Quinn Roemer

Dr. Yuan Cheng

CSC 139

30 November 2020

Exercise 1. Consider the code example for allocating and releasing processes shown below:

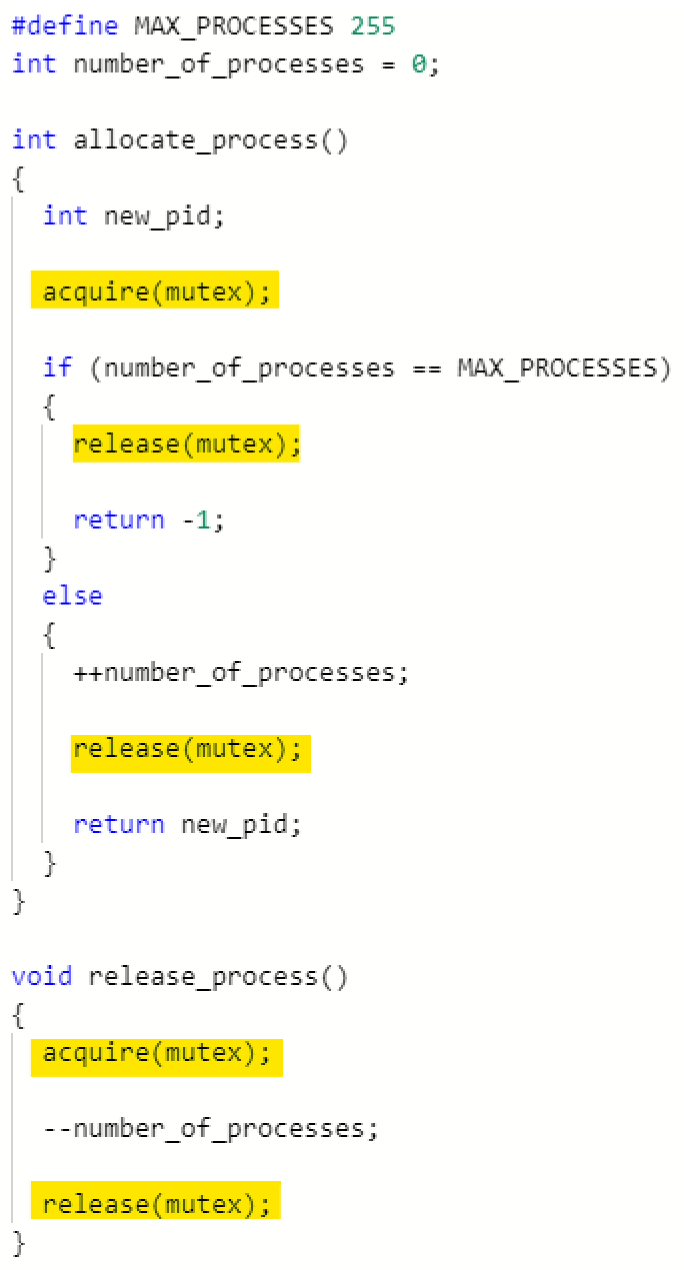
```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
    int new_pid;

    if (number_of_processes == MAX_PROCESSES)
        return -1;
    else {
        /* allocate necessary process resources */
        ++number_of_processes;

        return new_pid;
    }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

1. Identify the race condition(s):

    - A race condition occurs when either function decrements or increments *number_of_processes.* In addition, in the if statement of the first function, when comparing the current value of *number_of_processes* with *MAX_PROCESSES* another race condition occurs. If the value in *number_of_processes* is being updated while reading a race condition would result.

2. Assume you have a mutex lock named *mutex* with the operations *acquire()* and *release()*. Indicate where the locking needs to be placed to prevent the race conditions:

   - *acquire()* would need to be placed before the if statement in *allocate_process()*. In *release_process()*, *acquire()* would need to be used before decrementing *number_of_processes.*
   - *release()* would have to be used inside the if statement in *allocate_process()* before returning. Or inside the else statement before returning. In *release_process()*, *release()* would need to be called after decrementing *num_of_processes.*



```c
#define MAX_PROCESSES 255
int number_of_processes = 0;

int allocate_process()
{
    int new_pid;

    acquire(mutex);

    if (number_of_processes == MAX_PROCESSES)
    {
        release(mutex);

        return -1;
    }
    else
    {
        ++number_of_processes;

        release(mutex);

        return new_pid;
    }
}

void release_process()
{
    acquire(mutex);

