CS-284: Introduction to Operating Systems

QUESTION/ANSWER POOL FOR TEST I

Q: What was the original purpose/goal in the design of operating systems?
A: To increase the efficiency of hardware.

Q: What is the main purpose/goal today in the design of operating systems?
A: Convenience for user, efficient utilization of the computer resources (CPU, memory, I/O devices), and expandibility.

Q: Explain why the primary purpose of having an O.S. changed from the 1960 to the 1990s.
A: In the past, the efficiency is the most important goal because the computer is much more expensive. The operating systems concentrated on the optimum use of computer resources. The systems such as batch processing, spooling, multiprogramming, and time sharing are developed to reduce the cpu idle time which is wasted. In 1990s, as the cost of hardware decreased, it was feasible to have a computer dedicated to a single user. The cpu utilization is not a primary concern. The user's convenience has become more important. OS such as MS-DOS, Windows, and Apple Macintosh have been developed.

Q: What is spooling?
A: The use of secondary memory as buffer storage to reduce processing delays when transferring data between peripheral equipment and the processors of a computer

Q: Early system designers wanted to increase CPU efficiency by increasing the number of programs in memory from 1 to some small n. Explain the reasoning behind that goal; ie. why would having more programs in RAM increase total CPU efficiency?
A: Since several jobs can be kept simultaneously in memory, when one job needs to wait, the CPU is switched to another job and execute it. After the first job finishes its waiting, it can get the CPU back. Thus there will always be some jobs resident in memory and ready to execute, the CPU will never be idle, and the overhead associated with task switching will be small.

Q: True or False
Spooling is the use of primary memory (RAM) as buffer storage to reduce processing delays when transferring data between peripheral equipment and the processors of a computer.
A: False. Spooling uses secondary memory as buffer storage, not the primary memory.

Q: True or False
Multiprogramming (having more programs in RAM simultaneously) decreases total CPU efficiency.
A: FALSE. It increases CPU efficiency (see question above)

Q: True or False
Unlike time sharing systems, the principal objective of batch processing systems is to minimize response time.
A: False.
Principal objective for time sharing is to minimize response time. Whereas, the principal objective for batch processing is to maximize processor use.

Q: True or False
   A forked process shares its parent's data space at all times.
A: FALSE

Q: True or False
   A forked process may share code space with its parent.
A: TRUE

Q: True or False
   A forked process may share a Run-Time Stack with its parent
A: FALSE

Q: Who are the 2 individuals generally credited with the invention of C/Unix?
A: Ken Thompson and Dennis Ritchie.

Q: What is dual mode operation?
A: User mode and monitor mode.

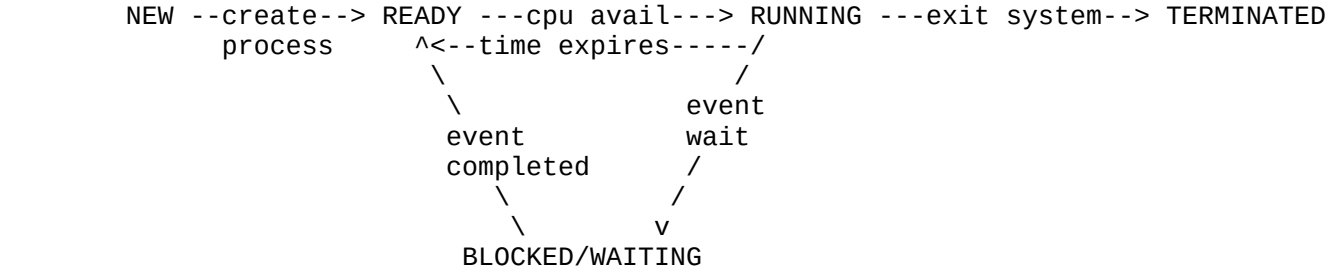
Q: Why does a machine need dual mode operation?
A: To ensure proper operation, we must protect the operating system and all
the programs and their data from any malfunctioning program. Any shared
resource is needed to be protected. Dual mode operations can protect
the OS from errant users, and errant users from one another. The
protection is accomplished by designating some of the machine
instructions that may cause harm as "privileged" instructions.

Q: What is a context switch?
A: Switching the CPU to another process.

Q: What information is saved and restored during a context switch?
A: Context switch requires saving the state of the old process and
   loading the saved state for the new process. Process state
   minimally includes current contents of registers, program counter,
   stack pointer, file descriptors, etc.

