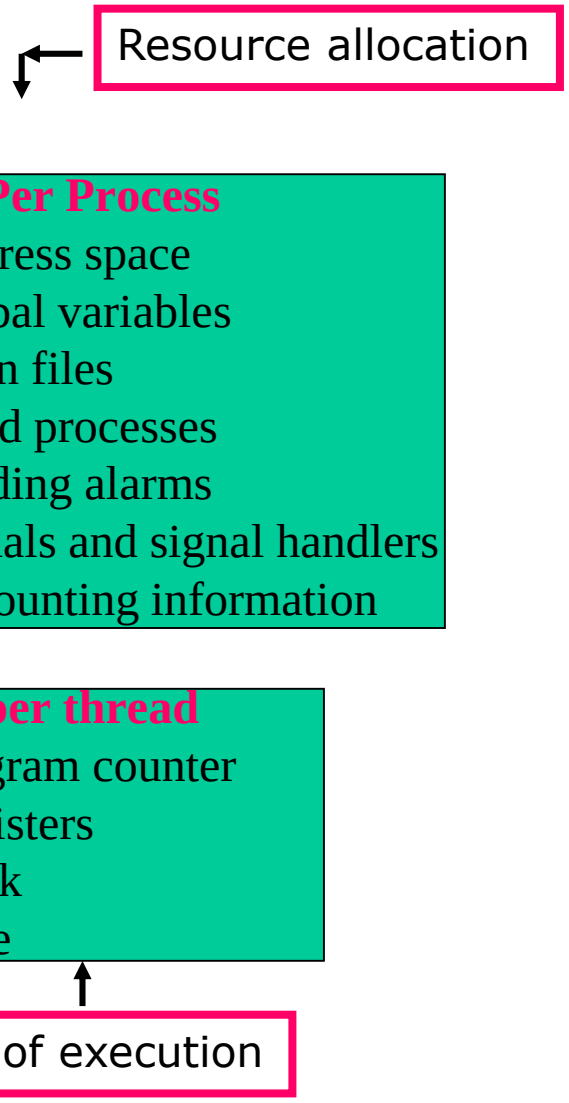
- **Running** – using CPU
- **Ready** – waiting for CPU, on the ready list
- **Blocked** – unable to run until completion of an external event
  - e.g. waiting for I/O or Synchronization with another thread



Threads

Definition:

- A process always has a single thread of execution
- The idea is to allow a process to contain several threads of execution within a single process (multithreading)
- Similarities between threads and processes led to threads being referred to as lightweight processes
- threads are the entities scheduled for execution on the CPU



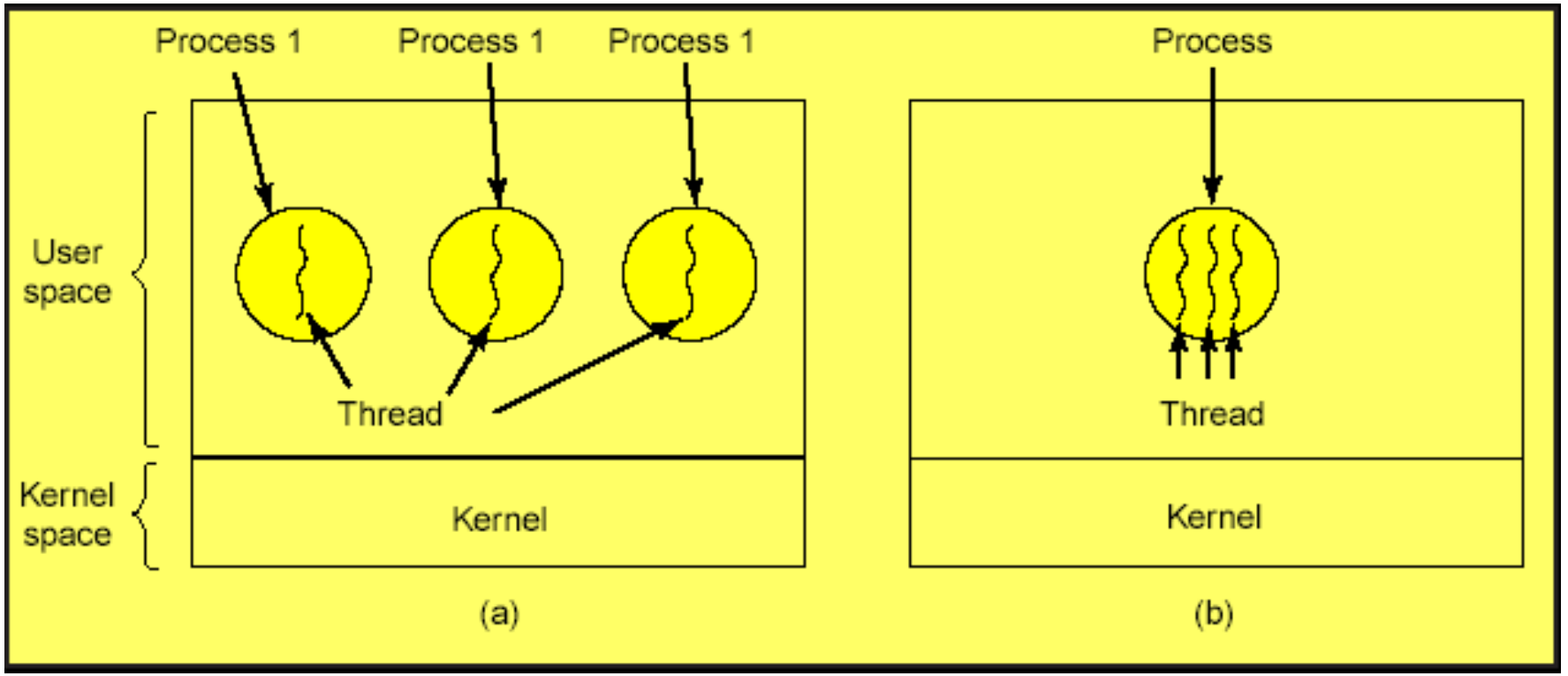
Extending the process model

- LinuxThreads Interface model provides an interface for concurrency within the application process
- The weight of a process is determined by two factors:
  - Amount of information needed to context switch
  - Degree of memory sharing

Process context & context switch

- The system is executed in in the context of the process
- The context includes (state):
  - Global registers
  - Stacks
  - Memory map
- Preempting the process:
  - **OS** must remember the **process state**
  - The process state must be restored when the process is rescheduled:
    - **CONTEXT SWITCH**

Model



Threads of all processes compete among themselves for time on the CPU and like processes, each thread will at any given moment in time be in one of the running, ready, or blocked states

Thread creation and termination

- The basic runtime library calls for managing **Linux threads** are:
  - ***pthread_create***
    - creates a thread to execute a specified function.
  - ***pthread_exit***
    - causes the calling thread to terminate without the whole process terminating.
  - ***pthread_kill***
    - It sends a signal to a specified thread.
  - ***pthread_join***
    - It causes the calling thread to wait for the specified thread to exit. This is similar to waitpid for processes.
  - ***pthread_self***
    - It returns the callers identity (The thread ID).
- More details can be found by exploring the online man pages
  - eg *man pthread create*

Create a child process and child thread

```
main()
{
     pid_t childPid = fork();  // create a child process
    if (childPid == 0)
          printf("I am the child process\n");
    else    printf("I am the parent process\n");
}
```

```
main()
{
    pthread_t  childId;
    pthread_create (&childId, NULL, ChildCode, NULL); // create a child thread
    printf("I am the parent thread\n");
}
void* ChildCode (void* arg)   {   printf("I am the child thread\n");    }
```

Create a child process and child thread

After fork(), there are now two processes and both continue executing the same line of code after fork().

So fork() created a separate child process, after which the program counter points to the "if" in both processes.

After pthread_create(), there is still only one process but now with two threads, where the new thread begins executing code in a call-back function (named "ChildCode()" in this example).

So pthread_create() creates a second thread in the same process, after which the program counter points to the next statement in the main thread but the new thread's program counter points to the first line of code in the call-back function, ChildCode().

All threads within a process continue to have access to the same data. But although the child process begins with a copy of all the parent's data, neither parent nor child can access each other's data.

```
int global = 5;
void* ChildCode(void* arg) {
        int local = 10;
        global++;
        local++;
        printf("[Child]  child thread id: 0x%x\n", pthread_self());
        printf("[Child]  global: %d  local: %d\n", global, local);
}
main() {
        pthread_t  childPid;
        int   local = 10;
        printf("[At start]  global: %d  local: %d\n", global, local);
        /* create a child thread */
        if (pthread_create (&childPid, NULL, ChildCode, NULL) != 0) {
                perror("create");
                exit(1);
        } else { /* parent code */
                global++;
                local--;
                printf("[Parent] parent main thread id : 0x%x\n",  pthread_self());
                printf("[Parent] global: %d  local: %d\n", global, local);
                sleep(1);
        }
        printf("[At end] global: %d  local: %d\n", global, local);
        exit(0);
}
```

```
int global = 5;
main() {
        pid_t  childPid, status;
        int    local = 10;
        printf("[Start with] global: %d  local: %d\n", global, local);

        childPid = fork(); /* create a child process */
        if (childPid < 0) {perror("fork"); exit(1); }

        else if (childPid == 0) {     /* child code */
                global++;
                local++;
                printf("[Child]  childPid: 0x%x\n", getpid());
                printf("[Child]  global: %d  local: %d\n", global, local);
                sleep(4);     /* wait 4 seconds */
        } else {    /* parent code */
                global--;
                local--;
                /*  while (wait(&status) != childPid) ;*/
                printf("[Parent] childPid: 0x%x  parent: 0x%x\n", childPid, getpid());
                printf("[Parent] global: %d  local: %d\n", global, local);
                sleep(2);     /* wait 2 seconds */
        }
        printf("[At end (0x%x)] global: %d  local: %d\n", getpid(), global, local);
        exit(0);
}
```

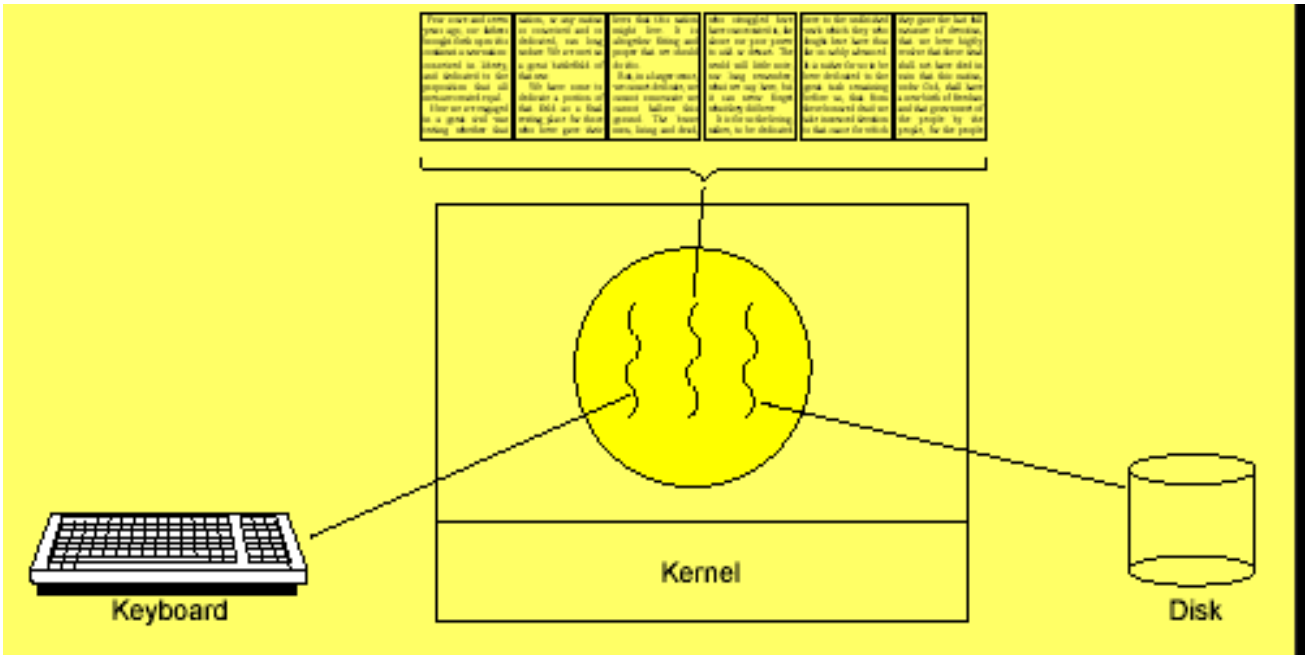
Motivation for using threads

- **Reasons for threads:**
  1. All threads within a process have access to the same data (share)
     - no "clever" communication mechanisms are needed as would be the case for processes to share data;
  2. Creating a thread is considerably easier (perhaps 100 times easier/faster) than creating a process because threads have relatively little resources attached to them
  3. Threads provide possible performance gains that can be maximized when one thread blocked for I/O can be overlapped with another thread running on the CPU.
     - Switching from one thread to another within a process is easier than switching between processes
  4. Threads are useful when there are multiple CPUs
- **Possible problems**
  - Difficult to write and debug the code
    - Typically concurrency problems are encountered

