Processes

Agenda

- What constitutes a process
- Process control block
- Overview of process scheduling
- Processes and threads
- Inter-process communication
- Network communication
- More on multithreading
- UNIX signals
- Saving and restoring contexts in UNIX

What constitutes a process

- A process is an executable in action
  - An executable is a passive entity and a process is an active instantiation of the executable
  - A new process gets created when an executable starts running
  - At any point in time, there may be several processes inside the system (all may not be executing)
  - Multiple processes may execute the same program
    - Multiple open shells, multiple internet browsers
  - Process states: created, ready, executing, waiting/sleeping, terminated

What constitutes a process

- Handling multiple processes is necessary to maximize hardware resource utilization
  - User executables run as user processes
  - Kernel executables run as kernel processes
  - Hardware cannot distinguish between these without looking at the mode bit
- Each process has its own text, data, and stack regions
  - Stack and heap grow in opposite directions
  - Intermediate values are maintained in processor registers; stack pointer and the program counter are two special registers

What constitutes a process

- Anything that is needed to reconstruct the state of a process is in the process context
  - Processor registers
  - Text, data, stack regions in memory
  - Various kernel data structures such as list of open files and their seek pointers (encoded in terms of the UFDT entry), etc.
  - The process control block (PCB) is a kernel data structure that maintains all information pertaining to the process context including the current state of the process

Process control block

- The PCB entries start off as invalid for a newly created process
- The process enters the ready queue by linking its PCB to the tail of the queue
  - Ready queue is a linked list of PCBs of processes that are ready to execute
- On transition from executing to sleep state
  - The PCB is used to remember the process context by maintaining
    - a pointer to the memory region holding the saved registers and other context information
    - a pointer to the process region table

Process control block

- On transition from executing to sleep state
  - The PCB is linked at the tail of the waiting list of PCBs for the particular device or event
- On transition from ready to executing state
  - Context is restored from PCB
  - Known as a context switch
- The PCB of a non-executing process is either in the waiting queue of some device/event or in the ready queue

Overview of process scheduling

- Every process in the ready queue can request CPU cycles for execution
  - An algorithm must select one process per processor for execution; this is the scheduling algorithm
  - Two types of process schedulers: short-term and medium-term schedulers
- A short-term scheduler selects one process from the ready queue for execution
  - Invoked whenever the currently executing process enters the sleep state or encounters a timer interrupt or terminates
  - Decides the overall utilization of the CPU

Overview of process scheduling

- Two types of processes
  - CPU-bound: These processes mostly compute i.e., major portion of life spent in ready or executing state
  - I/O-bound: These processes spend significant amount of time in I/O i.e., major portion of life spent in sleep state
  - The time spent computing between two consecutive I/O operations is known as a CPU burst
  - A CPU burst of a process can get executed through multiple transitions between the ready and executing states culminating into a sleep state when the CPU burst completes
    - The CPU burst of a process does not include the time spent waiting in the ready queue

Overview of process scheduling

- Primary goal of a short-term scheduler
  - Every time a scheduling event occurs, it should pick a process so that
    - all I/O bursts are overlapped by CPU bursts
    - the CPU is busy all the time executing the CPU burst of some process (this may still not lead to 100% CPU utilization)
    - the scheduled process can execute for the full allocated scheduling quantum (minimizes the chance of too frequent context switches); this is usually hard to guarantee
- Scheduling quality plays an important role in determining the degree of multiprogramming

Overview of process scheduling

- Medium-term scheduler
  - If the system is running low on resources (e.g., memory), some processes may have to be swapped out from memory to disk and later swapped in when needed
  - The medium-term scheduler selects the processes to be swapped out from memory and swapped in from disk
  - Not very critical to overall performance as long as a currently running process is not swapped out
  - Invoked whenever the resource usage goes above or falls below a threshold (usually infrequent)

Overview of process scheduling

- Long-term scheduler
  - Found in old computers where jobs (or processes) used to be submitted in batches
  - The long-term scheduler decides which of the submitted jobs will be loaded in memory and entered in the ready queue
  - Invoked whenever a job completes so that a new job can be loaded (even less frequent than the medium-term scheduler)

Processes and threads

- A process has a single thread of control
  - This thread refers to the sequence of instructions executed by the process
  - A fork() call generates a new thread of control
    - Is it a thread or a process? Often used interchangeably
- Historically, a thread refers to a piece of code in execution and is a part of a parent process
  - As a result, lighter-weight than a full process

Processes and threads

- Multiprocessing: executing independent or communicating processes simultaneously (or in a time-shared manner)
- Multithreading: executing possibly dependent and/or communicating parts of the same process simultaneously in different threads of control
- Multiprogramming: executing independent programs in different processes simultaneously
- Process scheduler is usually not aware of the dependencies, if any, between the processes in the ready queue
  - Does the scheduler see threads or processes?