Q: Why are context switches considered undesirable (to be minimized) by OS
   designers?
A: Because context switches waste a considerable amount of CPU time when
   they save and load the state of the processes.

Q: Draw a simple Unix 'Process State' graph.
A:



Q: Give an example event for each transition to occur.(You need to
   give at least five example events).
A:{create process}: user starts a new process or a program
      issues a fork() or a thr_create() call.

```
  {cpu available}: the process currently occupying CPU stops running
     and the process in front of the READY queue is scheduled to run
  {time expires}: currently running process uses up its alloted time
     slot for this turn (timer interrupt).
  {event wait}: the process issues an I/O read
  {event completed}: DMA controller interrupts CPU signalling the
     completion of an I/O read
  {exit system}: process terminates or segmentation fault

Q: Give an example of something that a process might do to initiate a
   voluntary context switch.  Relate this answer to the Process State graph.
A: When a process initiates an I/O Read/Write or event wait(). In the Process
   State graph, the transition is from Running State to Waiting State.


Q: Give an example of something that would cause a process to experience an
   involuntary context switch.  Relate this answer to the Process State graph.
A: When a process is interrupted by an exernal source such as a timer.
   In the Process State graph, the transition is from Running to Ready State.

Q: True or False
   If a process uses up its allocated time slot, a timer interrupt occurs
   and the process is placed in a Blocked Queue.
A: False. It is placed in Ready queue.

Q: What is changed by putting '&' at the end of a command line to unix?
   Explain this in terms of the shell's behavior.
A: Running the job in the background

Q: How do you change the name of a file in Unix?
A: mv filename1 filename2

Q: If the following shell script is stored in an executable file named
   ``dirList'' and then is executed by issuing the command ``dirList .'',
   what would happen?

     PATH=$path:/bin:usr/local/bin:$HOME/bin
     # input argument: a directory name
     cd $1
     echo " "
     echo $1 : directory
     ls

     for i in *
     do if test -f $i
             then :
             else  dirList $i
             fi
     done

A: Displays all the files and the directories in the given directory
   and the subdirectories (in the directory tree structure) recursively.


Q: Give two reasons why the following block of code is logically
   and semantically wrong.

     if ( fork() == 0 )
     {    printf("Howdy partner");
           args = n;
```

```
            execl("/afs/umr.edu/users/ercal/284/mypgm", "mypgm", "abc", "xyz", 0);
            printf("Now I'm back");
      }
     .
     .
      ......
```

A: (1) printf("Now I'm back") will be overlaid by the "execl" function
call and will not get executed if "execl" is successful.

(2) _exit(1) should follow printf("Now I'am back").
Without it, child process will not terminate properly when execl() fails.

Q: When a process resumes execution after returning from a fork(), how can it tell if it is the original process or the new one?
A: By the process ID which is returned by the fork() call. If it is equal to 0, it is the new one (child process); if it is a non-zero positive value, it is the original process.

Q: Write the name of the system call for obtaining the process id of a process in UNIX?
A: getpid()

Q: Write the name of the system call for obtaining the process id of a process' parent in UNIX?
A: getppid()

Q: GIVEN: a command line entry like -

```
    a.out XX YY ZZ
```

complete the def. of main() and the printf to print the YY argument

```
    int main(                                              )
    {
       printf("                       ",                                 );
    }
```

A:

```
int main( int argc, char* argv[] )
{
   printf( "%s\n",  argv[2] );
}
```

Q: What is the difference between a blocking function call and a nonblocking call?
A: Blocking call: Upon a call to a blocking function, the calling process sleeps until the blocking function finishes execution.

Nonblocking call: A nonblocking call only returns a boolean value that indicates "success" or "failure", and the calling process will continue its execution.

Q: What are the 3 possible dispositions that a process may specify with respect to an interrupt?
A: 1) ignore signal;
2) run the default signal handler provided by the OS;
3) catch the signal and run the user's signal handler.

