5.8) This algorithm follows the 3 requirements of the critical section problem. It is mutually exclusive because in order for both process to enter their critical section flag[i] would have to be both true and false which isn’t possible. The algorithm also follows the progress requirement since the algorithm switches control to the other process once the first process finishes it’s critical section and the first process cant select it’s self. It also follows the bounded waiting requirement as the second process is automatically selected after the execution of the first is complete and vice versa.

5.10) Implementing synchronization primitives by disabling interrupts is not appropriate in most systems, especially not in a single-processor system because this would essentially allow the process using the primitive to take full control of the cpu and prevent other processes from getting cpu time.

5.14) Since compare and swap only provides us with mutual exclusion and not bounded waiting we need to change the algorithm a little to get it have bounded waiting. The easiest solution is to modify the test and set algorithm, which does provide bounded waiting, to call compare\_and\_swap instead.

So it is essentially test and set except it calls compare and swap, below is a modification of the test and set algorithm provided by the book.

do *{*

waiting[i] = true;

key = true;

while (waiting[i] && key)

key = compare\_and\_swap(&lock , &key);

waiting[i] = false;

/\* critical section \*/

j = (i + 1) % n;

while ((j != i) && !waiting[j])

j = (j + 1) % n;

if (j == i)

lock = false;

else

waiting[j] = false;

/\* remainder section \*/

*}* while (true);

5.17)

The lock is to be held for a short duration. – In this scenario spin lock is better since it is only being held for a short duration so its better to just have thread be busy waiting then have the overhead of context switches that mutex has.

The lock is to be held for a long duration. – Mutex is better in this scenario because the over head of the context switch is lower than the overhead of busy waiting if the lock is held for a long time.

A thread may be put to sleep while holding the lock. – In most implementations of spinlock, the lock is relinquished when a thread is put to sleep so mutex would be the better option in this case.

5.20) a. This code has race condition on the variable number\_of\_processes. This variable is shared between multiple processes and can be accessed by them at the same time. One example of an issue that could arise is that the allocate function doesn’t increment the amount of processes until the resources are already allocated. This means that number\_of\_processes could be at MAX\_PROCESSES -1 and one process could be busy allocating resources meanwhile another process is checking if number\_of\_process == MAX\_PROCESSES since the variable hasn’t been incremented yet it will be allowed to allocate resources even though there might not be enough space.

b) using the mutex lock calls acquire() and release() we can modify the code to get rid of the race condition. The lock only needs to be acquired before a shared variable is accessed and then the lock can be released. So the new code should look like this:

#define MAX PROCESSES 255

int number of processes = 0;

/\* the implementation of fork() calls this function \*/

int allocate process() *{*

int new pid;

acquire() ;

if (number of processes == MAX PROCESSES)

return -1;

else *{*

/\* allocate necessary process resources \*/

++number of processes;

release();

return new pid;

*}*

*}*

/\* the implementation of exit() calls this function \*/

void release process() *{*

/\* release process resources \*/

acquire();

--number of processes;

release();

*}*

c) Making it an atomic type wouldn’t solve the race condition as another thread would still be able to pass the size check while another thread is busy allocating resources and hasn’t yet incremented the number yet.

5.21) Essentially we could use P and V semaphores to accomplish this. Have a variable keep track of the sockets available, which we’ll call availSockets. If P(availSockets) is called and passes availSockets will be decremented and a new socket will be allocated. If someone disconnects V(availSockets) will be called and availSockets will be incremented.

7.16)

a. Increasing the available resources is always safe and won’t create a deadlock under in condition.

b. This is only safe when the new amount of resources is still sufficient for the max amount of resources required by the processes.

c. Increasing max is still safe as long as max is still less than the available resources.