An example of thread usage

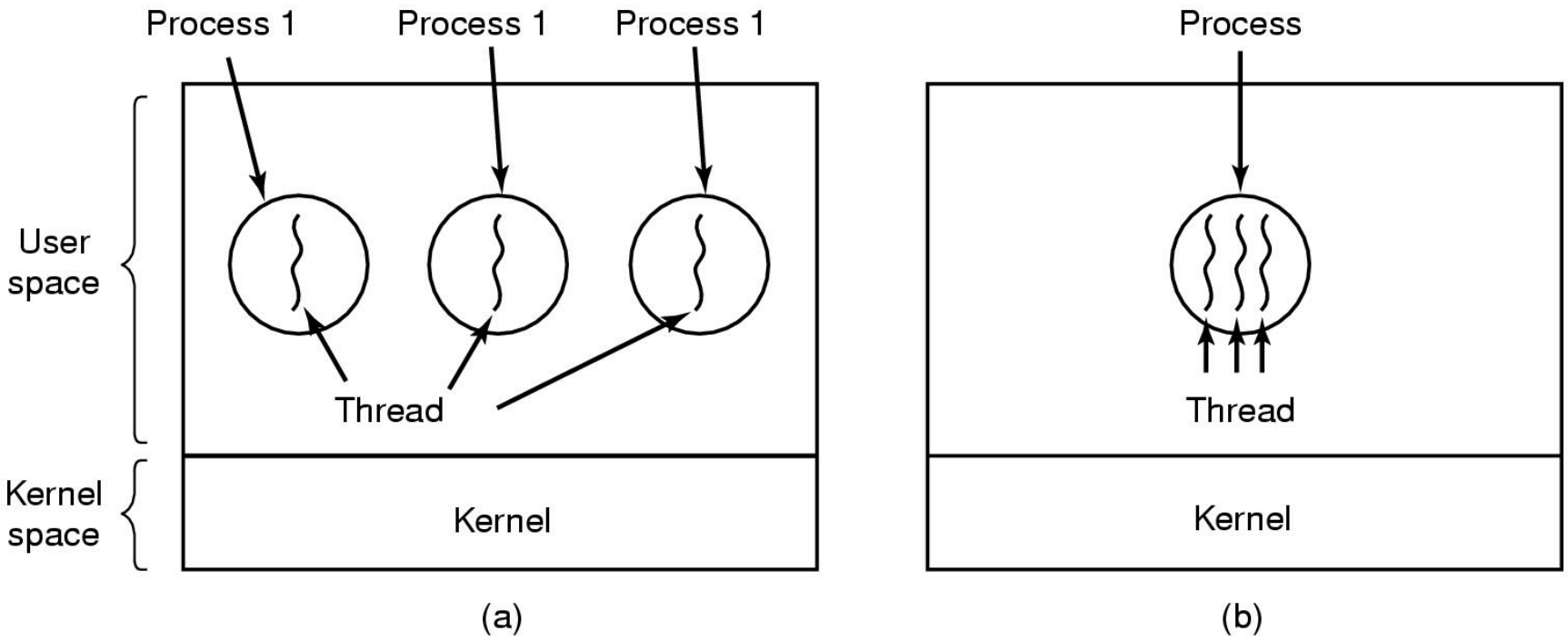
Consider a word processing application with three threads:

- one thread for interacting with the user;
- a second thread is used to reformat the content of the document being edited;
- the third automatically creates backups of the document being edited



Threads

The Thread Model (1)



(a) Three processes each with one thread

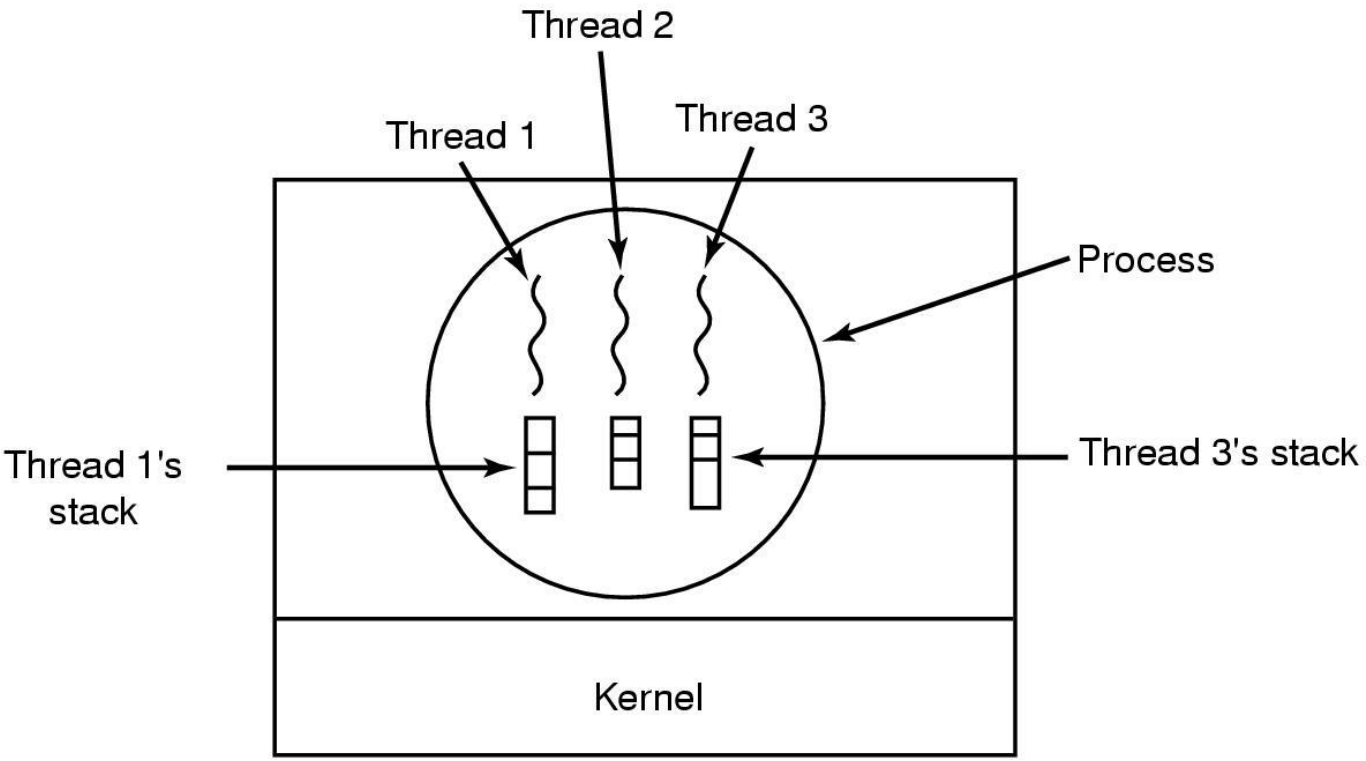
(b) One process with three threads

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	

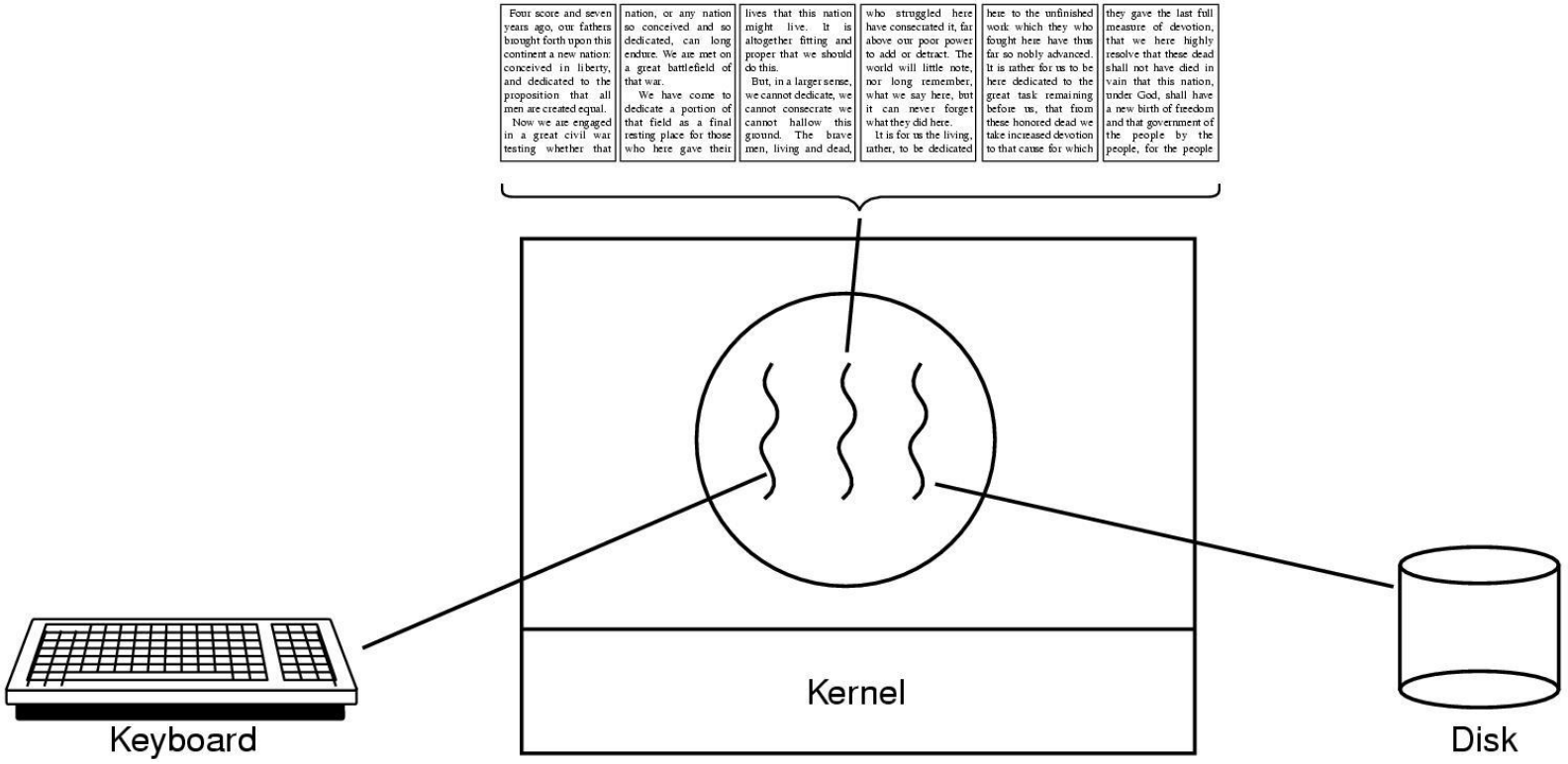
Items shared by all threads in a process **Items private to each thread**

The Thread Model (3)



Each thread has its own stack

Thread Usage (1)

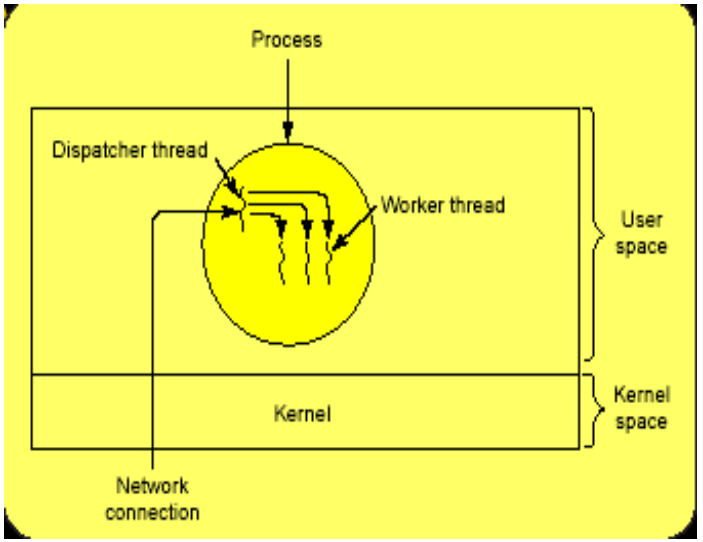


A word processor with three threads

Organizing threads within a process

Dispatcher/worker model:

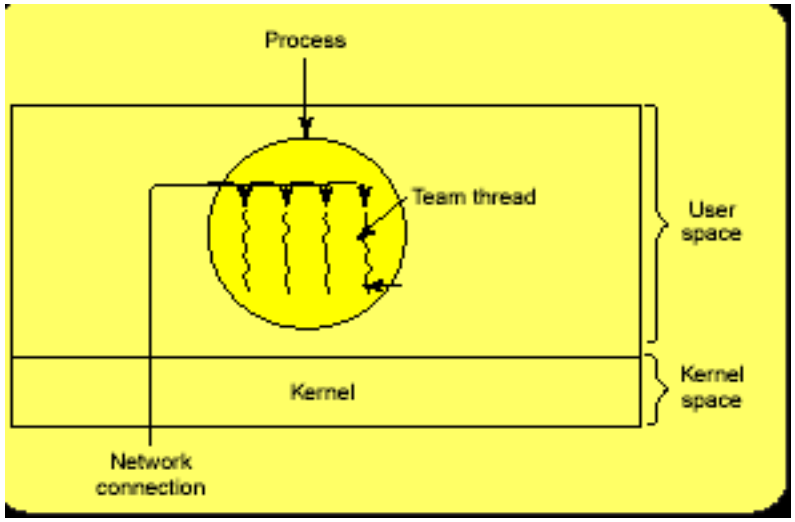
```
 while (TRUE) {
        getNextRequest(&buf );
        handoffWork(&buf );
}

 while (TRUE) {
        waitForWork(&buf );
        process(&buf );
}
```



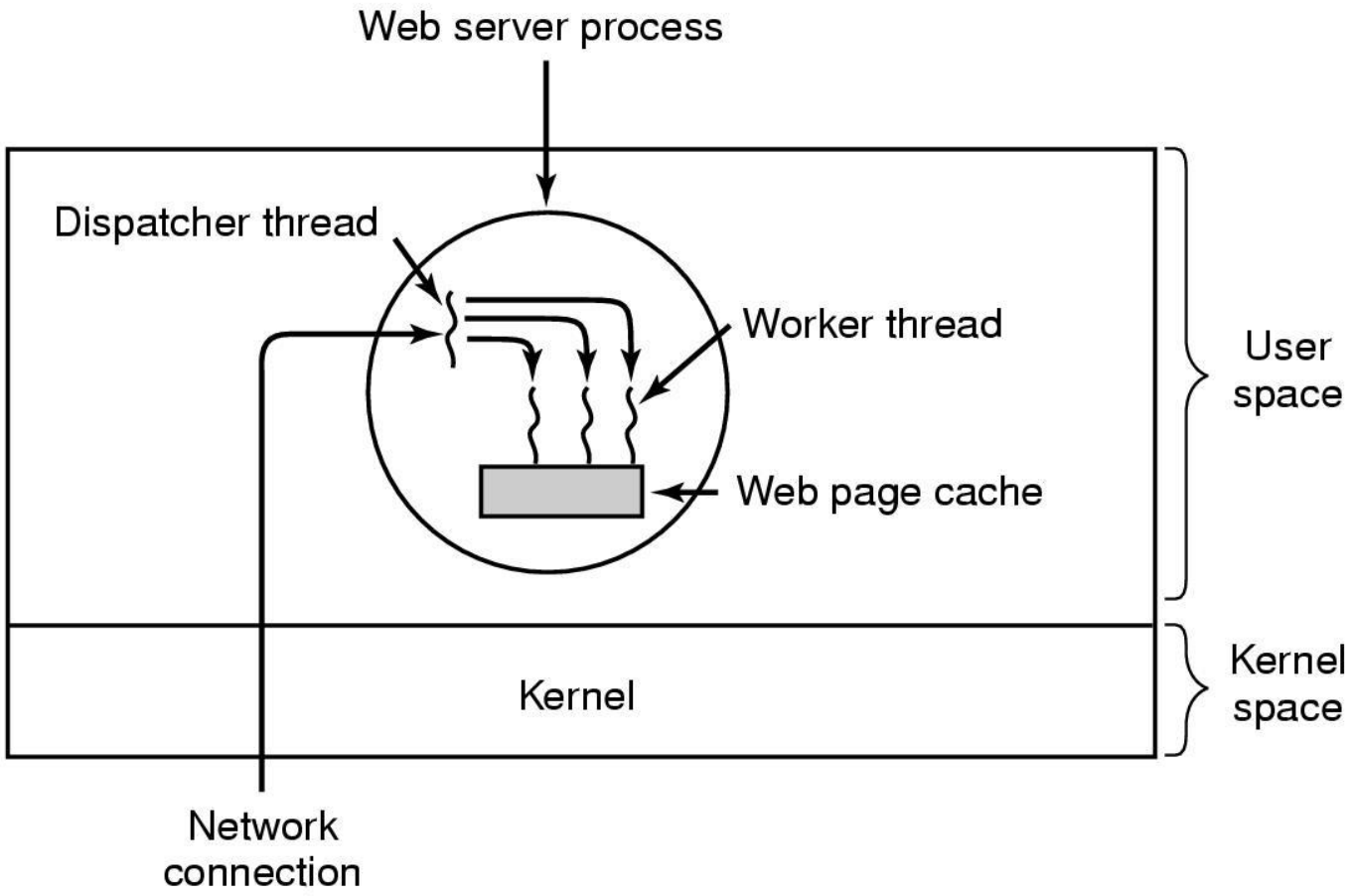
The process is multithreaded with all requests going to one, **the dispatcher thread**, while the other worker threads process the requests.
Workers may be specialized and specific requests could be dispatched to specific workers.

Team model

- Typically all team members have the same functionality in order to process any request
  - All threads are equal
  - Each thread processes incoming requests on its own
- Alternatively, **specialized threads** that place requests they cannot process onto an **internal job queue**. This queue ought to be checked by threads prior to getting new requests



Thread Usage (2)



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)

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker threads

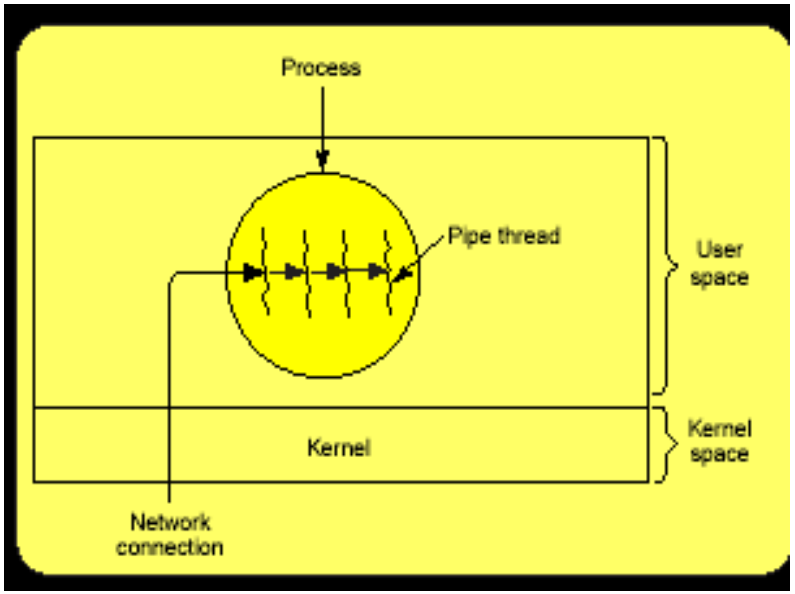
Thread Usage (4)

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

Three ways to construct a server

Pipeline

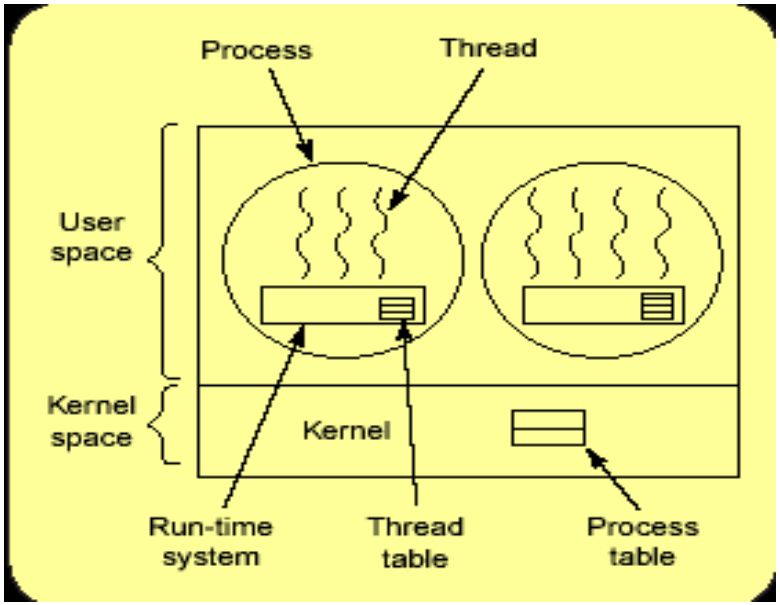
- One thread accepts new requests and may do some processing before passing the request onto the next thread.
- The next does some more processing and passes this on to the next and so forth.. . . a chain of producers and consumers.



Methods of Implementation

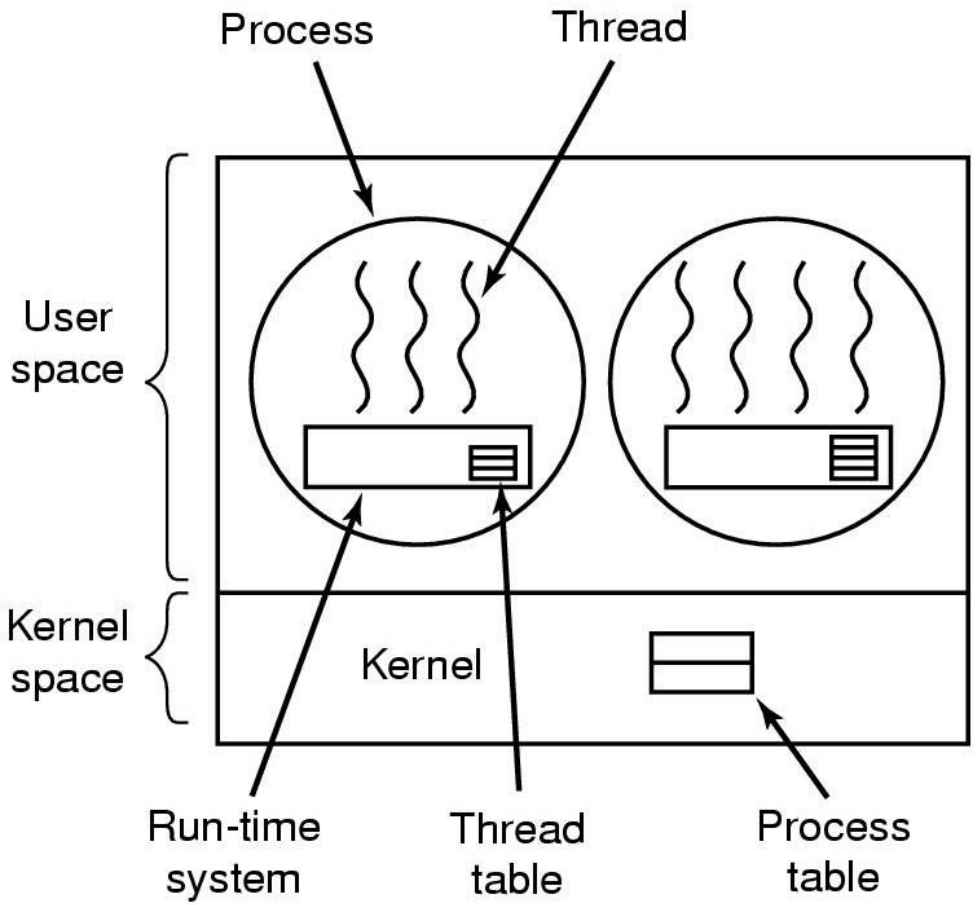
Variety of methods may be used to implement threads, each having strengths and weaknesses to be considered:

- **User level threads**
  - User-level threads are implemented in a library which provides a run time environment for threads outside of the operating system: therefore, **the kernel doesn't know about these threads.**
  - The run time environment manages the threads within a process by way of **a thread table** which is analogous to the process table maintained by the kernel.
  - Can run on any OS
  - Scheduling can be application specific



