Processes & Threads

Process Concept (1)

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
  - One of the most important abstractions managed by an OS

Process Concept (2)

- A process includes:
  - stack
  - data section
  - Code (text) section
  - program counter

The Process Model (1)

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

- Four program counters
  - A
  - B
  - C
  - D

(b) Conceptual model of four independent, sequential processes.

The Process Model (3)

- D
- C
- B
- A

(c) Only one program is active at any instant.

Process State

- As a process executes, it changes state:
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution

Diagram of Process State

States diagram for a process. The label at an edge describes a possible event that causes the transition.
Process Control Block (PCB) (1)

- The system manages a data structure for each process: the Process Control Block
- It holds at least the following information about the process:
  - Process state - what state is the process (running, ready, …)
  - Process id number - used by the OS to identify the particular process
  - Program counter - where it goes in its code
  - CPU registers - contents of each register

Process Control Block (PCB) (2)

- CPU scheduling information - process priority, pointer to scheduling queues, etc
- Memory-management information - value of base and limit registers, page or segment tables, etc.
- Accounting information - CPU time used, real time used, time limits, etc.
- I/O status information - list of I/O devices allocated to the process, open files, etc.

The PCB in Linux

The PCB in Linux is represented by a C structure: task_struct

- Some of its fields are:
  ```c
  pid_t pid;  /* process identifier */
  long state;  /* state of the process */
  unsigned int time_slice;  /* scheduling information */
  struct task_struct *parent;  /* this process's parent */
  struct list_head children;  /* list of children */
  struct files_struct *files;  /* list of open files */
  struct mm_struct *mm;  /* address space of this process */
  ```

As an illustration of how the kernel might manipulate one of the fields in the task struct for a specified process, let's assume the system would like to change the state of the process currently running to the value new_state. If current is a pointer to the process currently executing, its state is changed with the following:

```
current -> state = new_state;
```
Implementation of Processes

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>UMASK mask</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to bss segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Exit status</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Process state</td>
<td>Signal status</td>
<td>Effective uid</td>
</tr>
<tr>
<td>Time when process started</td>
<td>Process id</td>
<td>Effective gid</td>
</tr>
<tr>
<td>CPU time used</td>
<td>Parent process</td>
<td>System call parameters</td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td>Process group</td>
<td>Various flag bits</td>
</tr>
<tr>
<td>Time of next alarm</td>
<td>Real uid</td>
<td></td>
</tr>
<tr>
<td>Message queue pointers</td>
<td>Effective uid</td>
<td></td>
</tr>
<tr>
<td>Pending signal bits</td>
<td>Real gid</td>
<td></td>
</tr>
<tr>
<td>Process id</td>
<td>Effective gid</td>
<td></td>
</tr>
<tr>
<td>Various flag bits</td>
<td>Bit maps for signals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Various flag bits</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-4. Some of the fields of the MINIX 3 PCB. The fields are distributed over the kernel, the process manager, and the file system.

Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.
- Context of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Process Creation

- Principal events that cause processes to be created:
  1. System initialization.
  2. Execution of a process creation system call by a running process.
  3. A user request to create a new process.
  4. Initiation of a batch job.

Process Termination

- Conditions that cause a process to terminate:
  1. Normal exit (voluntary).
  2. Error exit (voluntary).
  3. Fatal error (involuntary).
  4. Killed by another process (involuntary).

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.
- Context of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.
Process Creation

• **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes

• Generally, process identified and managed via a **process identifier** (pid)
  – Stored in PCB

---

Process Creation (2)

• **Resource sharing**
  – Parent and children share all resources
  – Children share subset of parent’s resources
  – Parent and child share no resources

• **Execution**
  – Parent and children execute concurrently
  – Parent waits until children terminate

---

Process Creation (3)

• **Address space**
  – Child duplicate of parent
  – Child has a program loaded into it

• **UNIX examples**
  – **fork** system call creates new process
  – **exec** system call used after a **fork** to replace the process’ memory space with a new program

---

Process Creation

[Diagram showing fork(), exec(), exit(), parent, child, wait, resumes]
Java programs can generate new processes, which are executed separately from the JVM process. Such processes can only be used to execute existing executable programs on the particular OS.