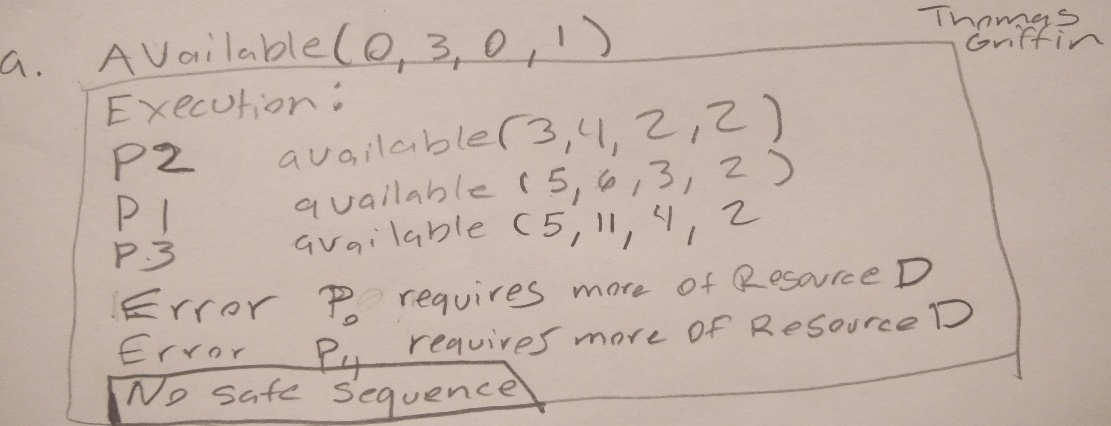
d. Decreasing max is always safe.

e. Increasing the amount of programs is safe as long as the programs don’t require more resources than the available.

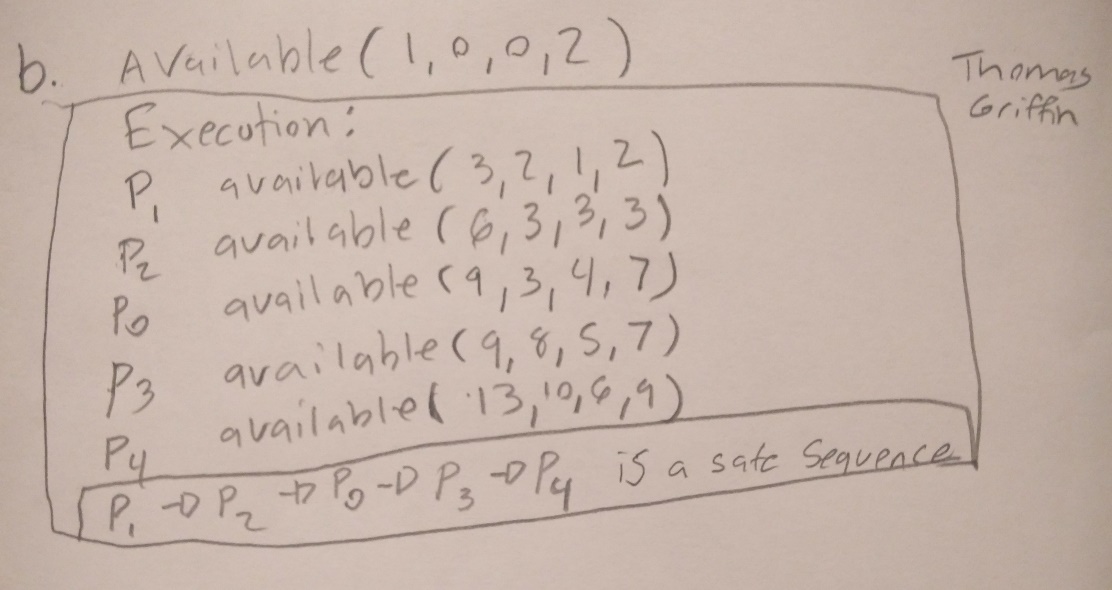
f. Decreasing the amount of programs should be always safe.

7.22)

