Amy Harlan

Midterm Review

1. F, T, T, T, F
2. Mode, Owners, Time Stamps, Size, Reference Count, Block Index
3. A deadlock happens when one or more threads will have to block forever (or until process is terminated) because they have to wait for a resource that will never be available. Ex. A cycle in a graph. Starvation, ex. read/write locks, happens when a thread may need to wait for a long time before a resource becomes available.
4. Advantages of Threads: fast thread creation, fast context switch, fast communication across threads

Disadvantages of Threads/Advantages of Processes: threads are less robust than processes, threads have more synchronization problems than processes

1. /etc/passwd – User information

/etc/groups – Group information

/etc/inetd.conf – Configuration of Internet Services (deamons)

1. A. T2d, T2e, T3g, T1a, deadlock!

B.

C.

|  |  |  |
| --- | --- | --- |
| T1:   1. mutex\_lock(&m1); 2. mutex\_lock(&m3); 3. mutex\_lock(&m5); | T2:   1. mutex\_lock(&m2); 2. mutex\_lock(&m4); 3. mutex\_lock(&m5); | T3:   1. mutex\_lock(&m1); 2. mutex\_lock(&m3); 3. mutex\_lock(&m4); |

1. There could be a context switch between array[count]=value; and count = count + 1; causing what was just put into array[count] to be written over.

**#include <pthread.h>**

**#define MAXCOUNT;  
int count = 0;  
int array[MAXCOUNT];  
pthread\_mutex\_t mutex;**

**int addToArray(int value)  
{  
  if (count == MAXCOUNT) {  
     return -1;  
  }  
 pthread\_mutex\_lock(&mutex);  
  array[count]=value;  
  count = count + 1;  
  return count-1;**

**pthread\_mutex\_unlock(&mutex);**

**}**

1. The user program calls the write(file\_descriptor, buffer, nbytes) system call to write to disk.

The write wrapper in libc generates a software interrupt for the system call and switches to kernel mode.

The OS in the interrupt handler checks the arguments. It verifies that file\_descriptor is a file descriptor for a file opened in write mode. And also that [buffer, buffer+ nbytes-1] is a valid memory range. If any of the checks fail write return -1 and sets errno to the error value.

The OS tells the hard drive to write the buffer in [buffer, buffer+ nbytes] to disk to the file specified by file\_descriptor.

The OS puts the current process in wait state until the disk operation is complete. Meanwhile, the OS switches to another process.

The Disk completes the write operation and generates an interrupt switching back to user mode.

The interrupt handler puts the process calling write into ready state so this process will be scheduled by the OS in the next chance.

1. while [ “1”]

do

sleep 60

cd $1

ls –lR > $2/ls-lR.new

cd $2

if ! diff ls-lR.new ls-lR.last; then

echo “Hello $USER” > tmp-message

echo >> tmp-message

echo “A file has been modified.” >> tmp-message

/usr/bin/mailx –s “change” $USER < tmp-message

echo “Message Sent”

fi

done

1. int main( int argc, char \*\* argv) {

if (argc < 3) {

fprintf(stderr, "usage: lsgrep arg1 arg2\n");

exit(1);

}

// Strategy: parent does the

// redirection before fork()

//save stdin/stdout

int tempin = dup(0);

int tempout = dup(1);

//create pipe

int fdpipe[2];

pipe(fdpipe);

//redirect stdout for "ls“

dup2(fdpipe[0],1);

close(fdpipe[0]);

// fork for "ls”

int ret= fork();

if(ret==0) {

// close file descriptors

// as soon as are not

// needed

close(fdpipe[1]);

char \* args[3];

args[0]="ls";

args[1]=“-al";

args[2]=NULL;

execvp(args[0], args);  
 // error in execvp

perror("execvp");

\_exit(1);

}

//redirection for "grep“

//redirect stdin

dup2(fdpipe[1], 0);

close(fdpipe[1]);

//create outfile

int fd=open(argv[2], O\_WRONLY|O\_CREAT|O\_TRUNC, 0600);

if (fd < 0){

perror("open");

exit(1);

}  
 //redirect stdout

dup2(fd,1);

close(fd);

// fork for “grep”

ret= fork();

if(ret==0) {

char \* args[3];

args[0]=“grep";

args[1]=argv[1];

args[2]=NULL;

execvp(args[0], args);  
 // error in execvp

perror("execvp");

\_exit(1);

}

// Restore stdin/stdout

dup2(tempin,0);

dup2(tempout,1);

// Parent waits for grep process

waitpid(ret,NULL);  
}

1. #include <pthread.h>

class SynchronizedStackSemaphores {  
 // Add your member variables here  
 int top;

int tail;  
 int \* stack;

mutex\_t \_mutex;

sema\_t \_emptySem;

sema\_t \_fullSem;  
  
  
  
 public:  
 SynchronizedStackSemaphores (int maxStackSize );  
 void push(int val );  
 int pop();  
};  
  
SynchronizedStackSemaphores::SynchronizedStackSemaphores( maxStackSize ) {

top = 0;

tail = 0;

stack = malloc(maxStackSize \* sizeof(int));

pthread\_mutex\_init(&\_mutex, NULL, NULL);

sema\_init(&\_emptySem, 0, USYNC\_THREAD, NULL);

sema\_init(&\_fullSem, maxStackSize, USYNC\_THREAD, NULL);  
}  
  
void SynchronizedStackSemaphores::push(int val) {

sema\_wait(&\_fullSem);

mutex\_lock(\_mutex);

\*(stack+tail) = val;

tail = (tail+1) %maxStackSize;

mutex\_unlock(\_mutex);

sema\_post(\_emptySem);  
}  
  
int SynchronizedStackSemaphores::pop(){

sema\_wait(&\_emptySem);

mutex\_lock(\_mutex);

int val = \*(stack+top);

top = (top+1) %maxStackSize;

mutex\_unlock(\_mutex);

sema\_post(\_fullSem);

return val;  
}

12. class SynchronizedStackCondVariables {  
 // Add your member variables here  
 int top;

int tail;

int [] stack;

mutex\_t mutex;  
  
 SynchronizedStackCondVariables(int maxStackSize ) {

top = 0;

tail = 0;

stack = new int[maxStackSize];

pthread\_mutex\_init(&\_mutex, NULL, NULL);  
 }

void push(int val) {

if(tail < maxStackSize){

mutex\_lock(\_mutex);

stack[tail] = val;

tail = (tail+1) %maxStackSize;

mutex\_unlock(\_mutex);  
}

}  
  
int pop(){

if(tail > 0){

mutex\_lock(\_mutex);

int val = stack[top];

top = (top+1) %maxStackSize;

mutex\_ unlock(\_mutex);

return val;

}  
}

}