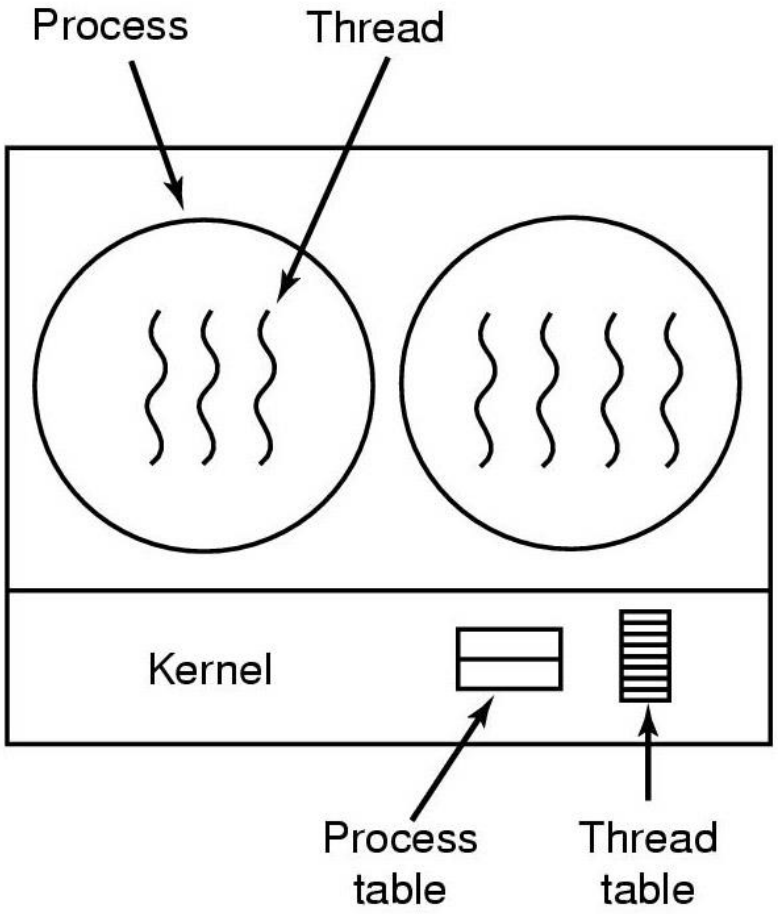
Note: when a thread makes a system call the whole process is blocked. The thread lib thinks that the thread is in running state

Implementing Threads in User Space



A user-level threads package

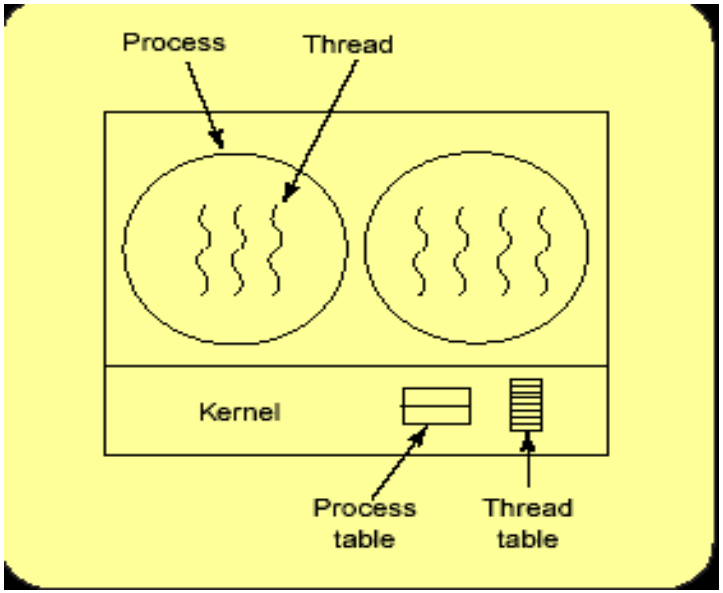
Implementing Threads in the Kernel



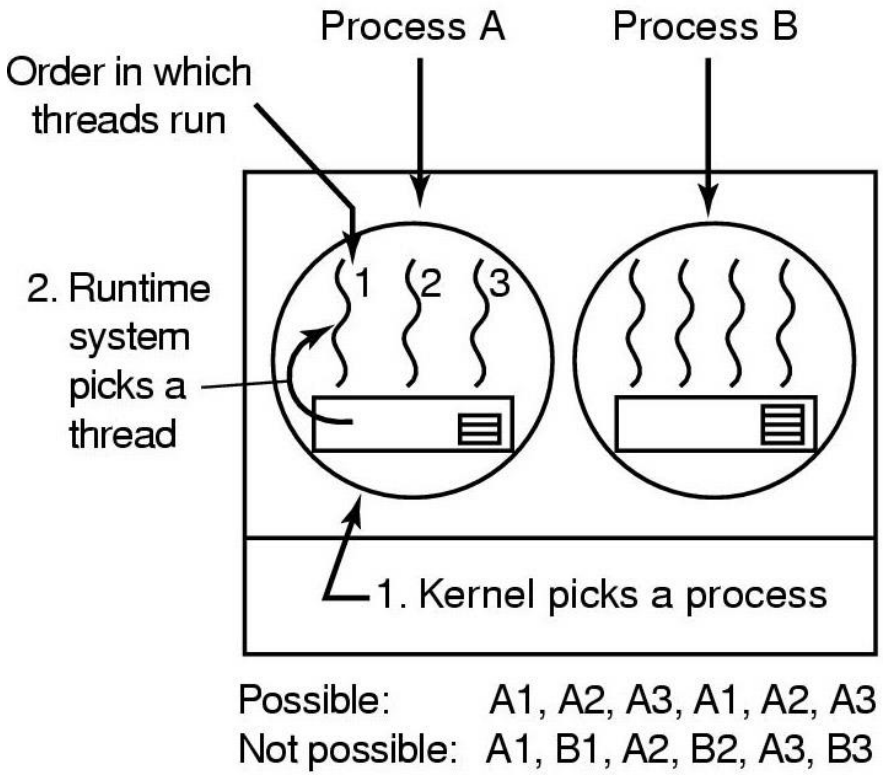
A threads package managed by the kernel

Kernel level threads

- Managed by the operating system and are,
  - in effect, first class citizens,
  - They compete for time on the CPU and for other system resources along with all other threads within all processes.
  - The **kernel** manages both **the process table** and **the thread table**, and is responsible for creating, scheduling and destroying threads.
  - The kernel does all scheduling
  - Thread switching is slower



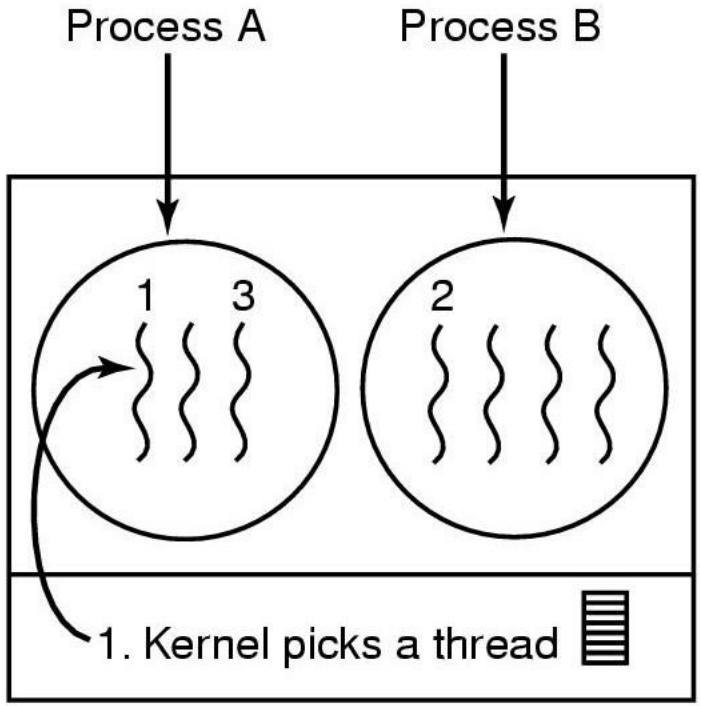
Thread Scheduling (1)



Possible scheduling of user-level threads

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

Thread Scheduling (2)



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

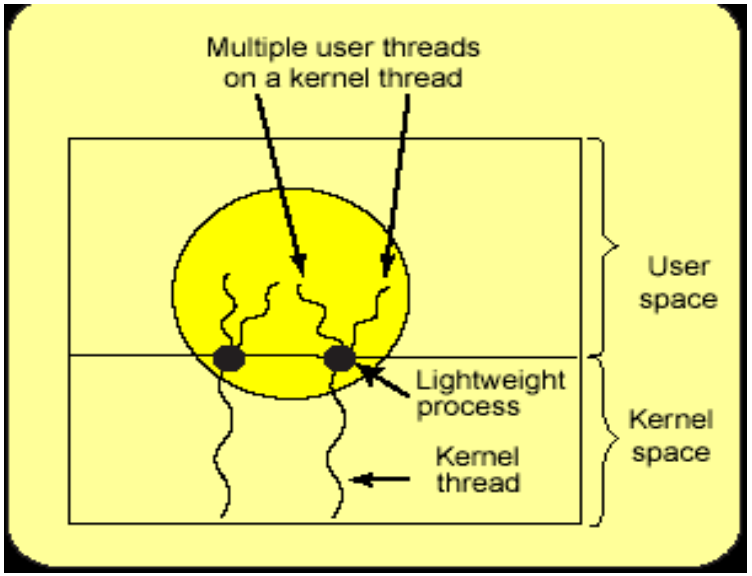
Possible scheduling of kernel-level threads

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

Hybrid implementation of threads

It involves both a run time system at the user level and kernel threads:

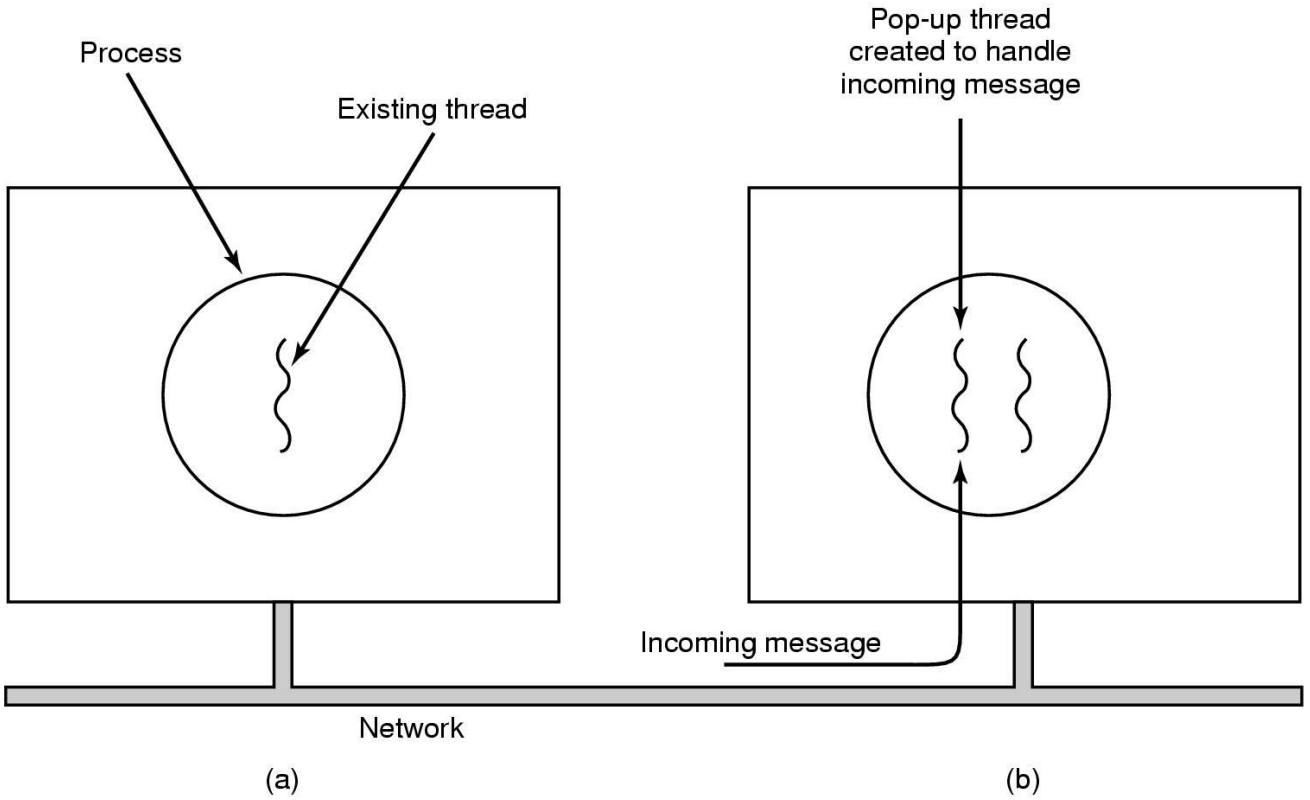
- The **user writes** the program in terms of **user-level threads but specifies how many kernel-schedulable entities are associated with the process**.
- The user-level threads are mapped to the kernel-schedulable entities. (E.g. Solaris uses a hybrid implementation of threads.)
- On Solaris, a user level thread is called a thread, and a kernel-schedulable item is called **lightweight process**



Scheduler Activations

- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  Fundamental reliance on kernel (lower layer)
  calling procedures in user space (higher layer)

Pop-Up Threads



- Creation of a new thread when message arrives
  (a) before message arrives
  (b) after message arrives

Strengths and Weaknesses of each method

User level threads:

- **Advantages:**
  - they do not require system calls
  - they do not involve kernel –
    - all management of threads occur outside the operating system in the runtime system implementing the threads.
  - more efficient requiring no context switch (save/restore of process information) and no interruption to the kernel.
  - They run on any OS
  - They may also have a customized scheduling algorithm allowing the selection of which thread is to be executed next to be determined according to the individual requirements of each process.
- **Disadvantages:**
  - Most system calls are blocking –
    - From the operating system's point of view there is only one thread that executes all user level threads and the runtime system implementing these threads.
    - Consequently, if one of these user level threads invokes a system call that blocks, all user threads are blocked.
  - A greedy thread that does no system calls or library calls won't let the thread management code schedule another thread. The programmer may have to explicitly yield control at appropriate points.