Q: What function is used from within a process to send an interrupt

```
   (signal) to a process?
A: kill().

Q: True or False
   kill() function, when called from within a process, always causes the
   target process to be killed (terminated)
A: FALSE. It is used to send an interrupt (signal) to a process.


Q: Explain why you would expect to find a different frequency of context
   switches in a purely batch system compared to an interactive system.
A: In an interactive system, quick response for interruption is needed, I/O
exchange is frequent, so the context switch is very frequent. In a batch
system, quick response for interruption is not needed. I/O is in low
frequency, so the context switch has low frequency.

Q: If the instruction to set the privilege bit is a privileged instruction,
   how does a process in user mode get privileged operations performed?
A: The hardware allows privileged instructions to be executed only  in
monitor mode. If an attempt is made to execute a privileged
instruction in user mode, the hardware does not execute the
instruction, but rather treats the instruction as illegal and traps to
the OS and automatically switches to monitor mode. When a user program
does a system call, the switch to the monitor mode is done
automatically by the OS and the priviledged instructions that are
needed by the system routine can be executed.

Q: What is the difference between a pre-emptive scheduler and a time-sliced one?
A: For a pre-emptive scheduler, process with highest priority runs first.
Time slice is CPU based. It divides CPU time into equal time slots and
each process gets one time slot to run.

Q: Using signal(...), show the program parts necessary to cause a
   process to print a message each time the keyboard operator presses ^C.
   The message will be printed N times and then the program will
   terminate.(assume that N is hard coded).

A:

#include <signal.h>
#include <stdio.h>
int N=5 ;   /* global counter N is initialized to some arbitrary number */
void onsigint( int sig )  /* signal handler */
{  signal( sig, onsigint );
   if( N-- > 1 ) printf( "You can press Ctrl-C %d times w/o termination.\n",N );
   else{ signal( sig, SIG_DFL );  /* set to default sig handler */
         printf( "Press Ctrl-C to terminate.\n");   }
}
int main(){
 if(N > 0) signal( SIGINT, onsigint ); /* signal handler is onsigint */
 printf("You have %d chances to press Ctrl-C w/o termination.\n", N);
 for(;;);  /* loop forever */
 return (0);
}


Q: Many modern O.S.s use a microkernel design.  What does that mean?  What
   adjective is applied to 'old' O.S.s?
A: Microkernel removes all nonessential components from the kernel, and
implements them as system and user-level programs. The result is a smaller
```

kernel. The microkernel provides a communication facility between the client program and the various services that are also running in user space. Old OSs had Monolithic kernels which are large kernels containing virtually the complete operating system, including scheduling, file system, device drivers, and memory management. All the functional components of the kernel have access to all of its internal data structures and routines. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space.

Q: True False ? UNIX kernel is designed as a microkernel
A: FALSE. UNIX is a monolithic kernel, meaning that the process, memory, device, and file managers are all implemented in a single software module.

Q: True False ? LINUX is designed as a monolithic kernel.
A: TRUE. But, with a new twist for expanding OS functionality: LINUX supports dynamically installable modules. A module can be compiled and installed on a running version of the kernel. This is accomplished by providing system calls to install/remove modules.

Q: What are the important differences between a unix fork() and pthread_create()?
A: Fork(): child process has the separate data space with parent process,
but they have the same code space.
pthread_create(): threads share data space, code space, and os resources,
but they have the unique thread ID, register state,
stack and priority, PC counter.

Q: Name at least two important differences between a Solaris thread created through a thr_create() call and a child process created through a fork() call.
A:
1) Threads share data space and file descriptors with the other threads and the parent process while forked process doesn't
2) a forked process has a seperate and unique PID and hence a process control block while a thread uses the parent's PID and its process control block.
3) it takes less time to create a new thread, less time to switch between two threads within the same process, less time to terminate a thread
4) Thread Library provides more control over the execution of concurrent threads through system calls such as thr_yield(), thr_suspend(), thr_kill(), and thr_continue()

Q: What is the function prototype for Solaris call thr_create()?
A:

```
int thr_create( void *stack_base, size_t stack_size,
                void *( *start_func )( void * ), void  *arg, long flags,
                thread_t *new_thread_ID)
```

Q: What is the function prototype for the POSIX pthread_create() call?
A: int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *
(*start_routine)(void *), void * arg)