(. Try the previous example. Execute “java OSProcess cat” and “ps” from another terminal window... )
Some Important **System Calls** for Process Management (2)

• `short wait(byte* return_code)` - process waits (sleeps) for one of its children to terminate. Returns the pid of the terminating children. Any terminating children wakes the process. The value at `return_code` is an integer value in the range (0-255) stating some termination status. Usually, 0 is used to represent successful termination of corresponding child.

Some Important **System Calls** for Process Management (3)

• `void exit(byte status)` - terminates the current process while returning the value of status. Such value is the one captured by the parameter of the parents wait statement (if any) that is affected by the termination. Terminating process goes into **zombie state** ...

Some Important **System Calls** for Process Management (4)

• `void execl(...)` - replaces the entire content of the current process’s address space by the address space corresponding to the executable file identified by the parameters and with the initial argument values stated.

Some Important **System Calls** for Process Management (5)

• Parameters is a variable number of string values. Last parameter is 0 (end of the list).
  - first parameter: name of the executable file
  - second parameter: name of the program (may be different from the name of the file). Such name is the first parameter received by the first entry of the **char argument in the main function of the corresponding executable** ...
  - other parameters are assigned to positions 1, 2, ... of main’s argument array ...

  (recall int main(int argc, char* argv[]) {...})
Some Important **System Calls** for Process Management (6)

- Other system calls: *open, close, read, write* (see some examples)
  
  Note: the file descriptor is a number (0 ...): the min value available when the *open* sc is executed.

---

**A tree of processes on a typical Solaris**

---

Some Important **System Calls** for Process Management (7)

- *pipe(int[] fd)* - Creates an external structure that allows communication between parent and child process. Different processes may share a pipe by sharing the corresponding file descriptors. Parameter is an array of two entries. Entry $fd[0]$ is set to the file descriptor for the output end of the pipe. Entry $fd[1]$ is set to the file descriptor for the input end of the pipe.

---

**Process Termination (1)**

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process’ resources are deallocated by operating system
Process Termination (2)

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - If a process terminates, all children must be terminated too (recursively) - **cascading termination**

Interprocess Communication (IPC)

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **IPC**
- Two models:
  - Shared memory
  - Message passing

Communications Models

![Diagram of Communications Models](image)

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Producer-Consumer Problem (1)

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded implementation of a Buffer

```java
public class BoundedBuffer<E> implements Buffer<E> {
    private static final int BUFFER_SIZE = ...;
    private int count; // items in buffer
    private int in;    // next free position
    private int out;   // next full position
    private E[] buffer;

    public BoundedBuffer() {
        count = 0; // initially empty
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    ... other methods next....
}
```

NOTE: This solution has some synchronization problems that we shall address later in the course ... for the moment ...

Producer-Consumer Problem (2)

Java Specification:

```java
public interface Buffer<E> {
    // producers call this method
    public void insert(E item);

    // consumers call this method
    public E remove();
}
```

Bounded implementation of a Buffer (2)

```java
public class BoundedBuffer<E> implements Buffer<E> {
    ... other methods next....

    // producers call this method
    public void insert(E item) {
        while (count == BUFFER_SIZE); // WAIT - if full
        buffer[in] = item; //place item
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }

    // consumers call this method
    public E remove() {
        E item;
        while (count == 0) ; // WAIT - is empty
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        return item;
    }
}
```
Simulating Shared Memory in Java

Interprocess Communication – Message Passing (1)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`

Interprocess Communication – Message Passing (2)

- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
Direct Communication

• Processes must name each other explicitly:
  – `send (P, message)` – send message to process P
  – `receive(Q, message)` – receive a message from process Q