Kernel level and hybrid

- **Kernel level threads**
  - They are almost as expensive to schedule as processes but, unlike user level threads, they can be distributed across multiple processors.
  - A kernel-level thread which invokes a system call that blocks is the only thread which is blocked.
- **Hybrid implementations**:
  - They seek to gain the best of both worlds.
  - They try to minimize the number of system calls required, the amount of kernel level scheduling required, without losing the flexibility of (pseudo) parallelism.

Thread libraries

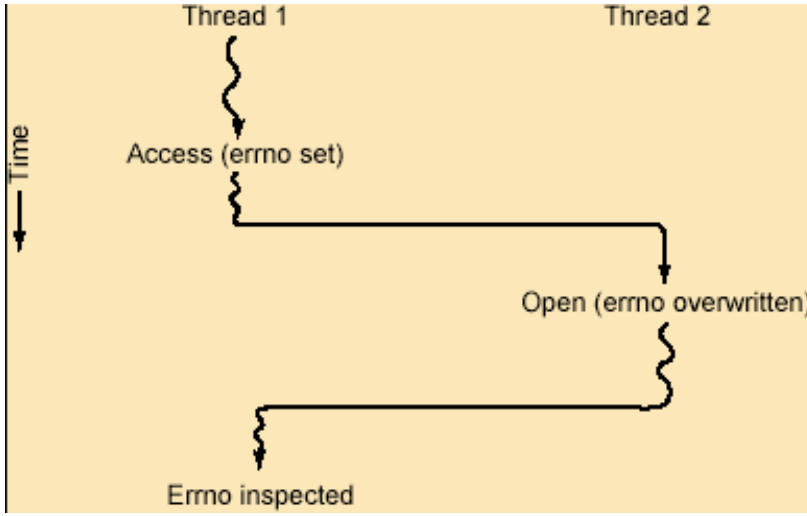
- **Linux**
  - ***libpthread***
    - ***<pthread.h>***
  - ***For linker***
    - ***specify –lpthread***
- **Some C++**
  - ***-mt option rather than -lthread***

Common problems:

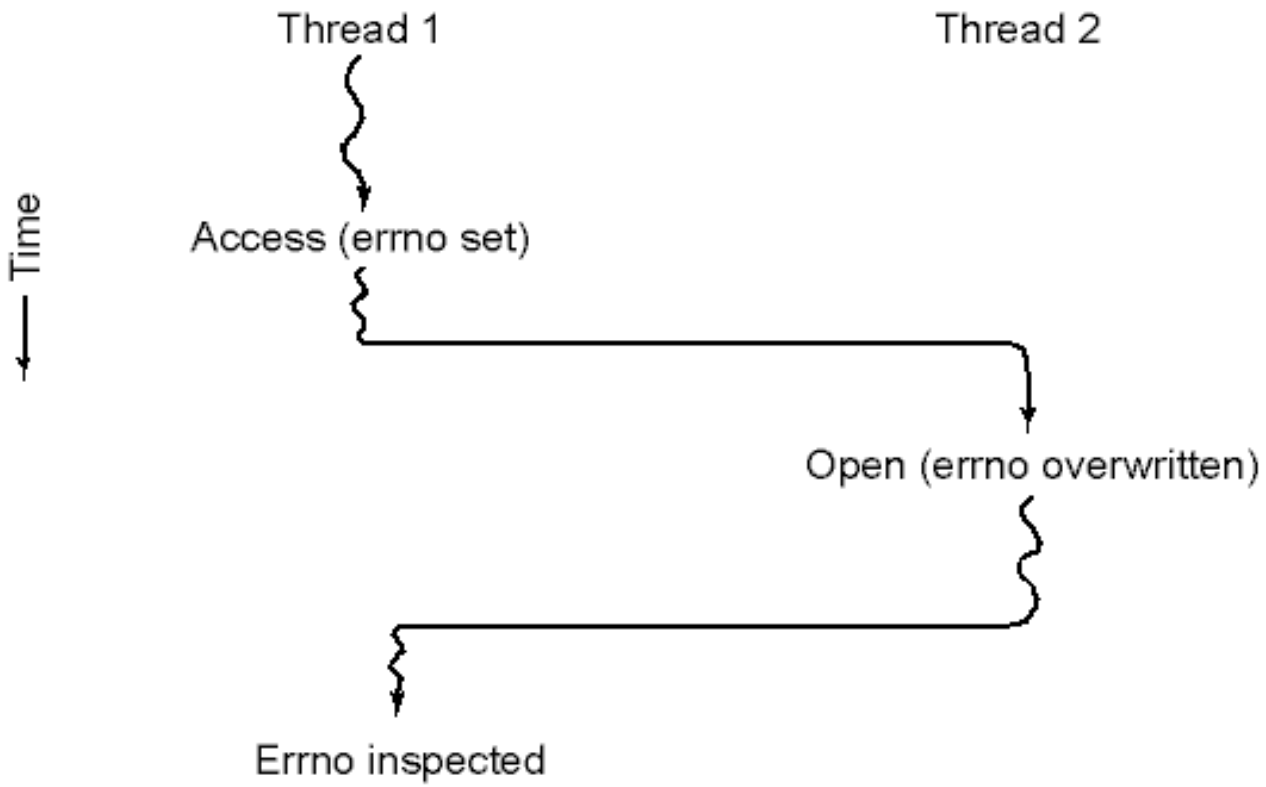
- Accessing **globals** without synchronization mechanism
- Creating **deadlocks** caused by two threads trying to acquire rights to the same global resource
- Trying to **reacquire the lock already held by a thread**
- Creating a hidden gap in synchronization
  - It happens when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result appears to the caller as the global variable is protected but it is not.
- Default threads are created as joinable (PTHREAD_CREATE_JOINABLE) and they must be reclaimed with *pthread_join()*
  - *NOTE: pthread_exit()* does not free up its storage space

Making single-threaded code multi-threaded

- There are many **legacy systems** that are single-threaded. Of these some would benefit by being restructured to use a multithreaded architecture.
- Such restructuring is often trickier than first thought for the following reasons:
  - Local variables are easy;
  - Variables that ought to be global to a single thread are more difficult.
    - One solution is for each thread to allocate memory in which to store variables global to itself.
  - Often library functions and procedures are implemented with the expectation that once called the execution of the procedure or function will be completed before being called by some other process or thread.
  - That is, they are not **reentrant code**
- Of course, there are other issues such as signal handling, stack management, and so on.

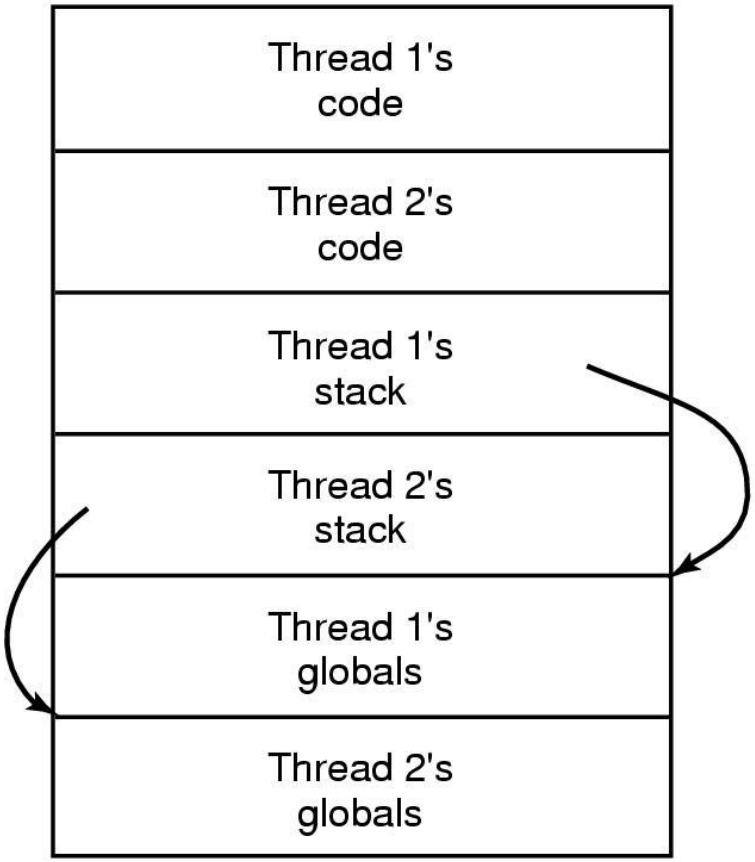


Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable

Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables

Some thread functions

int ***pthread_join(pthread_t tid, void **status);***

This function blocks the calling thread until the specified thread terminates. If status is not NULL, it points to the location that is set to the exit status of the terminated thread when the function successfully returns.

Multiple threads cannot wait for the same thread to terminate.

Some examples

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

void print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  thread_return1, thread_return2;

   /* Create two independent threads each of which will execute  a function
      and print a message*/

    thread_return1 = pthread_create( &thread1, NULL,
           (void*)&print_message_function, (void*) message1);
    thread_return2 = pthread_create( &thread2, NULL,
           (void*)&print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}
```

Thread with 'detach state'

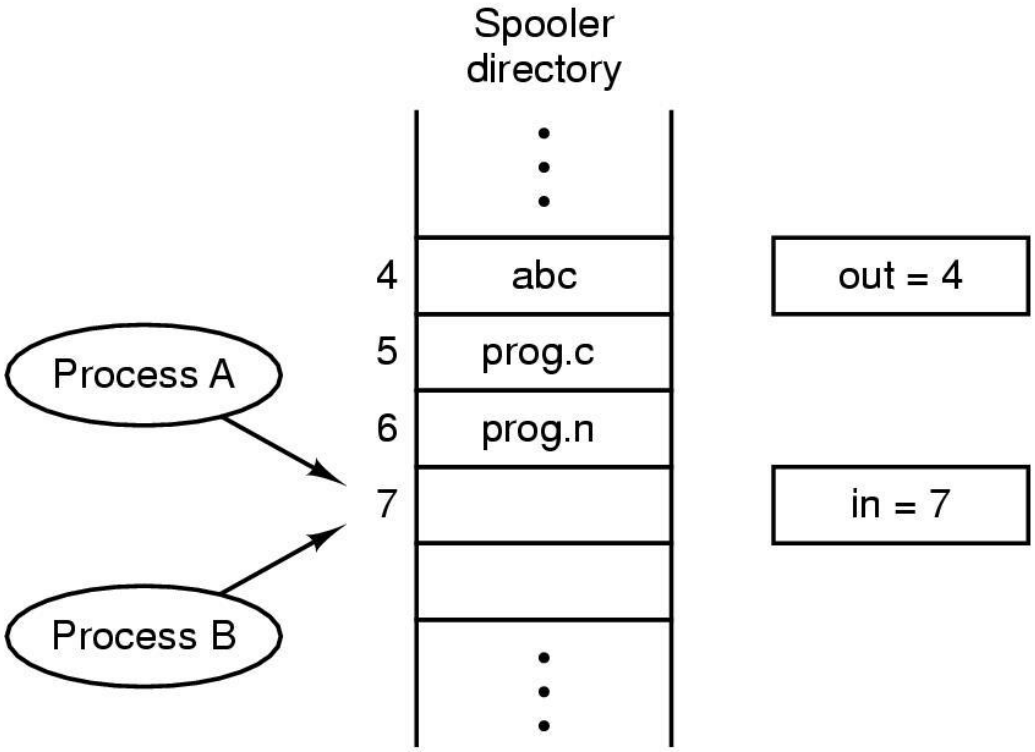
```
void main()
{
  pthread_attr_t       pta;
  int status1, status2;

  pthread_t main_thr;
  rc = pthread_attr_init(&pta);
  rc2 = pthread_attr_setdetachstate(&pta,  PTHREAD_CREATE_DETACHED );