Q: Given

```
 int X[10];
 some appropriate function MyFun
```

```
    thread_t Tid;
  Write the call to thr_create(), which executes MyFun, which accepts the array
  X as an argument
A: thr_create(NULL, NULL, MyFun, X, 0, &Tid)

Q: Answer the above question using POSIX call pthread_create()
A: pthread_t Tid; ...other decl. are the same....
   pthread_create(&Tid, NULL, MyFun, X)

Q: What is the required function prototype for the function MyFun()?
A: void* MyFun( void* X )

Q: Write 1 line of code for MyFun that prints the 3rd integer in the array
   passed to it.
A: printf( "%d", *(int*)( X + 2 ) );

Q: True  False  A thread may share data space with its parent.
A: TRUE

Q: True  False  A thread may share code space with its parent.
A: TRUE

Q: True  False  A thread may share a program counter with its parent.
A: FALSE

Q: True False  A thread may share a File Descriptor Table with its parent
A: TRUE

Q: Assume that you have globally defined
    struct Shared
    { char Name[10];
       int Value;
    } S1, S2;

    thread_t tid;
    void * RtnValue;
    void * FooBar(void * arg);   // For this problem, arg will point to an
instance of struct Shared

Show the correct way to:

  1. Store "THREAD 1" in S1's Name field:
       o _____________________________________________
  2. Pass S1 in the following call:
       o thr_create(NULL,NULL,FooBar,_____________________,0,&tid);
  3. From within the function FooBar, print the Name that was passed inside
     the struct:
       o printf("The Name Received = %s\n",
         ____________________________________________;
  4. store 2 times the thread id into the Value field of arg:
       o  _____________________ = _____________________________
  5. return the struct via a thr_exit:
       o thr_exit(  ___________________________________ );
  6. Have main retrieve the returned value:  Use the variable RtnValue since
     you don't know which thread has exit-ed.
       o thr_join(0,&tid, _____________________);
  7. Have main print which thread exit-ed and the integer returned:
       o printf("Thread %d exited with value = %d\n",
         ____________________, _____________________);
```

```
A:
  1. strcpy( S1.Name, "THREAD 1" );
  2. Thr_create(NULL,NULL,FooBar,&S1,0,&tid);
  3. printf("The Name Received = %s\n", ((struct Shared*)arg)->Name );
  4. ((struct Shared*)arg)->Value = 2 * thr_self();
  5. thr_exit( arg );
  6. thr_join( 0, &tid, &RtnValue );
  7. printf( "Thread %d exited with value = %d\n", tid,
           ((struct Shared*)RtnValue)->Value );

Q: Answer the above question using POSIX threads
A:
  1. strcpy( S1.Name, "THREAD 1" );
  2. pthread_create(&tid,NULL,FooBar, (void*)S1);
  3. printf("The Name Received = %s\n", ((struct Shared*)arg)->Name );
  4. ((struct Shared*)arg)->Value = 2 * pthread_self();
  5. pthread_exit( arg );
  6-7. There is no way to wait for "any" thread in Pthreads (a flaw?)

Q: True or False
   In the specification of pthread_join(), there is no way to wait for
   "any" thread (i.e. the call should specify a particular thread_id
   to join)
A: TRUE

Q: How does a thread acquire RAM that is NOT shared with other threads in the task?
A: Define local variables and use them only in this thread scope.

Q: What is the effect of executing

  1. thr_yield()?
  2. thr_suspend(...)?
  3. thr_continue(...)?

A: 1. thr_yield(): yield to another thread
   2. thr_suspend(): suspend thread execution
      A suspended thread will not be awakened by  a  signal.   The
      signal  stays  pending  until the execution of the thread is
      resumed by thr_continue()
   3. thr_continue(): continue thread execution

Q: The word 'mutex' is short for _____________________________________.
A: mutual exclusion

Q: What is an atomic operation?
A: An operation that can not have it's execution suspended until
   it is fully completed. Generally, it is a single operation that can
   not be interrupted or divided into smaller operations.

Q: True False: Atomic operations are needed in order to implement mutex locks.
A: True

Q: What does it mean to say that "mutex_lock(...) is a blocking call"?
A: A successful call for a mutex lock by mutex_lock() will cause another
thread which is also trying to lock the same mutex to block until the owner
thread unlocks it by mutex_unlock().

Q: What would you expect to see (what instructions) surrounding the 'critical
   section' code?
A: mutex_t mp;
```

```
   mutex_lock( &mp );
     <CRITICAL SECTION>
   mutex_unlock( &mp );

Q: Answer above question using Pthreads
A: pthread_mutex_t mp;
   pthread_mutex_lock( &mp );
     <CRITICAL SECTION>
   pthread_mutex_unlock( &mp );
}


Q: True or False. If mutual exclusion is not enforced in accessing a
   critical section, a deadlock is guaranteed to occur.
A: False.

Q: What does it mean to say "rand() call is not MT-safe"?
A: This means that the behavior of the function rand() is not stable when two
threads try to use it at the same time. To get around this problem, we use
MT-safe interfaces (such as rand_r()).  However, rand_r() doesn't exist
on some multithreaded operating systems (e.g. LINUX), therefore, you
need to execute such calls inside a Critical Section to make sure that
only one thread can call the function at any one time.

 Example:
    pthead_mutex_t rand_mutex;

    pthread_mutex_lock( &rand_mutex );
            int a = rand() % 5;
    pthread_mutex_unlock( &rand_mutex );


Q: What is a spin-lock?
A: When a process is in its critical section, any other process that tries
   to enter its critical section must loop continuously in the entry code of
   critical section until the first one gets out.

Q: What is an alternative to spin-locking?
A: Put the process in a wait queue, so it doesn't waste CPU cycles and
allow it to sleep until the block is released.

Q: Explain under what circumstances a spin lock might be more efficient than
   the alternative.
A: If you have more processor power than you need (e.g. in
multiprocessor systems). Spin lock do not require context switch when
a process must wait on a lock, and context switch may take considerable time.
Thus when the locks are expected to be held for short time, a spin lock might
be efficient.

--------------------------------------------------------------------------------
The code below is written to provide one possible solution to the READERS/WRITERS pr
--------------------------------------------------------------------------------

int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
semaphore readBlock=1, writeBlock=1,

reader() {
  while(TRUE) {
    <other computing>;
```

```
      P(readBlock);
        P(mutex1);
          readCount++;
          if(readCount == 1)
            P(writeBlock);
        V(mutex1);
      V(readBlock);

      access(resource);
    P(mutex1);
      readCount--;
      if(readCount == 0)
        V(writeBlock);
    V(mutex1);
  }
}

writer() {
  while(TRUE) {
    <other computing>;
    P(mutex2);
      writeCount++;
      if(writeCount == 1)
        P(readBlock);
    V(mutex2);
    P(writeBlock);
      access(resource);
    V(writeBlock);
    P(mutex2)
      writeCount--;
      if(writeCount == 0)
        V(readBlock);
    V(mutex2);
  }
}
```