• Properties of communication link
  – Links are established automatically
  – A link is associated with exactly one pair of communicating processes
  – Between each pair there exists exactly one link
  – The link may be unidirectional, but is usually bi-directional

Indirect Communication

• Messages using mailboxes (or ports)
  – Each mailbox has a unique id
  – Processes can communicate only if they share a mailbox

• Properties of communication link
  – Link established only if processes share a common mailbox
  – A link may be associated with many processes
  – Each pair of processes may share several communication links
  – Link may be unidirectional or bi-directional

Indirect Communication

• Operations
  – create a new mailbox
  – send and receive messages through mailbox
  – destroy a mailbox

• Primitives are defined as:
  • `send(A, message)` – send a message to mailbox A
  • `receive(A, message)` – receive a message from mailbox A

Indirect Communication

• Mailbox sharing
  – \( P_1, P_2, \) and \( P_3 \) share mailbox A
  – \( P_1 \), sends; \( P_2 \) and \( P_3 \) receive
  – Who gets the message?

• Solutions
  – Allow a link to be associated with at most two processes
  – Allow only one process at a time to execute a receive operation
  – Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered *synchronous*
  - `send` sender blocks until the message is received
  - `receive` receiver blocks until a message is available
- **Non-blocking** is considered *asynchronous*
  - `send` sender sends message and continue
  - `receive` receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of $n$ messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

Examples of IPC Systems - POSIX

- Process first creates shared memory segment
  - `segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- Process wanting access to that shared memory must attach to it
  - `shared memory = (char *) shmat(id, NULL, 0);`
- Now the process could write to the shared memory
  - `sprintf(shared memory, "Writing to shared memory");`
- When done a process can detach the shared memory from its address space
  - `shmdt(shared memory);`

Examples of IPC Systems – Windows XP (1)

- Message-passing centric via **local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
Examples of IPC Systems – Windows XP (2)

- Communication works as follows:
  - The client opens a handle to the subsystem’s connection port object
  - The client sends a connection request
  - The server creates two private communication ports and returns the handle to one of them to the client
  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

LPCs in Windows XP

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a (unique) pair of sockets
Sockets (2)

- Different types:
  - Connection Oriented (TCP)
  - Connectionless (UDP)
  - Multicasting...

Socket Communication in Java (1)

```java
public class DateServer {
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            // now listen for connections
            while (true) {
                Socket client = sock.accept();
                PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
                pout.println(new java.util.Date().toString());
                // close the socket and resume
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems; built on top of local IPC mechanism
- **Stubs** — client-side proxy for the actual procedure on the server, typically one stub per each remote procedure
- The client-side stub locates the server and *marshalls* the parameters (packaging in a form that can be transmitted, endianess...)
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object

RMI Example

- Begin by declaring and interface that specifies the methods that can be invoked remotely...