  /* When thread is created detached (PTHREAD_CREATE_DETACHED),
       its thread id and other resources can be reused as soon as the thread terminates */

  rc = pthread_create(&thread, &pta,
      (void*)&print_message_function, (void*) message2);
  rc = pthread_create(&thread1, NULL,
       (void*)&print_message_function, (void*) message1);

  pthread_join(thread1, &status1);
  pthread_join(thread, &status2);

  printf("Joining threads 1 and thread %d  %d\n", status1, status2);

  /*function destroy is used to remove the storage allocated
     during the initialization*/
  rc = pthread_attr_destroy(&pta);
  sleep(5);
  main_thr = pthread_self();
  printf("Main thread = %d\n", main_thr);
}
```

Example of output:

I am non-default parms thread
I am default parms thread 1
Joining threads thread1 and thread 29 0
Main thread = 8192*/

Interprocess Communication

Race Conditions



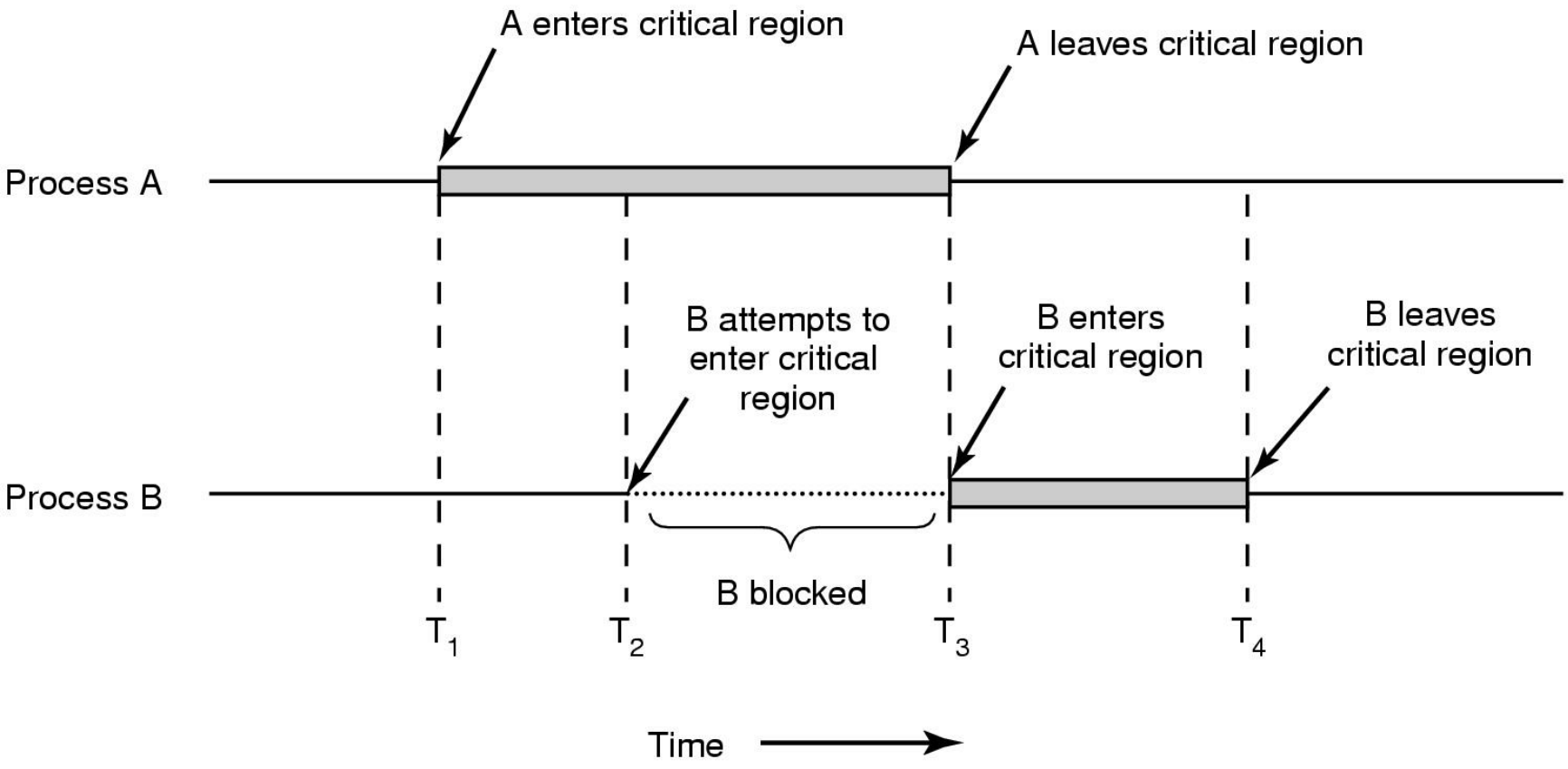
Two processes want to access shared memory at same time

Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions (2)



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting

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

(a)

Process 0

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

(b)

Process 1

Proposed solution to critical region problem

(Note: lock using busy waiting is called a spin lock)

Semaphore function names

Various books/etc use different function names to mean the same semaphore action:

For example:

semaphore		mutex
`up()=sem_post()=signal()`	=	`pthread_mutex_unlock()`
`down()=sem_wait()=wait()`	=	`pthread_mutex_lock()`

Note:

up/post/signal/unlock => **+1**

down/wait/lock => **-1** until 0: **suspend/block on 0**

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

mutex for mutual exclusion, empty/full for synchronisation

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

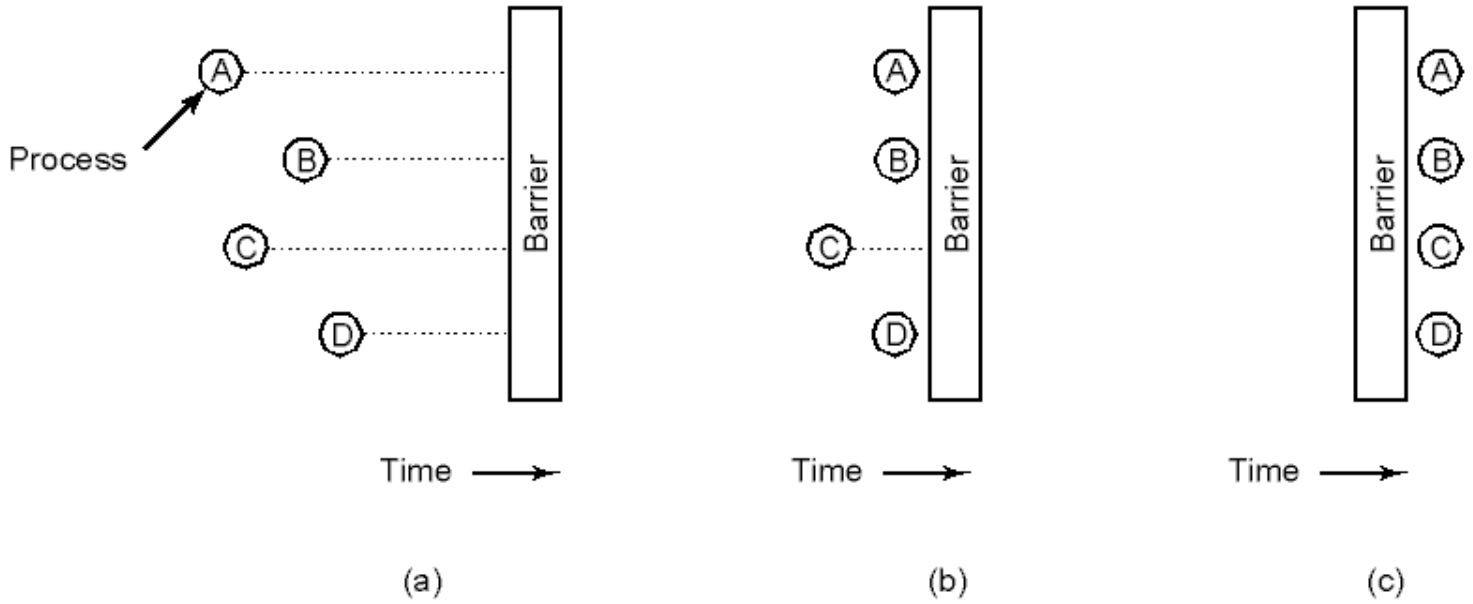
- only one monitor procedure active at one time
- buffer has *N* slots

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 (using no shared memory)

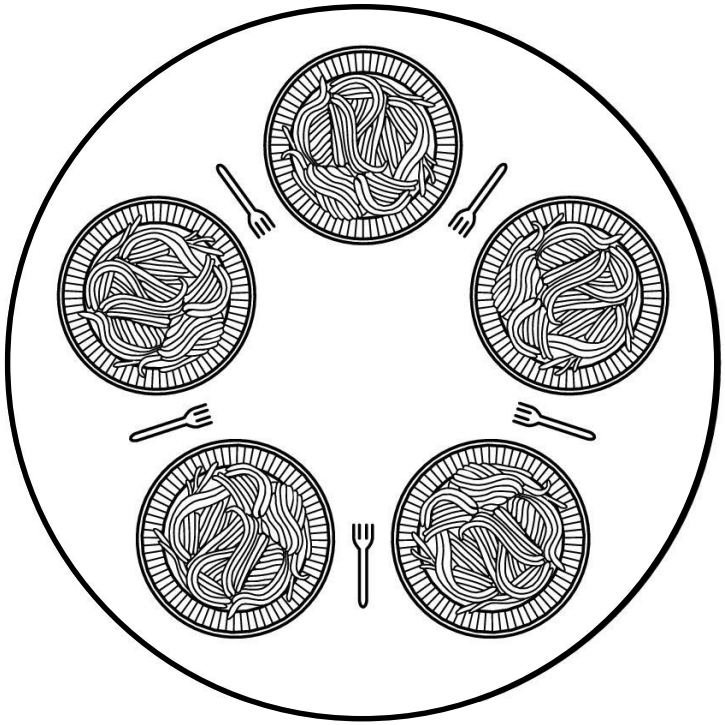
Barriers



- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

Dining Philosophers (1)

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



Dining Philosophers (1)

Five philosophers sitting at a circular table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating.

In order to eat, a philosopher must be in possession of both forks. A philosopher may only pick up one fork at a time. Each philosopher attempts to pick up the left fork first and then the right fork. When done eating, a philosopher puts both forks back down on the table and begins thinking. Since the philosophers are sharing forks, it is not possible for all of them to be eating at the same time.

Demo: www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html

Dining Philosophers (1)

The **dining philosophers problem** is an illustrative example of a common computing problem in concurrency. It is a classic multi-thread synchronization problem of **deadlock** and **resource starvation**.

The philosophers never speak to each other, which creates a dangerous possibility of **deadlock** when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa).

Starvation (and the pun was intended in the original problem description) might also occur independently of deadlock if a philosopher is unable to acquire both forks due to a timing issue.

In 1971, Edsger Dijkstra set an examination question on a synchronization problem where five computers competed for access to five shared peripherals. Soon afterwards the problem was retold by Tony Hoare as the dining philosophers problem.

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 nonsolution to the dining philosophers problem

Dining Philosophers Solution (3)

In this problem of multiple threads accessing multiple resources, **the solution is** a **semaphore per philosopher** and **one mutex**.

To avoid deadlock, a semaphore is used for each philosopher's acquisition of both forks – i.e. to keep track of which threads are currently using resources (forks).

To avoid starvation one mutex protects the critical region of:

- **being hungry** and attempting to acquire both forks
- **wanting to think** and releasing both forks

Dining Philosophers (4)

```
#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 (5)

```
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 */
}

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 2)

The Readers and Writers Problem

```
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)
{
      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

The Sleeping Barber Problem (1)

The Sleeping Barber Problem (1)

The sleeping barber problem is a classic **inter-process communication** and **synchronization** problem between multiple threads. The problem is analogous to that of keeping barbers working when there are customers, resting when there are none and doing so in an orderly manner. The barbers and their customers represent threads.