    --number_of_processes;

    release(mutex);
}
```

Exercise 2. Consider the following snapshot of a system: Answer the following questions using banker's algorithm:

	Allocation				Max			
	A	B	C	D	A	B	C	D
T_0	3	1	4	1	6	4	7	3
T_1	2	1	0	2	4	2	3	2
T_2	2	4	1	3	2	5	3	3
T_3	4	1	1	0	6	3	3	2
T_4	2	2	2	1	5	6	7	5

Available			
A	B	C	D
2	2	2	4

1. Illustrate that the system in in a safe state by demonstrating an order in which the threads may complete.

	Allocated					Need			
	A	B	C	D		A	B	C	D
✓ T_0	3	1	4	1	T_0	3	3	3	2
✓ T_1	2	1	0	2	T_1	2	1	3	0
✓ T_2	2	4	1	3	T_2	0	1	2	0
✓ T_3	4	1	1	0	T_3	2	2	2	2
✓ T_4	2	2	2	1	T_4	3	4	5	4

First Run — Safe Sequence $\langle T_2, T_3, T_0, T_1, T_4 \rangle$

$Need_0$	3 3 3 2	Work	2 2 2 4	Larger
$Need_1$	2 1 3 0	Work	2 2 2 4	Larger
$Need_2$	0 1 2 0	Work	2 2 2 4	New Work 4 6 3 7
$Need_3$	2 2 2 2	Work	4 6 3 7	New Work 8 7 4 7
$Need_4$	3 4 5 4	Work	8 7 4 7	Larger

Second Run

$Need_0$	3 3 3 2	Work	8 7 4 7	New Work 11 8 8 8
$Need_1$	2 1 3 0	Work	11 8 8 8	New Work 13 9 8 10
$Need_4$	3 4 5 4	Work	13 9 8 10	New Work 15 11 10 11

- Since the safe sequence (T2, T3, T0, T1, T4) exists the system is in a safe state.

2. If a request from thread T_4 arrives for (2, 2, 2, 4), can the request be granted immediately?

T_4 requests $\langle 2,2,2,4 \rangle$

Allocated

	A	B	C	D
T_0	3	1	4	1
T_1	2	1	0	2
T_2	2	4	1	3
T_3	4	1	1	0
T_4	2	2	2	1

Need

	A	B	C	D
T_0	3	3	3	2
T_1	2	1	3	0
T_2	0	1	2	0
T_3	2	2	2	2
T_4	3	4	5	4

1. Request $\leq$ Need
2. Request $\leq$ Available
3. New Allocated, Need, & Available

Allocated

	A	B	C	D
T_0	3	1	4	1
T_1	2	1	0	2
T_2	2	4	1	3
T_3	4	1	1	0
T_4	4	4	4	5

Need

	A	B	C	D
T_0	3	3	3	2
T_1	2	1	3	0
T_2	0	1	2	0
T_3	2	2	2	2
T_4	1	2	3	0

Available

A	B	C	D
0	0	0	0

Run Safety Algorithm

$Need_0$	3 3 3 2	Work	0 0 0 0	Larger
$Need_1$	2 1 3 0	Work	0 0 0 0	Larger
$Need_2$	0 1 2 0	Work	0 0 0 0	Larger
$Need_3$	2 2 2 2	Work	0 0 0 0	Larger
$Need_4$	1 2 3 0	Work	0 0 0 0	Larger

The request cannot be granted immeditatelly as it would put the system in an unsafe state.

- The request **cannot** be granted immediately as it would put the system in an unsafe state.

3. If a request from thread T_2 arrives for (0, 1, 1, 0), can the request be granted immediately?

T_2 requests $\langle 0, 1, 1, 0 \rangle$

	Allocated				Need			
	A	B	C	D	A	B	C	D
T_0	3	1	4	1	3	3	3	2
T_1	2	1	0	2	2	1	3	0
T_2	2	4	1	3	0	1	2	0
T_3	4	1	1	0	2	2	2	2