a) not safe

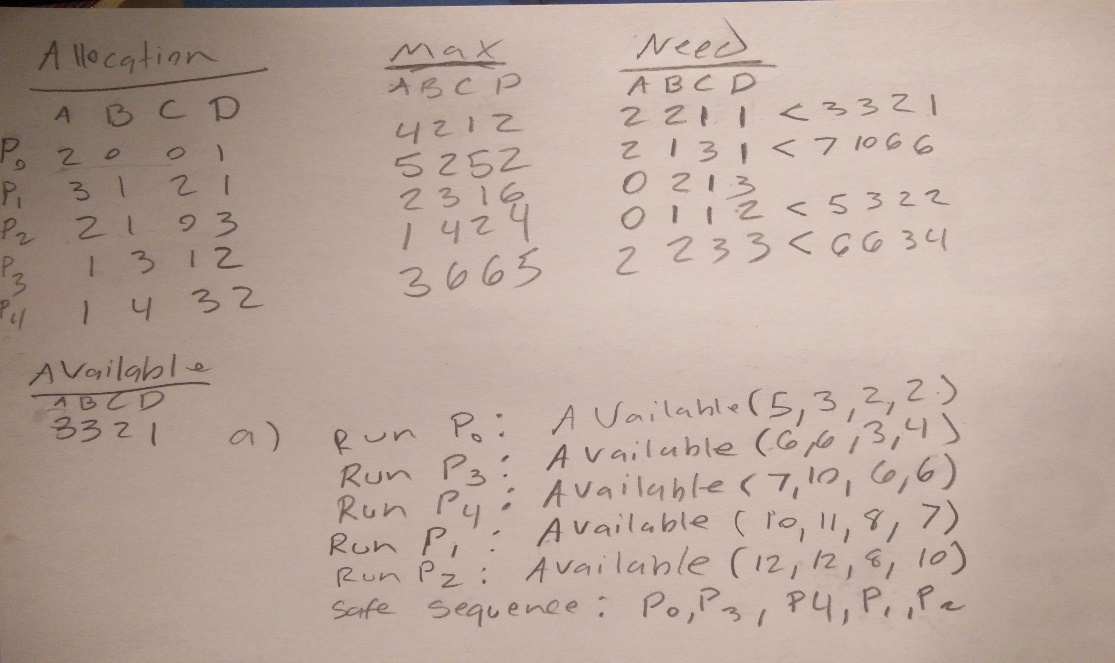


b) p1 , p2, p0, p3, p4 is one of the safe sequences.

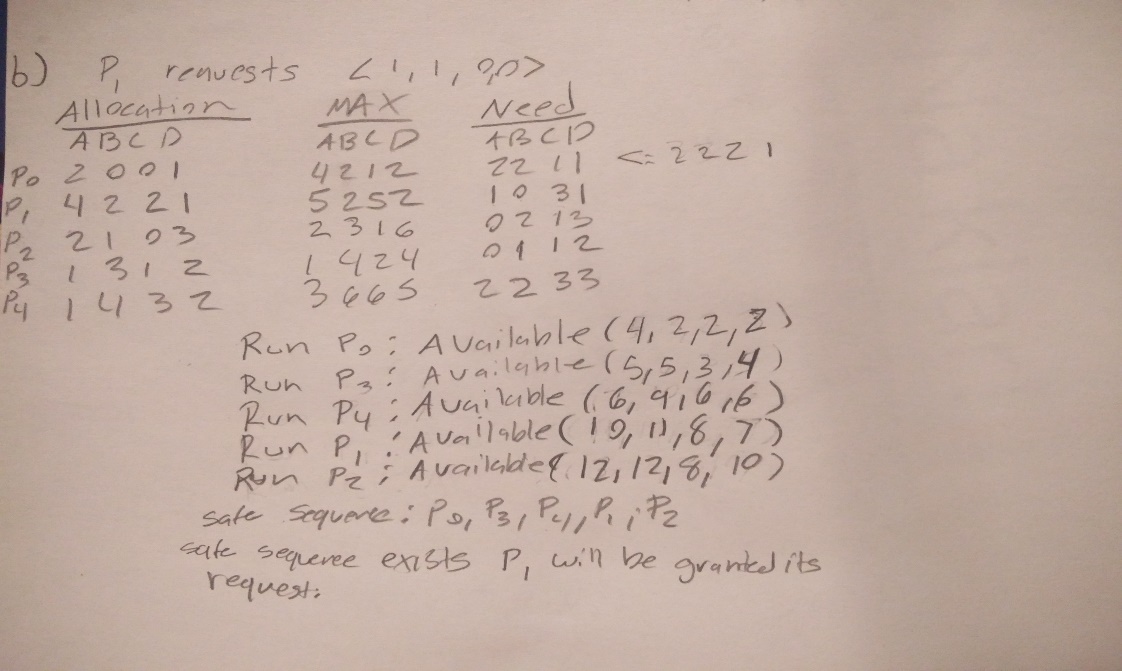


7.23)

a.safe sequence: p0, p3, p4, p1, p2



b. safe sequence: p0, p3,p4,p1,p2, since a safe sequence with the request exists, the request will be granted.



c. No safe state exists for the request so the request will not be granted.