Processes and threads

- Threads can be user-level and kernel-level
  - Depends on whether the process scheduler sees processes or threads as scheduling units
  - If the scheduler can control the scheduling of threads, it will consider multithreading as a form of multiprogramming
    - Threads are just independent execution units from the perspective of a scheduler
- Today, multithreading refers to shared memory parallel programming
  - Slave threads are spawned by a master process using fork() or some other threading library calls

Process creation

- A process is always created by another process
  - Created process is the child of the creating process
- The system boots up as a process
  - *init* process in UNIX, *sched* process in Solaris
  - This is the root of all processes
  - Every process gets a unique integer ID known as the process ID or pid. The root process has pid zero.
    - In UNIX, the system boot process has pid zero and *init* has pid one.
- A process can be created by calling fork()
  - Child pid is returned to parent, zero is returned to child. A negative return value indicates error in UNIX.

Process creation

- Typical process tree in Solaris
  sched → init, pageout, fsflush
  init → inetd, dtlogin
  inetd → telnetdaemon → csh →…
  dtlogin → Xsession → sdt_shel → csh →…
- The *pstree* command shows the process tree in UNIX (*pstree –p* shows the pid also)
- The *pgrep processname* command shows the pid of a process
  – *pgrep init* returns 1

Process creation

- To find the parent pid of a process with pid x
  – Check the fourth entry in file /proc/x/stat on UNIX
- When the fork() call returns
  – The child process has been created
  – Its text and global data are loaded in memory
  – Its PCB holds the starting program counter
  – Its PCB is linked to the tail of the ready queue
  – The process will start executing when it is scheduled

Process termination

- A process usually terminates when it calls exit()
- If a process terminates, all its children may have to be terminated depending on the kernel
  – Known as cascaded termination
  – Part of the exit system call
  – Does not happen in UNIX. Children get attached to *init*. In some implementations, the owner of these processes cannot terminate a session until all processes terminate (e.g., in RHEL, but not in ubuntu)
- The return value of wait() is the pid of terminated child

Inter-process communication

- Two ways to communicate between processes
  – Shared memory and message passing
  – We will focus on communication between two processes running on the same computing node
    - No network communication involved
  – Message passing involves sending a formatted stream of bytes to another process
    - Send and receive calls involve heavy-weight system call handlers
  – Shared memory communication involves setting up a region of memory shared between a set of communicating processes
    - System calls needed to only set up the shared region

Inter-process communication

- A third possibility is through file system
  - Usually avoided due to slow disk I/O
  - May become attractive if file systems are implemented on storage media faster than magnetic disk
    - Flash memory or solid state disks

Message passing

- For each communication, two system calls and two copy operations
  - Bytes to be communicated are copied from sender's memory region to a reserved kernel region (done by send system call handler)
  - Bytes to be communicated are copied from the kernel region to receiver's memory region (done by receive system call handler)
  - The reserved kernel region is called message queue or mailbox or message port
    - We will look at UNIX message queues in detail

Message passing

- Send and receive calls come in four flavors
  - Blocking send: the send operation blocks (the calling process goes to sleep) until the message is received
  - Non-blocking send: the sender copies the message into kernel area and continues; also known as asynchronous send
  - Blocking receive: receiver process blocks until there is a matching message to be consumed
  - Non-blocking receive: the process calling receive either gets a matching message or an error status, but does not go to sleep; may have to check periodically to ultimately receive the message

Message passing

- Kernel area used to deposit messages may have some amount of buffering
  - Can hold multiple outstanding posted, yet unreceived, messages
  - With zero buffering, only blocking send can be supported
    - Still possible to support non-blocking receive

