Answers to mid-term

1. producer() {

while(TRUE) {

b = produceByte(...);

P(mutex);

putByte(b);

V(mutex);

V(byteCount);

}

}

consumer() {

while(TRUE) {

P(byteCount);

P(mutex);

takeByte(b);

V(mutex);

}

}

semaphore mutex = 1, byteCount = 0;

fork(producer, 0);

fork(consumer, 0);

(2) type resource=monitor

var P: array[0..2] of boolean

X:condition

procedure acquire (id: integer, printer-id:integer)

begin

ii P[0] and P[1] and P[2] then X.wait(id)

if not P[0] then printer.id=0

else if no P[1] then printer.id=1

else printer.id=2

P[printer.id]=true

end

procedure release (printer.id:integer)

begin

P[printer.id]=false

X.signal

end

begin

P[0]=P[1]=P[2]=false

End

(3) Suppose the page replacement algorithm is LRU, the page frame allocation is 3, and the loop is loaded in 2 pages, but every instruction references 2 data pages, e.g., the code has the form:

for (...) {

y = f(x);

---- page boundary ----

y = g(x);

}

Since the 2 data pages are referenced on every instruction, these pages will not be unloaded, leaving only one page for the instructions. When the process executes the top half of the loop, the page containing the code for the top half of the loop, including y = f(x), is loaded, but when the code for the bottom half of the loop is executed, including y = g(x), the first code page is unloaded and the second is loaded. That is, each time through the loop there will be a missing page fault for the code pages.

1. A user space implementation of threads relies on a single OS process. The process schedules its threads by coroutining. The OS has no knowledge of the process activity, meaning that it cannot infer thread semantics based on its process-oriented view of the computation. So, if one of the threads blocks, the from the OS perspective, the process has blocked; therefore it will not schedule the process to run (and hence the collection of thread coroutines to execute) until the single thread becomes unblocked.

(\*\*\*They don’t have to mention the work co-routines here\*\*\*)

(5) User name – process Stack bottom – thread Resources blocking me – thread Primary memory allocated to me – process Files allocated to me – process Execution state – thread

(6) For a one-level page table, there are 2\*32 */* 2\*12 or 1M pages needed. Thus the

page table must have 1M entries. For two-level paging, the main page table

has 1K entries, each of which points to a second page table. Only two of these

are used. Thus in total only three page table entries are needed, one in the

top-level table and one in each of the lower-level tables.

(\*\*\*Comes from Tanenbaum\*\*\*)

(7) The *R* bit is never needed in the TLB. The mere presence of a page there

means the page has been referenced; otherwise it would not be there. Thus the

bit is completely redundant. When the entry is written back to memory, how-

ever, the *R* bit in the memory page table is set.