Answer the following questions based on the code given above.

Q: Explain the role of each semaphore used in this code.

A:

mutex1: to provide mutually exclusive access to the shared variable "readcount" by multiple readers.

mutex2: to provide mutually exclusive access to the shared variable "writecount" by multiple writers.

readBlock: if a reader arrives while a writer is inside the critical section(CS) and other writers are waiting, this semaphore will cause that reader to get blocked until the last writer leaves the CS.

writeBlock: This semaphore causes any writer to get blocked if there is one writer or reader(s) currently accessing the CS.

Q: Does this code provide a correct and fair solution for the readers/writers problem?

A: Even though this code is correct, it does not provide a fair solution. With the way it is written, there is a possibility that writers may starve readers if they arrive one after another. In other words, readers may never get a chance to enter the critical section (i.e. access the shared resource) if the writers don't take

```
   a break.

Q: Could you suggest a solution which is both correct and fair?
A: Yes.
   Introduce a new semaphore called "writePending" which is
   shared by both readers and writers. It will force the blocked readers
   and writers to enter a waiting queue in the order that they have
   arrived. The associated P and V calls are added to the original
   code as follows:

int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
semaphore readBlock=1, writeBlock=1;
semaphore writePending=1;

reader() {
  while(TRUE) {
    <other computing>;
    P(writePending);
      P(readBlock);
        P(mutex1);
          .......
        V(mutex1);
      V(readBlock);
    V(writePending);
      access(resource);
    .........
  }
}

writer() {
  while(TRUE) {
    <other computing>;
   P(writePending);
    P(mutex2);
      ........
    V(mutex2);
    P(writeBlock);
      access(resource);
    V(writeBlock);
   V(writePending);

    ...........
  }
}


Q: True or False
   In Readers/Writers problem, it is possible to write code which is
   functionally correct but may lead to the starvation of
   writers. However, it is impossible to write the code such that
   readers may starve instead of writers.
A: False. It is possible to write such code.


Q: Explain under what circumstances a spin lock might be less efficient than
   the alternative.
A: In uniprocessor systems, processes can use CPU one at a time.
   If a process is "busy waiting", no other process could
   execute. Thus a spin lock might be less efficient.
```

Q: In dining philosophers problem, why examining and acquiring a fork is done inside a critical section? Explain by giving an example what may go wrong if critical section is not used.
A: Each fork is shared by two neighboring philosophers. If it is not handled inside a critical section, both philosophers may think that they acquired the same fork and start eating using the same fork. This situation leads to an incorrect simulation.

Q: In dining philosophers problem, which senario may lead to a deadlock situation?
A: If the simulation code is written in such a way that each philosopher first acquires the right fork and then attempts to acquire the left fork, a deadlock may occur. Because it is possible that philosophers may enter a circular wait situation in which each one holds the fork on the right and tries to get the fork on the left (which will never happen).