```java
public interface RemoteDate extends Remote {
    public abstract Date getDate() throws RemoteException;
}
```
• A thread is an execution context for a processor, which handles an execution of the instructions in a process or part of them.
• In a system that does not support multiple threads in a process, each process has only one thread.
• In a system that supports multiple threads in a process, there can be many threads executing concurrently on the same process.

Each thread in a process can be executing a different portion (method or subroutine) of the process’s code.
• In a uniprocessor system, they alternate execution as different processes do. In a multiprocessors system, they can execute in parallel.
• For each thread, the following data is maintained: ID, program counter, register set, and stack. This is in addition to the other information maintained in PCB of the owner process.
Single and Multithreaded Processes

Thread Content

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>Stack</td>
</tr>
<tr>
<td>Pending alarms</td>
<td>State</td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-7. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

Thread use

- Potential uses of threads:
  - Word Processor – 1 thread display, 1 thread keystrokes, 1 thread spelling & grammar
  - Web browser – 1 thread display images, 1 thread network
  - Web server – ???

Example: Multithreaded Server Architecture

1. Initialization
2. While (true)
   a. get request
   b. initiate thread to serve request concurrently

General Algorithm Scheme followed by server:

1. request
2. create new thread to service the request
3. resume listening for additional client requests
Benefits (1)

- **Responsiveness** - threads allow a program to continue executing even if part of it is blocked or is performing a lengthy computation. This may result in faster responses during program execution.
- **Resource Sharing** - data and resources can be shared in an efficient manner. That kind of sharing in processes is achieved by message passing or by shared memory, and at a major cost in terms of effort from the programmer.

Benefits (2)

- **Economy** - several of the overhead issues involved in creating and managing processes are highly minimized in threads.
- **Scalability** - benefits or multithreading can be greatly increased in multiprocessor architectures.

Benefits (3)

**Thread vs Process Cost**

<table>
<thead>
<tr>
<th></th>
<th>User threads</th>
<th>LWP/kernel threads</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation time</td>
<td>52</td>
<td>350</td>
<td>1700</td>
</tr>
<tr>
<td>Synchronization with semaphores</td>
<td>66</td>
<td>390</td>
<td>200</td>
</tr>
</tbody>
</table>

* on SPARC station 2 (Solaris), from Unix Internals by Uresh Vahalia, PH 1996.

Multicore Programming (1)

- Multicore or multiprocessor systems putting pressure on OS designers as well as on application programmers.
- Challengers for OS designers include:
  - Scheduling to allow efficient use of cores and parallel execution
  - Synchronization of threads running in different cores.
**Multicore Programming (2)**

- Challengers for application programmers include:
  - **Dividing activities** - which can be parallelized
  - **Balance** - equal effort on each parallel activity
  - **Data splitting** - divide data managed by each activity
  - **Data dependency** - between different activities and properly synchronize responsible threads.
  - **Testing and debugging** - involves more effort than in sequential processes since order of execution of parallel tasks may be undetermined.

**Concurrent Execution on a Single-core System**

- Concurrency = Interleaved execution (time sharing systems)

**Parallel Execution on a Multicore System**

- Concurrency = Parallel execution (one thread/core + time sharing)

**Management of Threads (1)**

- Thread management can be provided at either:
  - **user level** (user threads) - are supported above the kernel
  - **kernel level** (kernel threads) - are supported and managed directly by the OS.

- Virtually all modern OS’s support kernel threads
  - Windows XP/2003, Solaris, Linux, Tru64 UNIX, HP-UX, Mac OS X
Management of Threads (2)

- Ultimately, there must be a relationship between a running user thread and a kernel thread to allow the execution of that user thread.
- Different models are used to match user threads to corresponding kernel threads for their execution.

Many-to-One

- Many user-level threads mapped to single kernel thread.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Threads execute concurrently, but not in parallel...
Threads are scheduled by process itself.

One-to-One

- Each user-level thread maps to kernel thread.
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

Different threads in a process can run in parallel. The kernel schedules them...
(-) more overhead since kernel needs to create one kernel thread for each user thread created.

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads (multiplexed).
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package.
Two-level Model
• Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
• Examples
  – IRIX
  – HP-UX
  – Tru64 UNIX
  – Solaris 8 and earlier

Thread Libraries
• **Thread library** provides programmer with API for creating and managing threads
• Two primary ways of implementing
  – Library entirely in user space
  – Kernel-level library supported by the OS
• Example of thread libraries:
  – POSIX Pthreads
  – Win32 threads
  – Java threads

Pthreads
• May be provided either as user-level or kernel-level
• A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
• API specifies behavior of the thread library (specification), implementation is up to development of the library
• Commonly implemented in UNIX operating systems (Solaris, Linux, Mac OS X)

```c
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of attributes for the thread */
  if (argc != 2) {
    fprintf(stderr, "usage: a.out <integer value>\n");
    /*exit(1);*/
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr, "Argument %d must be non-negative \n", atoi(argv[1]));
    /*exit(1);*/
    return -1;
  }
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
```

Pthreads (example)
Pthreads (example)

```c
/* now wait for the thread to exit */
pthread_join(tid,NULL);
printf("sum = %d\n",sum);
}
/**
 * The thread will begin control in this function
 */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;
  if (upper > 0) {
    for (i = 1; i <= upper; i++)
      sum += i;
  }
  pthread_exit(0);
}
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface (most common)

Java Threads (1)

- Runnable Interface
  ```java
  public interface Runnable {
    void run();
  }
  ```
- A class implementing the interface defines objects corresponding to threads
  - the thread object will start executing its run() method...