UNIX message passing

- Kernel maintains message queues in the form of an array of linked lists
- Each queue is indexed by the array index
  - This index is called the message queue descriptor
- The header of each queue contains a numeric key used to identify a queue
- To create or get the handle of an already created message queue
  - msgget() call is used; takes two arguments: key to identify the queue and a flag specifying read/write/execute permissions

UNIX message passing

- The msgget() call
  - The flag argument can also be used to create a new message queue with the given key, if such a queue does not exist
    - Need to OR the permission bits with IPC_CREAT

    queue_id = msgget (100, 0777 | IPC_CREAT);
  - If IPC_PRIVATE is passed as the key, a new message queue is always created with this special key
  - Returns the queue descriptor (an integer)

    queue_id = msgget (100, 0777);
  - Notice the similarities with the open() call

UNIX message passing

- Queue descriptor
  - Kernel identifies a queue by its descriptor modulo the message array size
  - When freeing an entry, kernel increments its descriptor value by the message array size
    - Next process to occupy this entry will get a new descriptor
- Queue header
  - Key, Permission bits
  - uid and gid of the owner of the process which created the queue
  - Status information: pid of the most recent process to operate on the queue and the timestamp

UNIX message passing

- Queue header also stores
  - A pointer to the last message in the queue
  - A pointer to the first message in the queue
  - Number of messages in the queue
  - Number of data bytes in the queue
  - Maximum limit on the data bytes

UNIX message passing

- Sending a message
  - msgsnd() call is used
  - Takes four arguments: queue descriptor, a pointer to a message structure, number of data bytes in the message, and an integer flag
  - The message structure should have two fields: an integer message type and a character array holding the data bytes
  - The flag specifies what to do if there is no space in the message queue
    - IPC_NOWAIT returns the call immediately with an error
    - Specifying zero puts the calling process to sleep

UNIX message passing

- Kernel does the following on a msgsnd() call
  - Checks if the calling process has write permission to the message queue
  - Checks if the message length exceeds the system limit
  - Checks if there is space in the message queue
  - Checks if the message type is positive integer
  - Allocates memory for message data, puts the rest of the message in a message header structure, and links the message header to the tail of the queue
    - Message header contains a pointer to the data area
  - Wakes up all processes waiting for a message in this message queue and moves them to ready queue

UNIX message passing

- Receiving a message
  - The msgrcv() call is used
    - Locates the appropriate message in the queue, copies the message data from kernel area to user area, and de-allocates the message from the queue (one-to-one communication)
  - Takes five arguments: the message queue descriptor, the pointer to a message structure, number of data bytes to receive, the type of the message to receive, and an integer flag

UNIX message passing

- Execution of msgrcv() call
  - Usual read permission checks
  - Searches for a matching message type in the queue
  - If the passed type is zero, the kernel returns the first message in the queue
  - If the passed type is negative, the kernel returns the message with the smallest type provided it is less than the absolute value of the passed type
    - Recall that msgsnd() only accepts positive integer types

UNIX message passing

- Execution of msgrcv() call
  - The passed number of bytes must be at least the data size of the matching message
  - If the passed size is smaller and the flag does not specify MSG_NOERROR, the kernel returns an error status; if the flag specifies MSG_NOERROR, the message data is truncated during copying, but the entire message is de-allocated
  - On a successful receive, the message data is copied to the character array of the passed message structure

UNIX message passing

- Execution of msgrcv() call
  - The flag can specify what the kernel should do if there is no matching process e.g., put to sleep or not
  - On a successful receive, the kernel wakes up all processes that are waiting for space in this message queue
    - These processes are moved to the ready queue

UNIX message passing

- Querying and updating the message queue status
  - The msgctl() call is used
  - Takes three arguments: the queue descriptor, the query/update command, and a pointer to a user data structure containing the status to be written or read
  - The most common command is IPC_RMID, used to free a message queue descriptor
    - The last argument can be NULL in this case
  - A process querying the status of a queue must have read permission

UNIX message passing

- Querying and updating the message queue status
  - A process trying to update the status of a queue must be owned by the same user as the one who created the queue or must be owned by the owner of the queue or must be owned by a superuser
    - Queue header contains a owner user id field. This can be set through msgctl() call.