T_4	2	2	2	1	3	4	5	4

1. Request ≤ Need
2. Request ≤ Available
3. New Allocated, Need, & Available

	Allocated				Need			
	A	B	C	D	A	B	C	D
T_0	3	1	4	1	3	3	3	2
T_1	2	1	0	2	2	1	3	0
✓ T_2	2	5	2	3	0	0	1	0
T_3	4	1	1	0	2	2	2	2
T_4	2	2	2	1	3	4	5	4

Available			
A	B	C	D
2	1	1	4

Run Safety Algorithm

Safe Sequence $\langle T_2, T_3, T_0, T_1, T_4 \rangle$

$Need_0$	3 3 3 2	Work	2 1 1 4	Larger	
$Need_1$	2 1 3 0	Work	2 1 1 4	Larger	
$Need_2$	0 0 1 0	Work	2 1 1 4	New Work	4 6 3 7
$Need_3$	2 2 2 2	Work	4 6 3 7	New Work	8 7 4 7
$Need_4$	3 4 5 4	Work	8 7 4 7	Larger	
$Need_0$	3 3 3 2	Work	8 7 4 7	New Work	11 8 8 8
$Need_1$	2 1 3 0	Work	11 8 8 8	New Work	13 9 8 10
$Need_4$	3 4 5 4	Work	13 9 8 10	New Work	15 11 10 11

This request can be granted immeditally as the system will still be in a safe state since a safe sequence exists.

- The request **can** be granted immediately as a safe sequence exists after granting the request.

4. If a request from thread T_3 arrives for (2, 2, 1, 2), can the request be granted immediately?

T_3 requests $\langle 2,2,1,2 \rangle$

Allocated

	A	B	C	D
T_0	3	1	4	1
T_1	2	1	0	2
T_2	2	4	1	3
T_3	4	1	1	0
T_4	2	2	2	1

Need

	A	B	C	D
T_0	3	3	3	2
T_1	2	1	3	0
T_2	0	1	2	0
T_3	2	2	2	2
T_4	3	4	5	4

① Request ≤ Need

② Request ≤ Available

③ New Allocated, Need, & Available

Allocated

	A	B	C	D
T_0	3	1	4	1
T_1	2	1	0	2
T_2	2	4	1	3
T_3	6	3	2	2
T_4	2	2	2	1

Need

	A	B	C	D
T_0	3	3	3	2
T_1	2	1	3	0
T_2	0	1	2	0
T_3	0	0	1	0
T_4	3	4	5	4

Available

A	B	C	D
0	0	1	2

Run the Safety Algorithm

Safe Sequence $\langle T_3, T_0, T_1, T_2, T_4 \rangle$

$Need_0$	3 3 3 2	Work	0 0 1 2	Larger	
$Need_1$	2 1 3 0	Work	0 0 1 2	Larger	
$Need_2$	0 1 2 0	Work	0 0 1 2	Larger	
$Need_3$	0 0 1 0	Work	0 0 1 2	New Work	6 3 3 4
$Need_4$	3 4 5 4	Work	6 3 3 4	Larger	
$Need_0$	3 3 3 2	Work	6 3 3 4	New Work	9 4 7 5
$Need_1$	2 1 3 0	Work	9 4 7 5	New Work	11 5 7 7
$Need_2$	0 1 2 0	Work	11 5 7 7	New Work	13 9 8 10
$Need_4$	3 4 5 4	Work	13 9 8 10	New Work	15 11 10 11

This request can be granted immeditally as the system will still be in a safe state since a safe sequence exists.

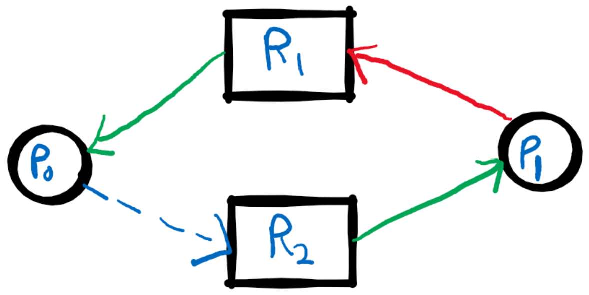
- The request **can** be granted immediately as a safe sequence exists after granting the request.

Exercise 3. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Is this system deadlock-free? Why or why not?