Java Threads (2)

- To create and start the execution of a thread object of type Runnable, the following is required:
  1. Create an object of type Thread while passing the and instance of the Runnable object as parameter of constructor.
  2. To that object of type Thread apply method start().

  *** The new thread will automatically start executing the run() method of the associated Runnable object.
Given an array of \( N \) integers and a fixed value \( d \), create \( M \) threads to add all its elements in portions of size \( d \).

\[
M = \begin{cases} 
\frac{N}{d} & \text{if } N \% d = 0 \\
\frac{N}{d} + 1 & \text{if not}
\end{cases}
\]

One possible solution:

- Main thread does the following
  - Create several threads
  - Each thread is assigned a portion of the array
  - Each thread computes the sum of its assigned portion
  - Results are placed in appropriate entries in a shared array

```java
public static void main(String[] args) {
    int[] a = ...; // the array to add
    int pSize = ...; // size of blocks
    int nThreads = (a.length % pSize == 0 ? a.length / pSize : a.length / pSize + 1);
    Barrier barrier = new Barrier(nThreads);
    int[] sum = new int[nThreads];

    for (int i=0; i<nThreads; i++) {
        Thread t = new Thread(new ArrayPortionAdder(a, sum, i, pSize, barrier));
        t.start();
    }

    // wait for all threads to finished as per the barrier object...
    try {
        barrier.waitForRelease();
    } catch (InterruptedException e) {} 

    // compute the final sum from the array sum ...
    ...
}
```
Java: Sum of Array Elements (2)

```java
public class ArrayPortionAdder implements Runnable {
    private int[] arr, sum;
    private int index, size;
    private Barrier barrier;
    public ArrayPortionAdder(int[] arr, int[] sum, int index, int size, Barrier b) {
        this.arr = arr;
        this.sum = sum;
        this.index = index;
        this.size = size;
        this.barrier = b;
    }
    public void run() {
        int low = index * size;
        int sup = low + size - 1;
        if (sup >= arr.length)
            sup = arr.length - 1;
        int asum = 0;
        for (int i = low; i <= sup; i++) { // thread sums its part
            asum += arr[i];
        }
        sum[index] = asum; // places result in shared array
        // count one more thread as finished
        barrier.incCount();
    }
}
```

Java: Object Type for Synchronization

```java
public class Barrier {
    private int threshold, count = 0;
    public Barrier(int t) {
        threshold = t;
    }
    public void reset() {
        count = 0;
    }
    public synchronized void waitForRelease() throws InterruptedException {
        while (count < threshold)
            wait();
    }
    public synchronized void incCount() throws InterruptedException {
        count++;
        if (count==threshold)
            notifyAll();
    }
}
```

Java Threads States

- **New** - Thread created but run() method has not started
- **Runnable** - when the run() method starts
- **Blocked** - thread is waiting for a lock
- **Waiting** - thread is waiting for an action of another thread (example: executing join())
- **Timed waiting** - same as waiting for for a specified maximum time. Useful to eliminate the possibility of starvation...
- **Terminated** - when the run method of the thread has finished.