UNIX shared memory

- One of the processes creates the shared memory region
- Once created, the region must be attached to the address space of all communicating processes
- Kernel maintains a shared memory table
  - As usual, entries point to the system-wide kernel region table entries (similar to the process region table entries)
  - The integer index of a shared memory region in the shared memory table is the descriptor of the shared memory region

UNIX shared memory

- To get the descriptor or to create a shared memory region
  - The shmget() call is used
  - Takes three arguments: a shared memory region key (an integer value), the size of the region in bytes, and the permission flag ORed with IPC_CREAT for creating the region
    - The second argument is used only if IPC_CREAT is specified in the third argument
  - The shmget() call triggers a search over the shared memory table for an entry with the matching key and proper permissions
    - Returns the descriptor of the region

UNIX shared memory

- Two flags are set at the time of assigning the entry to a region
  - One flag in the shared memory table entry indicates that the shared memory region has not yet been allocated and will be allocated when a process attaches itself to this region
  - Another flag in the region table entry indicates that the region should not be freed even if the attach count falls to zero (to start with, it is zero)
    - We will soon see why
    - Each entry of the region table maintains a count of the number of processes attached to this region

UNIX shared memory

- Attaching a shared memory region to a process
  - The shmat() call is used
  - Three arguments: the region descriptor, the starting address of the attached region, and an integer flag
    - Returns a char pointer indicating the starting address of the attached region (similar to malloc)
  - The passed starting address may not be obeyed by the kernel
  - The flag argument specifies if the region is read-only and if the kernel can round-off the user-specified starting address
    - Commonly passed value for the last two arguments is zero, which leaves all options entirely to the kernel

UNIX shared memory

- Attaching a shared memory region to a process
  - Once the memory region is allocated and attached, the shared memory table is updated to record this fact and the attach count is increased by one

UNIX shared memory

- Detaching a shared memory region from a process
  - The shmdt() call is used
  - Takes one argument: the starting char pointer (this was the return value of shmat)
  - Decrements the attach count by one
    - The attach count is also decremented when an attached process calls exit()
  - Kernel searches through the shared memory table looking for a match and once found, the time of last detach is updated

UNIX shared memory

- Querying and updating the status of a shared memory region
  - The shmctl() call is used
  - Takes three arguments (like msgctl): the region descriptor, the query/update command, and the pointer to a user data structure to return the status
    - The most popular command is IPC_RMID, in which case the last parameter can be NULL
    - The shared memory table entry is freed and the region table entry is updated to indicate that when the attach count reaches zero, the region should be de-allocated
    - After an IPC_RMID command is issued, no new process can attach to the region (the shared memory table entry does not exist any more)

Network communication: sockets

- Sockets can be used to communicate between two processes running on two different computers
  - Sockets can also be used to communicate between two processes running on the same computer
  - Socket is a kernel construct interfacing between the system call layer and the network transmission protocol
    - Transmission protocol layer talks to the Ethernet drivers
  - A socket is specified by concatenating a port number with an IP address
    - One socket per communicating process

Network communication: sockets

- Socket communication requires one socket for each communicating process
  - One process on computer X listens on port P_X and another process on computer Y listens on port P_Y
  - Server processes implementing specific services such as telnet, ftp, ssh, and http listen on fixed "well-known" ports
    - Telnet: port 23, ssh: port 22, ftp: port 21, http: port 80
    - All ports below 1024 are reserved for standard services
  - When a client initiates a socket communication, it is assigned a port number bigger than 1024
    - Example: a http client on host 146.86.5.20 may get port 1625 to communicate with a web server running on 161.25.19.8 through port 80

Network communication: sockets

- Socket communication is done through several system calls (UNIX functions are listed below)
  - Get a socket descriptor: socket()
  - Connect to a server (described by the IP address) at a certain port: connect()
  - Bind a socket descriptor to an IP address and port: bind()
  - Prepare an unconnected socket to accept incoming connection requests directed to this socket: listen()
  - Accept a connection from a client: accept()
  - Send and receive: send(), sendto(), recv(), recvfrom()
  - Close a connection: close()

Network communication: sockets

- Typical call sequence on a server
  - socket() to get a descriptor
  - bind() to a well-known port
  - listen()
  - accept(), blocks until any connection from client
  - recv(), send(), recv(), send(), …
  - close()
- Typical call sequence on a client
  - socket() to get a descriptor
  - connect() to establish a connection with server
  - send(), recv(), send(), recv(), …
  - close()