- Yes, the system is deadlock free. Because there exist enough resources that at least a single process will always be able to obtain two resources which will allow it to complete and release those resources.

Exercise 4. Can a system be in a state that is neither deadlocked nor safe? If yes, give an example system.

- Yes, a system can be in such a state when no safe sequence exists. This can be seen in the below resource allocation graph.



Exercise 5. Two processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order deadlock is not possible. However, if B asks for them in the order 3, 2, 1 , then deadlock is possible. With three resources there are 3! or six possible combinations in which each process can request them. What fraction of the combinations is guaranteed to be deadlock free?

- The only combinations that are guaranteed to be deadlock free are those that are identical for each process as this invalidates the circular requirement of a deadlock. This would mean **6/36** or **1/6** of all possible combinations are deadlock free.

Exercise 6. A machine has 48-bit virtual addresses, 32-bit physical addresses, and the page size is 8KB. How many entries are needed for one process's page table?

- 48bit logical address space (2^{48}) with 8KB pages (2^{13}): 2^{48} / 2^{13} = **2^{35} entries**

Exercise 7. A computer with a 32 bit address space uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field (the directory), an 11-bit second-level page table field and an offset. How large are the pages and how many are there in the address space?

9	11	12 (offset)

- The top level page is **2^9** and the second-level page is **2^{11}** this results in a 12 bit offset. The pages used an address space of **2^{20}**.

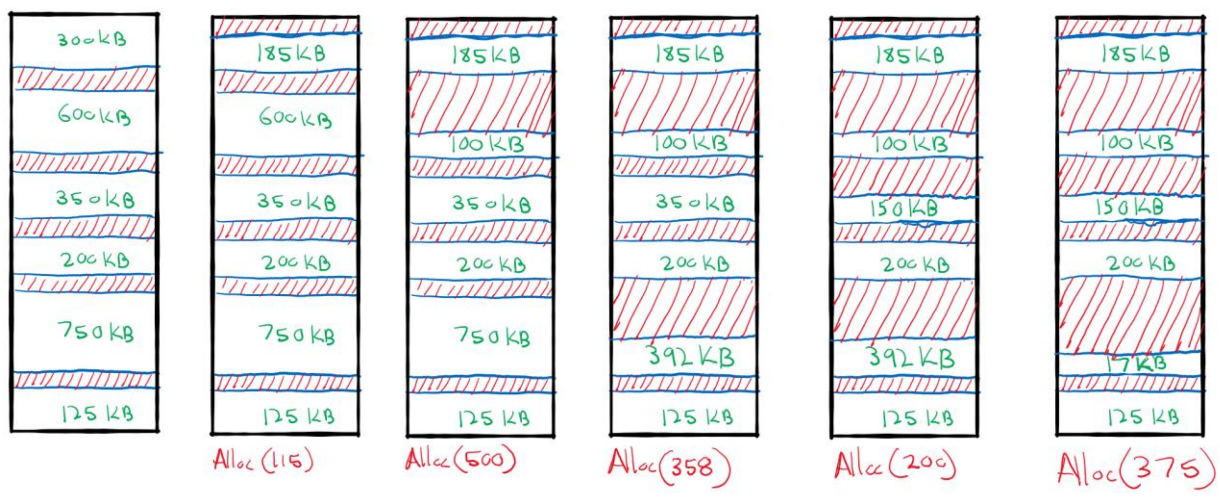
Exercise 8. Under what circumstances do page faults occur? Describe the actions taken by the operation system when a page fault occurs.

- Page faults occur when a process requests a page that is not currently loaded into memory. This often occurs during the first reference to a page by the process. In the case of a page fault the OS first checks if the process is referencing valid memory. If not, the process is aborted. If so, the referenced page must be loaded in memory using some algorithm, the processes page table updated, and the process restarted at the instruction that caused the fault.

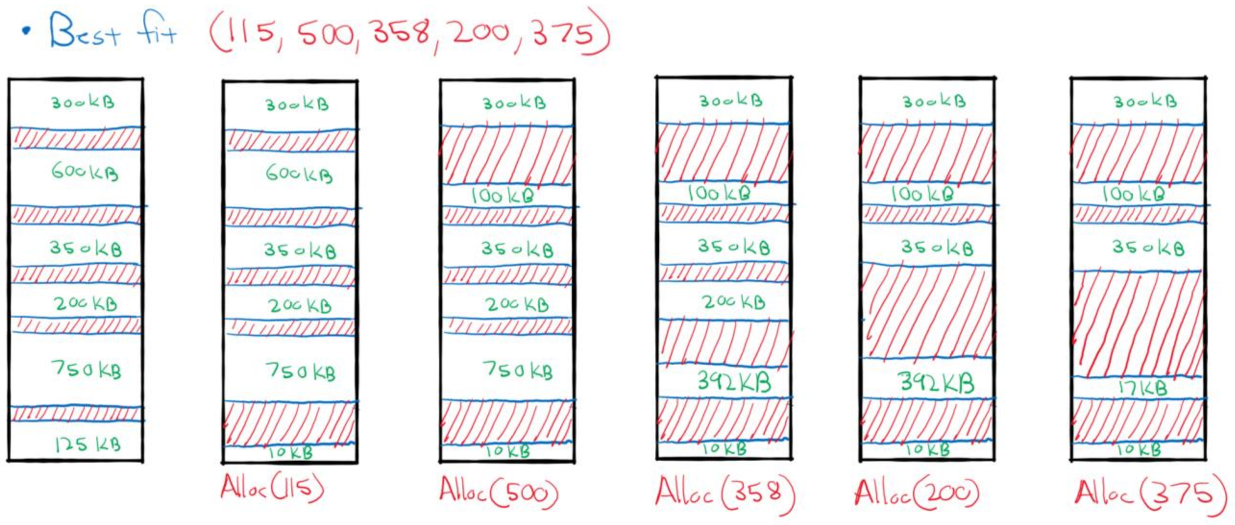