A barber shop with barbers, barber chairs, and a number of chairs for waiting customers. When there are no customers, barbers sit in their chairs and sleep. As soon as a customer arrives, he either awakens a barber or, if all barbers are cutting hair, sits down in one of the vacant chairs. If all of the chairs are occupied, the newly arrived customer simply leaves.

The Sleeping Barber Problem is often attributed to Edsger Dijkstra (1965), one of the pioneers in fundamental programming.

The Sleeping Barber Problem (1)

Semaphores & mutex: The most common solution involves using three semaphores: one for (waiting) customers, one for (idle) barbers and a mutex.

Customer: When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded. The customer then checks to see if there is an empty chair for him (either one in the waiting room or a barber chair), and if none of these are empty, leaves. Otherwise the customer takes a seat – thus reducing the number available (a critical section). The customer then signals a barber to awaken through his semaphore, and the mutex is released to allow other customers (or a barber) the ability to acquire it. If a barber is not free, the customer then waits.

Barbers: The barbers sit in a perpetual waiting loop, being awakened by any waiting customers. Once awoken, a barber signals the waiting customers through their semaphore, allowing them to get their hair cut.

The Sleeping Barber Problem (1)

Problem: program the barber and the customers without getting into race conditions.:

- Solution uses three semaphores:
  - *customers*; counts the waiting customers
  - *barbers*; the number of idle barbers
  - *mutex*; used for mutual exclusion
  - also need a variable *waiting* to count the waiting customers
    because it is not possible to read the value of a semaphore
- The barber calls the procedure ***barber***,
  causing him to block on the semaphore *customers* (initially 0)
- The barber then goes to sleep
- When a customer arrives, he executes ***customer***,
  starting by acquiring *mutex* to enter a critical region
- if another customer enters, shortly thereafter, the second one will not be able to do anything
  until the first one has released *mutex*
- The customer then checks to see if the number of waiting customers
  is less than the number of chairs
- if not, he releases *mutex* and leaves without a haircut
- if there is an available chair, the customer increments the integer variable, *waiting*
- Then he does an up on the semaphore *customers*
- When the customer releases *mutex*, the barber begins the haircut

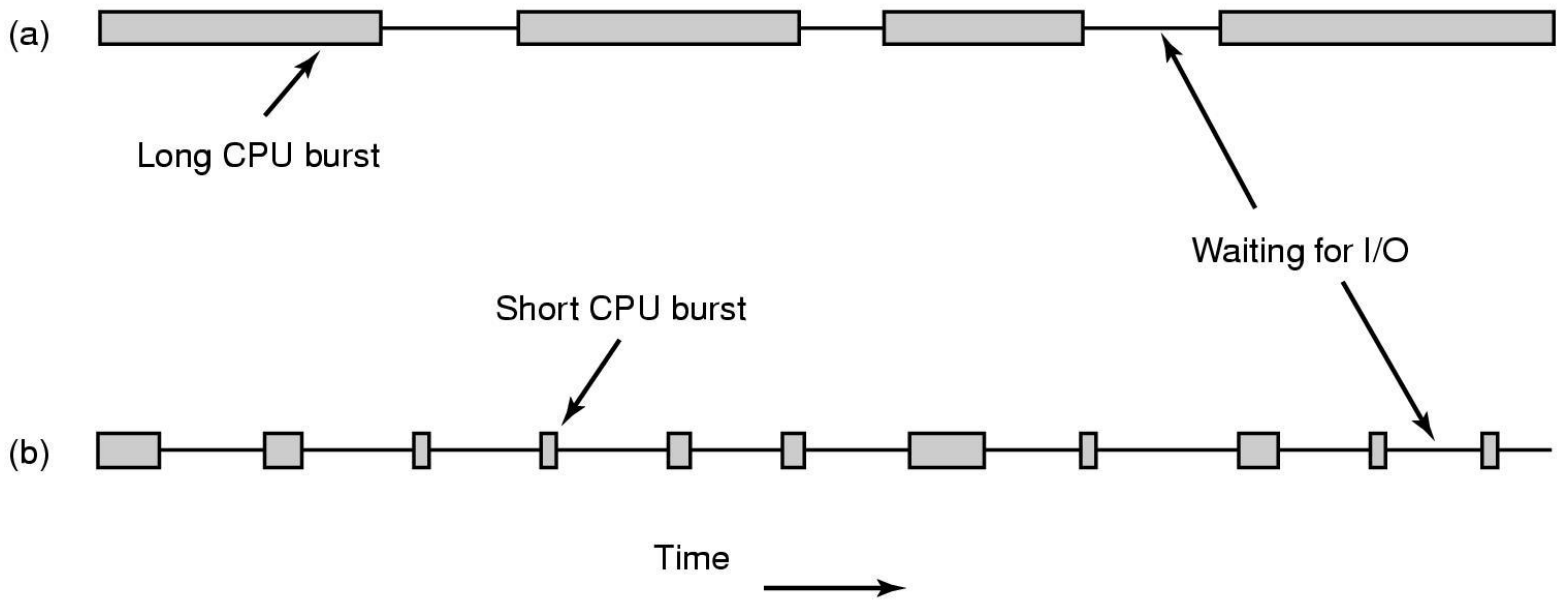
The Sleeping Barber Problem (2)

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting - 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Solution to sleeping barber problem.

Scheduling

Scheduling
Introduction to Scheduling (1)



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

Introduction to Scheduling (2)

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

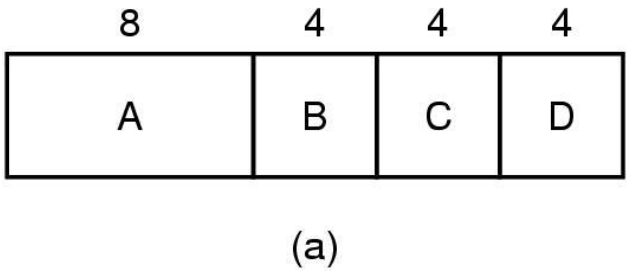
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

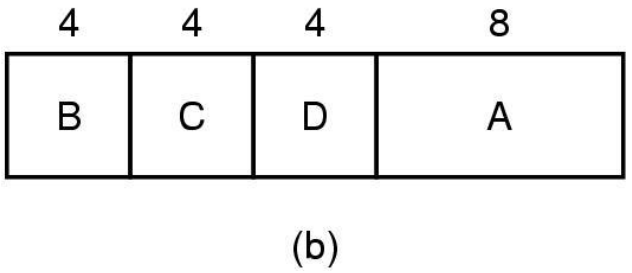
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

Scheduling in Batch Systems (1)



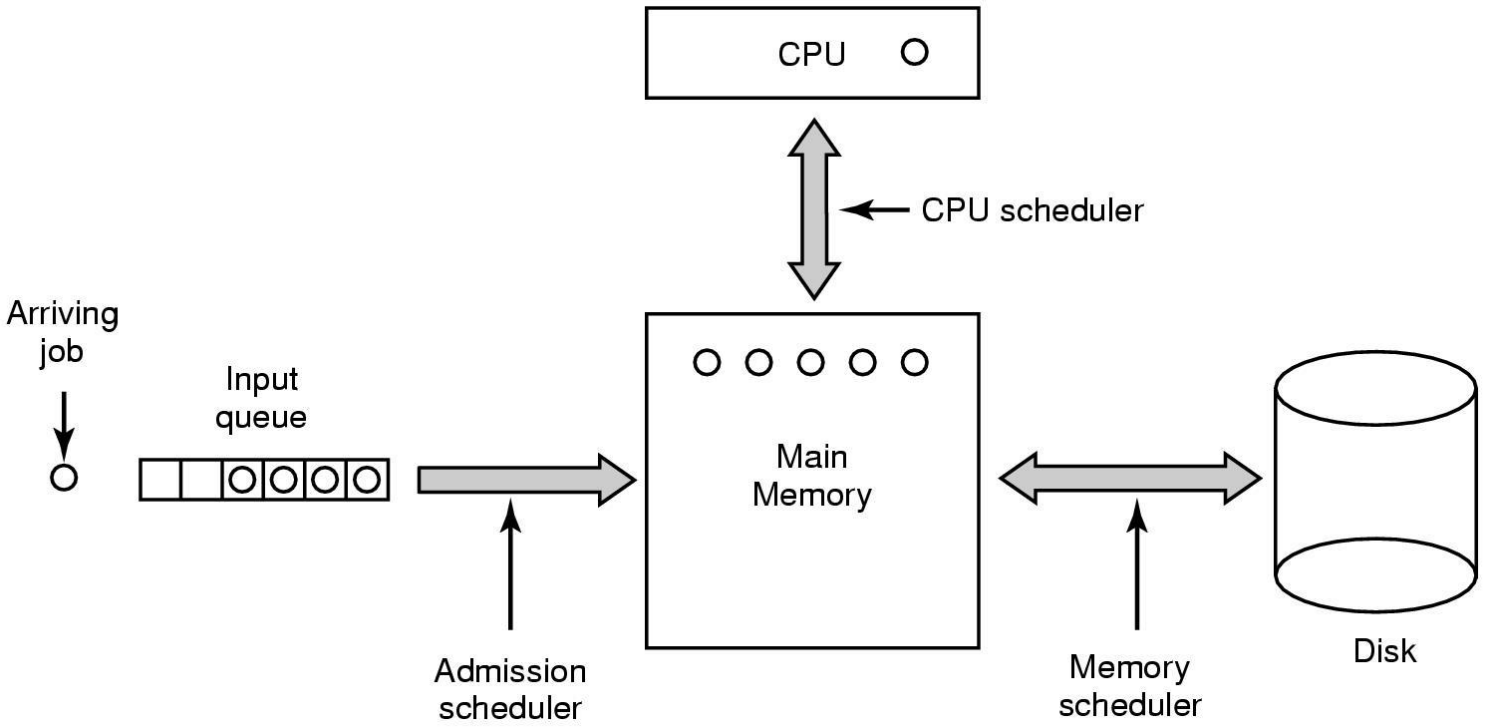
(a)



(b)

An example of shortest job first scheduling

Scheduling in Batch Systems (2)

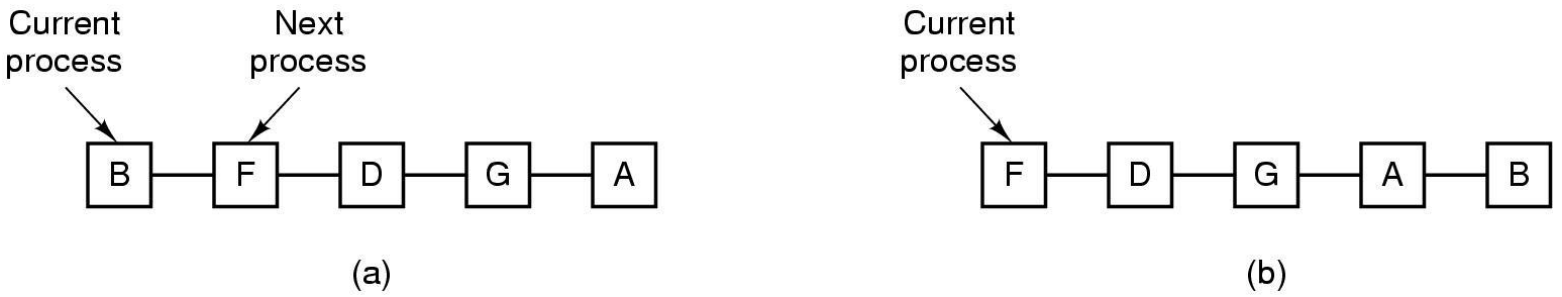


Three level scheduling

Scheduling in Interactive Systems (0)