More on multithreading

- Recap on threads
  - Every thread is attached to some parent process
  - Threads share code, data, files with parent process
  - Each thread has its own register values and stack region
  - Multithreading is preferred to multiprocessing to accomplish a task
    - Thread creation takes less time than process creation
  - Threads can be user-level or kernel-level
    - User-level threads can be made visible to the kernel by "attaching" them to kernel-level threads

More on multithreading

- How is a system call in a thread of a multithreaded process handled?
  - Many-to-one model: all system calls from any user thread in a process are handled by a single kernel-level thread
    - Any I/O call from any thread would lead to process switch
  - One-to-one model: every user thread gets attached to a distinct kernel-level thread
    - Found in Linux and all recent Windows and Solaris versions
  - Many-to-many model: a pool of user threads is attached to a pool of kernel threads
  - Hybrid model: many-to-many and one-to-one
    - Found in old versions of Irix, HP-UX, Solaris

More on multithreading

- Multithreading is usually implemented using the APIs of a thread library
  - A library can implement user-level or kernel-level threads
    - Gets decided by how a system call from a thread gets handled
    - A library implementing kernel-level threads necessarily requires supports from the kernel
  - POSIX thread library is popular in UNIX
    - Can implement user-level or kernel-level threads
  - Windows environment offers a set of APIs for multithreading implementing kernel-level threads

UNIX clone() call

- The clone() call is used to create a light-weight process or a thread
  - Present in Linux systems and used by pthread_create
  - Similar to fork() except that the created thread can share a part of parent's context
  - Takes four arguments: a function pointer where the child will start execution, the starting address of the child's stack, an integer flag specifying what is shared with the parent (bitwise Ored), and the argument list to the starting function
    - The function must have integer return type and a void* argument

UNIX clone() call

- The commonly passed flags for sharing in the third argument
  - CLONE_FS for sharing the file system
  - CLONE_FILES for sharing the file descriptor table
  - CLONE_VM for sharing memory
  - The lowest byte of the flag is used specify the signals delivered to the parent when the child terminates
    - Only SIGCHLD is commonly specified
- Both fork() and clone() calls could be implemented using the same system call sys_clone with different parameters
  - fork() is a special case of clone()

UNIX clone() call

- Thread libraries may want all the threads created by a process to share the same pid as the creator process
  - Obviates the need to deal with a large number of processes
  - Specifying CLONE_THREAD in the third argument makes each created thread inherit the pid of the process calling clone(), but each thread gets a unique tid and all threads are placed in the same thread group
    - The shared pid is also known as the thread group id (tgid)
    - This is what POSIX thread library uses to create the threads

UNIX clone() call

- When CLONE_THREAD is specified in the third argument of clone()
  - The created thread is said to have been detached from the creator
    - There is no parent-child relationship between these two threads
    - All created threads are siblings of the creator
    - They all inherit the parent of the creator as their parent
  - Cannot use wait() to wait for the threads in a thread group to complete
    - Need more involved solutions
    - Will discuss soon

Signal handling in UNIX

- A signal is used to notify a process that a particular event has taken place
  - A signal may be delivered to a process synchronously or asynchronously
    - A signal is delivered synchronously to a process if the event raising the signal is executed by the same process e.g., arithmetic exception, illegal memory access, etc.
    - A signal is delivered asynchronously to a process if the event raising the signal is external to the process e.g., terminating a process by hitting ctrl-c
  - A signal is handled by a process by executing either a default handler provided by the kernel or a user-defined handler

Signal handling in UNIX

- In a multithreaded process
  - A synchronous signal is delivered to the thread raising the signal
  - It is not clear which thread should receive an asynchronous signal
    - In most cases, the sender of the signal specifies which thread should receive the signal

Signal handling in UNIX

- Process group
  - Contains a number of processes
    - Different from file access permission groups
  - Useful for sending a common signal to a set of processes
  - The processes in a group have identical process group id
  - A process can set its group id to its own pid by invoking setpgrp(). Returns a new group id which is same as the pid
  - A child inherits its parent's process group id during fork() or clone() call