*** This an example of a class used for synchronization – we shall study this in more detail later in the course....
**Threading Issues**

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

**Semantics of fork() and exec()**

- Does fork() duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of fork():
  - one that duplicates all threads active in the parent process
  - one that duplicates only the thread invoking the fork()

**Thread Cancellation (1)**

- Terminating a thread before it has finished
  - Multiple threads performing database search, one returns value, others might be canceled…
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
    - Case where resources have been allocated to canceled thread, thread canceled while updating data needed by other threads
    - java – stop() (deprecated)

**Thread Cancellation (2)**

- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
  - pthreads – cancellation points
  - java – interrupt(), isInterrupted()
Signal Handling (1)

• Signals are used in UNIX systems to notify a process that a particular event has occurred
• A signal handler is used to process signals
  – Signal is generated by particular event
  – Signal is delivered to a process
  – Signal is handled

Signal Handling (example)

```c
/* Sample program to handle system signals */
#include <signal.h>
#include <stdio.h>

void * myhandler(int myint)
{
    printf("\nSignal Handled!!\n\n");
    exit(0);
}

int main()
{
    signal( SIGINT, (void *) myhandler );
    signal( SIGTERM, (void *) myhandler );

    while(1) {
        printf("Doing Nothing...\n");
        sleep(1);
    }
}
```

Signal Handling (2)

• Options:
  – Deliver the signal to the thread to which the signal applies
  – Deliver the signal to every thread in the process
  – Deliver the signal to certain threads in the process
  – Assign a specific thread to receive all signals for the process

Thread Pools (1)

• To create a new thread, although simpler than to create a process, has its impacts on the system:
  – To create a new thread consumes time
  – Too many active threads may negatively impact system’s performance
Thread Pools (2)

• An alternative is to create a number of threads in a pool where they await work. This has advantages:
  – Usually slightly faster to service a request with an existing thread than create a new thread
  – Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Pools in Java (1)

• Java allows the creation of thread pools as shown next.
• Java classes to support thread pools:

```java
public interface Executor {
    // executes the given thread
    // at some time in the future
    void execute(Runnable t);
}
```

Thread Pools in Java (2)

• Class Executors - provides a set of useful methods to create objects that are relevant to thread management in JVM. One static method creates a pool of a given number of threads. Such pool is created as an object of type ExecutorService.

```java
public static ExecutorService newFixedThreadPool(int maxNoThreads)
```

• The following creates a thread pool:

```java
Runnable r1 = new Worker();
Runnable r2 = new Worker();
Runnable r3 = new Worker();
ExecutorService pool = Executors.newFixedThreadPool(3);
pool.execute(r1);
pool.execute(r2);
pool.execute(r3);
pool.shutdown(); // shuts down the pool only after all threads have finished
```

Thread Pools in Java (3)

• Example of a Java thread pool:

```java
public class Worker implements Runnable {
    private static int NT = 0;
    public void run() {
        System.out.println(Thread number + (++NT));
    }
}
```
Thread Specific Data

- Allows each thread to have its own copy of data
  - Transaction processing system (each thread handle separate transaction)
- Useful when you do not have control over the thread creation process (i.e., when using thread pools)

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_order()</td>
<td>trans_T1</td>
</tr>
<tr>
<td>payment()</td>
<td>trans_T3</td>
</tr>
<tr>
<td>order_status()</td>
<td>trans_T5</td>
</tr>
<tr>
<td>delivery()</td>
<td>trans_T2</td>
</tr>
<tr>
<td>stock_value()</td>
<td>trans_T4</td>
</tr>
</tbody>
</table>

Scheduler Activations

- Both M:M and Two-level models require communication to dynamically maintain the appropriate number of kernel threads allocated to the application
- Place a data structure between user and kernel threads (lightweight process)
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Windows XP Threads (1)

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area

Windows XP Threads (2)

- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)
Windows XP Threads

Linux Threads (1)

- Linux refers to them as *tasks* rather than *threads or processes*.
- Thread creation is done through `clone()` system call.
- `clone()` allows a child task to share the address space of the parent task (process), several flags control amount of sharing between parent-child, if no flags are set:
  \[ clone() = fork() \]

Linux Threads (2)

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>