Exercise 9. Given six memory partitions of 300KB, 600KB, 350KB, 200KB, 750KB, and 125KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115KB, 500KB, 358KB, 200KB, and 375KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

- First-fit (**2nd** – Fits all the allocations, but with a large amount of external fragmentation)

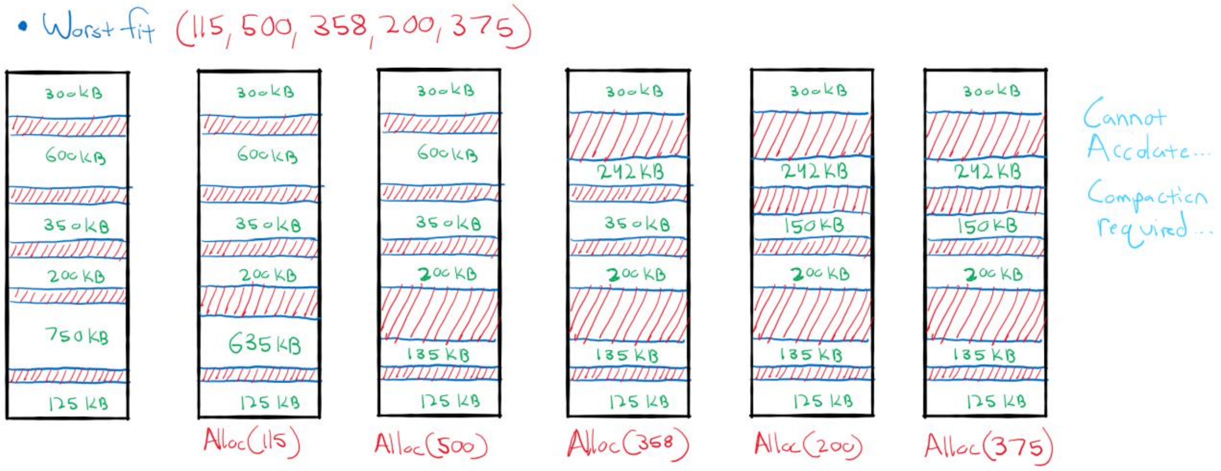
•First Fit (115, 500, 358, 200, 375)



- Best-ft (**1st** – Larger and fewer gaps resulting in less external fragmentation)



- Worst-fit (**3rd** – Failed to fit all the allocations, as a result it is the worst among the three)



Exercise 10. Consider the following page reference string **7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.** Assuming demand paging with three frames, how many page faults would occur for the following replace algorithms? (Show the steps and highlight the page faults).

- LRU Replacement – **18 page faults**

7	7	7	**1**	1	1	**3**	3	3	**7**	7	7	7	**5**	5	5	**2**	2	2	**1**
	2	2	2	2	2	2	**4**	4	4	4	**1**	1	1	**4**	4	4	**3**	3	3
		3	3	3	**5**	5	5	**6**	6	6	6	**0**	0	0	**6**	6	6	**0**	0

- FIFO Replacement – **17 page faults**

7	7	7	**1**	1	1	1	1	**6**	6	6	6	**0**	0	0	**6**	6	6	**0**	0
	2	2	2	2	**5**	5	5	5	**7**	7	7	7	**5**	5	5	**2**	2	2	**1**
		3	3	3	3	3	**4**	4	4	4	**1**	1	1	**4**	4	4	**3**	3	3

- Optimal Replacement – **13 page faults**

7	7	7	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	**5**	5	5	5	5	5	5	5	5	**4**	**6**	**2**	**3**	3	3
		3	3	3	3	3	**4**	**6**	**7**	7	7	**0**	0	0	0	0	0	0	0

Survey Questions:

Question 1. How much time did you spend on this homework?

- Around 4hrs. Drawing diagrams takes time.

Question 2. Rate the overall difficulty of this homework on a scale of 1 to 5 with 5 being the most difficult.

- 3/5 difficulty. The homework was not too challenging but felt relevant.

Question 3. Provide your comments on this homework (amount of work, difficulty, relevance to the lectures, form of questions).

- The amount of work this homework took was heavily dependent on how much work you were willing to show. Some guidance on how much work was necessary to show would have been useful. All the questions asked in this homework were relevant to what was taught in the lecture though I believe more time could have been on explaining exactly how the size of page tables are calculated and their entry count.