Signal handling in UNIX

- A process can send an asynchronous signal to another process by invoking the kill() call
  - Takes two arguments: pid of the receiving process and the signal number
  - If zero is passed as the pid, the signal is sent to all processes in the sender's process group
  - UNIX defined 19 signals and today there are many more found in the Linux kernel to handle GUI and other advanced features
  - Some of the commonly encountered signals are SIGINT, SIGILL, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGSYS, SIGCHLD, etc.
  - POSIX thread library offers a pthread_kill function for sending a signal to a specific thread within a process

Signal handling in UNIX

- Steps involved when process P calls kill() to send a signal X to process Q
  - The kernel invokes the sys_kill system call handler
  - Sets the bit at position X in a signal bitvector residing in the process table entry of Q
    - A process can receive multiple instances of the same signal, but handles that signal only once due to one bit per signal
  - If Q is currently in sleep state, it is moved to the ready queue and the signal is handled once Q returns to the user mode after getting scheduled
  - If Q is currently executing in the kernel mode, the signal is handled once Q returns to the user mode

Signal handling in UNIX

- Steps involved when process P calls kill() to send a signal X to process Q
  - If Q is currently executing in the user mode, the signal is handled immediately after the kernel returns from the sys_kill handler or any other interrupt handler that has raised the signal
  - In general, signals may not get delivered immediately
    - But a process cannot execute anything in the user mode before handling all outstanding already delivered signals

Signal handling in UNIX

- Steps involved when process P calls kill() to send a signal X to process Q
  - Q can specify how it wishes to handle signal X through the signal() call
    - Takes two arguments: a signal number and a handler function address
    - To always terminate on receiving a signal, zero should be passed in the second argument (this is also the default)
    - To ignore a signal, one should be passed in the second argument
  - A signal handler vector is maintained in the process u area
    - Filled with zero for default behavior and changed as and when signal() calls are received from the process

Signal handling in UNIX

- Steps involved when process P calls kill() to send a signal X to process Q
  - To handle a delivered signal to a process, the kernel first resets the corresponding bit in the signal bitvector residing in the receiving process's process table entry
  - Indexes into the corresponding entry of the signal handler array residing in the u area of the receiving process
    - If the entry is zero, the process is terminated
    - If the entry is one, the kernel returns the control to the process and the process continues execution in user mode
    - In all other cases, the specified signal handler is executed

Signal handling in UNIX

- Steps involved when process P calls kill() to send a signal X to process Q
  - Invoking a signal handler involves a few steps
    - The kernel resets the corresponding signal handler array entry (if the process wants to use the same signal handler for signal X in future, the signal handler must "re-install" itself through a signal() call)
    - Sets up a new stack frame on the stack of the receiving process so that it looks like the process is about to call the signal handler function
    - Moves the stack pointer to the top of the new stack frame, does a mode switch to user mode, and sets the program counter to the first instruction of the handler
    - The signal handler is now executing in user mode; the process will return to where it left off in user code after this

Context management in UNIX

- The designer of a user-level thread library needs a way to do context switch between threads
  - These user-level threads are not visible to the kernel
  - One possibility is to understand the context layout and write low-level assembly language routines to save and restore the context
  - A better option is to use the setjmp() and longjmp() calls in UNIX for saving and restoring the context
  - The setjmp() call saves the context of the calling process into a jmp_buf structure
    - Normal return value of setjmp() is zero
    - The layout of jmp_buf structure is platform-dependent

Context management in UNIX

- The setjmp() and longjmp() calls
  - The longjmp() call takes two arguments: a jmp_buf structure filled in through a prior setjmp() call and an integer
    - Restores the context from the passed jmp_buf structure
    - The second argument serves as the return value of the prior setjmp() call when the context is restored
    - Effectively the process starts executing right after the setjmp() call through which the jmp_buf was prepared
    - It looks as if the setjmp() call has just returned
  - Notice that the longjmp() calls never return
  - The setcontext() and getcontext() calls available in Linux systems can also be used

Context management in UNIX

- Using setjmp() and longjmp() in thread libraries
  - Before a context switch is done, let us assume that the head of the ready queue is the PCB of the thread to be executed next

```
Schedule() { // called on the currently running thread's
                    // context
  if (!setjmp(my_PCB->jmp_buf)) {
     enqueue my_PCB at the tail of ready queue;
     head_PCB = dequeue head of ready queue;
     longjmp(head_PCB->jmp_buf, 1);
    }
    else return;   // This is where the restored context
                        // will execute when the current
                        // thread is scheduled later
}
```