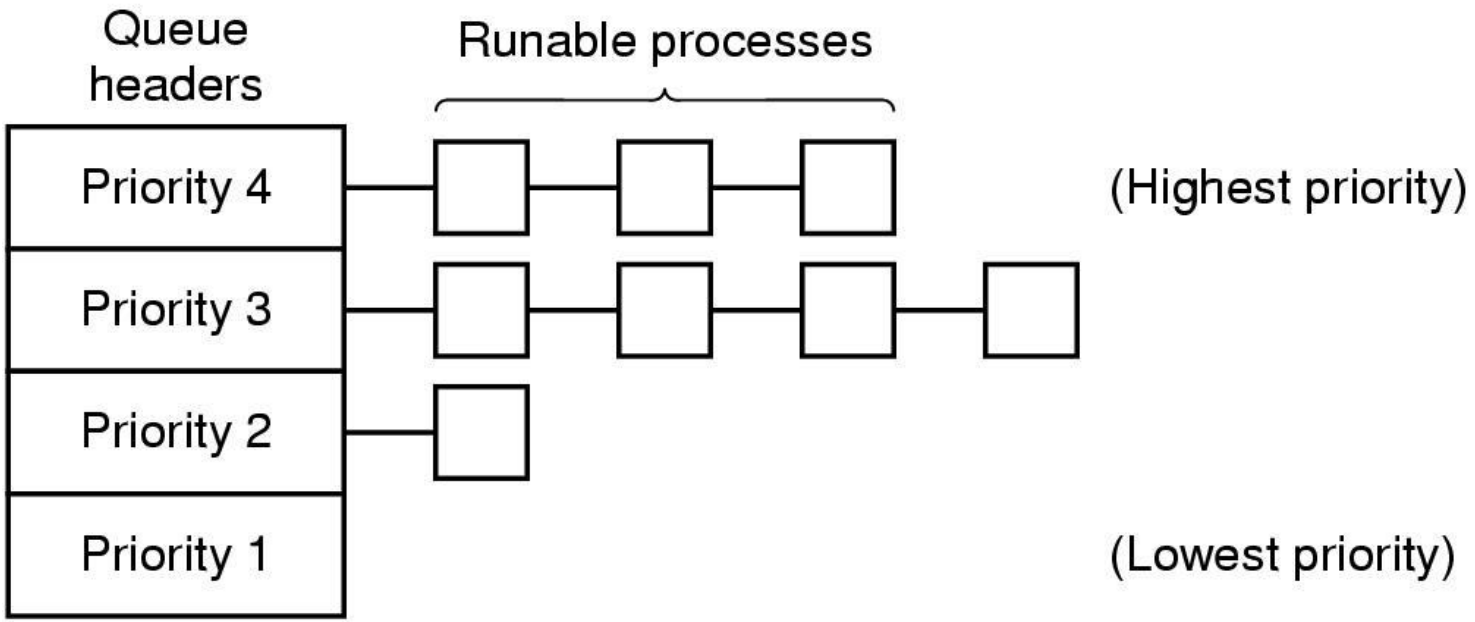
- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

Scheduling in Interactive Systems (1)



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

Scheduling in Interactive Systems (2)



A scheduling algorithm with four priority classes

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

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

Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process

Sample Exam Question

Operating systems support data structures (tables) for processes, threads, sockets and files/pipes. Each data structure contains a list of the allocated resource items.

Of the resources listed below,

(a) Which are allocated to each thread?

(b) Which of the remaining resources are allocated to each process?

Accounting information, Address space, Child processes, Children's CPU time, CPU time used, File descriptors, Global variables, Group ID, Parent process, Pending alarms, Pointer to data segment, Pointer to stack segment, Pointer to text segment, Priority, Process group, Process ID, Program counter, Registers, Root directory, Scheduling parameters, Signals and signal handlers, Stack pointer, State , Time of next alarm, Time when process started, User ID, Working directory

Items Per Thread

Program counter

Registers

Stack pointer

State

Items Per Process

All the remaining items

Sample Exam Question

Describe the differences between creating a thread and creating a process using the code below in your discussion.

```
void main (void)
{
    pid_t childId = fork();
   if (childId == 0)
            printf("I am Tweedledee\n");
   else     printf("I am Tweedledum\n");
}
```

```
void* myFunction (void* arg)   {   printf("I am Tweedledum\n");    }

void main (void)
{
   pthread_t  childId;
   pthread_create (&childId, NULL, myFunction, NULL);
   printf("I am Tweedledee\n");
}
```

Sample Exam Answer

(About 24 separate facts follow for full marks.)

After *fork()*, there are now **two processes** with **one thread each**, both continue **executing the same line of code** after *fork()*.

Whereas after *pthread_create()* (which has a **parameter which is a pointer to a function**), there is still only **one process** but now with **two threads**, where the **new thread begins executing code in a call-back** function (i.e. program counter pointing to first line in this function, not continuing from *pthread_create*).

Specifically, *fork()* creates a **separate child process**, after which the program counter points to the **"*if*" in both processes**. Processes are independent so that if one process ends the other continues. But if the main thread ends, all other threads may terminate.

Both threads share the **same global** address space but since each thread has its own stack, changing a local variable will not affect other threads. Whereas the child process has its **own address space** initialised with the **parents data** at the time of the *fork()* (except the *childId* value which returns **0 to the child** and the **child PID to the parent**). Unlike threads, processes cannot directly modify each other's data without using "clever" communication mechanisms such as files, pipes or sockets.

Sample Exam Answer

Threads can communicate using globals, but **processes need pipes, sockets, files, etc** to communicate.

No guaranteed order of execution in either example.

Also there is **no guarantee that the child thread will print if the parent thread terminates first**. **But the child process will print even if the parent process terminates first**.

Compared to multiple processes (with one thread), multiple threads in a process are **more difficult to debug** because they can change the same globals or deadlock using semaphores.

Switching execution between threads is much faster than context switching processes and creating a thread is considerably easier (up to 100 times **easier/faster) than creating a process** because threads have relatively **little resources** attached to them being, **Program counter, Registers, Stack pointer, State.**

Sample Exam Question

Briefly describe the following three multi-threaded application models:

(a) Team model, (b) Pipeline model, (c) Dispatcher/worker model

(a) Team model

Typically all team threads have the same functionality in order to process any request, where

* All threads are equal
* Each thread processes incoming requests on its own

(Team model is used in some active network toolkits such as ANTS and JANOS)

(b) Pipeline model

One thread accepts new requests and may do some processing before passing the request onto the next thread. The next does some more processing and passes this on to the next and so forth creating a chain of producers and consumers

(c) Dispatcher/worker model

The process is multithreaded with all requests going to one, the dispatcher thread, while the other worker threads process the requests. Workers may be specialized and specific requests could be dispatched to specific workers

Sample Exam Question

Describe the difference between a mutex and semaphore.

Mutexes and semaphores are both used in multiple thread/process applications as synchronisation primitives which block if they have a value of 0, and are not blocked if they have a higher value **- but with the following differences:** (next slide)

Mutex	Semaphore
two states (binary semaphore)	more than two states (counting semaphore)
only the thread that locked or acquired the mutex can unlock it	a thread waiting on a semaphore can be signalled by a different thread
Used to ensure exclusive access to a resource. Protect small critical regions of code – i.e. held for the shortest time possible. Ensures serialisation to non re-entrant code.	Used to restrict the number of simultaneous users of a shared resource up to a maximum number (to synchronise resources such as adding/removing items in a queue)
Locking mechanism: e.g. one thread at a time can use a shared resource	Signalling mechanism: e.g. "I am done so now you can continue"
only a resource locking mechanism (mutual exclusion)	may be used for both resource locking and event notifications
pthread_mutex_init() pthread_mutex_lock() pthread_mutex_unlock() pthread C lib uses different functions	sem_init() sem_wait() sem_post()

Sample Exam Question

Discuss the following multi-tasking problems and solutions, if any.

a) Busy waiting

b) Starvation

c) Race condition

Sample Exam Answer

a) Busy waiting (or spinning) is a technique in which a process repeatedly checks to see if a condition is true, such as waiting for keyboard input or waiting for a lock to become available or delay execution for some amount of time. Busy waiting should be avoided, as the CPU time spent waiting could have been reassigned to another task. Most operating systems and threading libraries provide semaphores and delay (sleep) functions to be used instead.

b) Starvation is a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task. Two or more programs become deadlocked together, when each of them wait for a resource occupied by another program in the same set. On the other hand, one or more programs are in starvation, when each of them is waiting for resources that are occupied by programs, that may or may not be in the same set that is starving. This can be avoided through using semaphores to ensure that as a process begins acquiring a resource, no other process will.

c) Race conditions arise in software when separate processes or threads of execution depend on some shared state whereby the result is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first. One possible solution to this problem, is to add synchronization.

Sample Exam Question

In 1971, Edsger Dijkstra set an examination question on a synchronization problem where five computers competed for access to five shared tape drive peripherals. Soon afterwards the problem was retold by Tony Hoare as the “dining philosophers problem”.

a) Name and briefly describe the two common multi-thread synchronization computing problems in concurrency that the dining philosophers problem is illustrating. [2 marks]

b) Describe how these two problems are solved in terms of semaphores and mutexes. [2 marks]

Sample Exam Answer

a) A classic multi-thread synchronization problem of **deadlock** and **resource starvation**. The philosophers never speak to each other, which creates a dangerous possibility of **deadlock when all threads are waiting for resources which cannot be obtained** - i.e every philosopher holds a left fork and waits perpetually for a right fork (or vice versa). **Starvation when some threads do not get access to any resources** (might also occur independently of deadlock) – i.e. if a philosopher is unable to acquire both forks due to a timing issue.

b) In this problem of multiple threads accessing multiple resources, **the solution is a semaphore per philosopher and one mutex**.

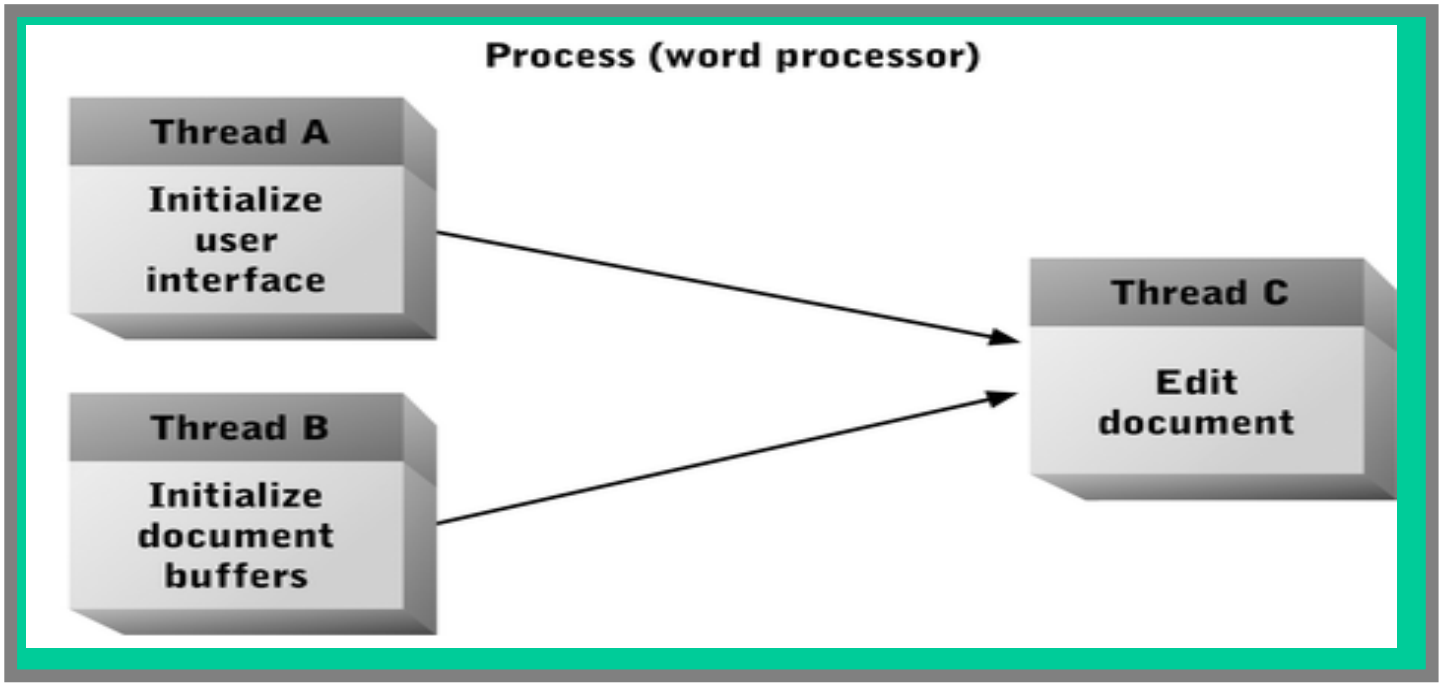
To avoid deadlock, a semaphore is used for each philosopher's acquisition of both forks – i.e. to keep track of which threads are currently using resources=forks.

To avoid starvation one mutex protects the critical region of:

- **being hungry** and attempting to acquire both forks
- **wanting to think** and releasing both forks

Interprocess Communication using Threads

- Signals are sent between cooperating processes to synchronize them
- Signals can be multivalued



Interprocess Communication Using Pipes

- Performed by writing to and reading from memory shared by processes
- Shared memory can be thought of as a pipe that supports the flow of data from one process to another process

Interprocess Communication Using Sockets