CSI3131 – Operating Systems
Tutorial 2 – Processes - Solution

1. In class, we studied execution of processes by assuming that all processes were in main memory and can be executed when ready. The state diagram in Figure 1 shows how processes move from a running state (i.e. the process' program is being executed by the CPU) to the ready state (to allow another process to run). The CPU scheduler (short term scheduler) is responsible for managing sharing the CPU among the processes ready for execution.

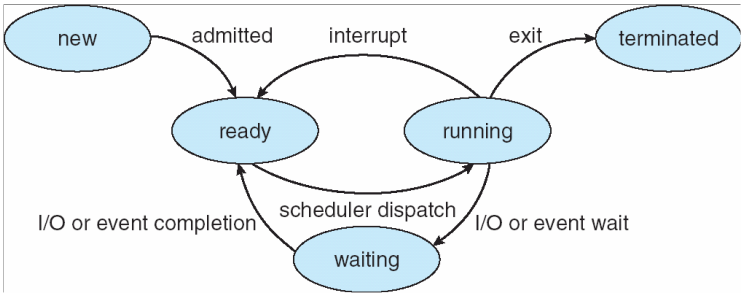


Figure 1

Recall that a medium term scheduler was responsible for "swapping out" processes to release memory for the execution of processes as illustrated in Figure 2. The use of such a scheduler increases the number of processes that can be managed by the OS.

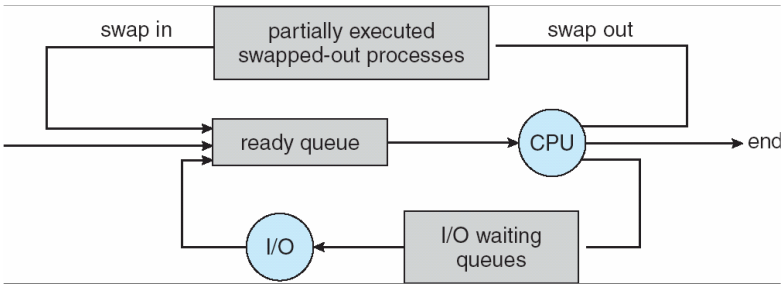
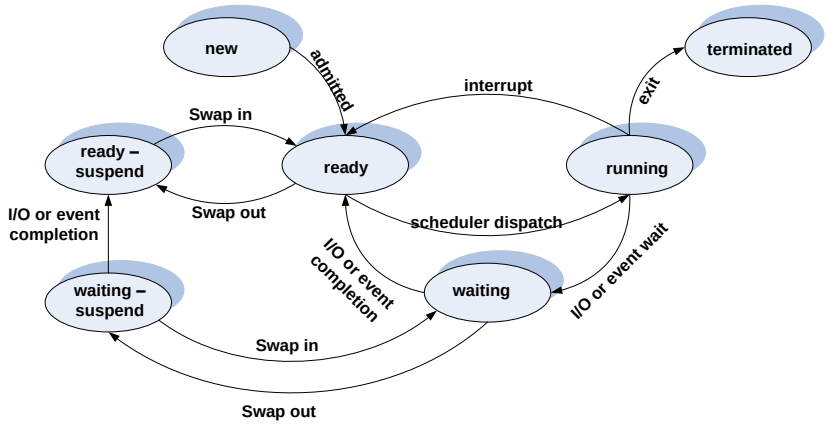


Figure 2

a) Modify the state diagram in Figure 1 to include two additional states that allow the OS to determine when a process has been swapped out. Include appropriate transitions for the new states and describe under what conditions the transitions occur.
b) Transitions from the ready or waiting state to the terminated state may be possible (these transitions are typically not shown to keep the state diagrams clear). Under what conditions would such transitions occur?

The process 7-state diagram



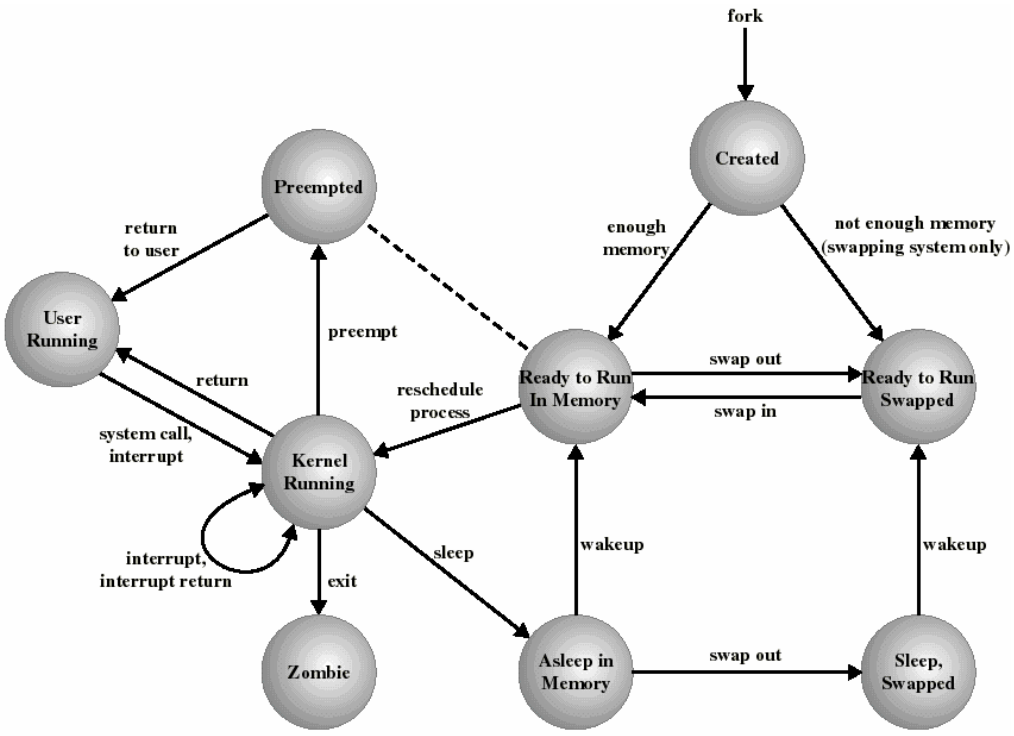
Requirement for swapping:

- **The OS can suspend (swap out) processes, i.e. transfer hem to disk. Two states are added.**
  - **waiting – suspend: block processes have been transferred to the disk**
  - **ready – suspend: processes in the ready state have been transferred to the disk.**
- **Note that a process is not swapped out when in the running state**
- **These states are used to reflect the state of the process at the moment it was swapped out.**
- **Note that the state of the process can still change event when it has been swapped out.**

New transitions

- **Two new transitions are used when the process is swapped out :**
  - **Waiting --> waiting-suspend**
    - **Processes in the waiting state are the preferred choice for swapping out processes to free memory.**
  - **Ready --> Ready- suspend (rare)**
    - **Processes in the ready state are swapped out only when not processes in the waiting state are available.**
- **Waiting-suspend --> Ready Suspend**
  - **Occurs when the expected event has occurred while the process is swapped out.**
- **Ready-Suspend --> Ready**
  - **When the process is swapped back in to main memory.**
- **Waiting-Suspend --> Waiting**
  - **Can be useful if the OS can determine that the event blocking the process in this state will occur soon and it is a high priority process (than any other process in the read-suspend state).**

The following is a state transitions diagram of a real operating system, the UNIX SVR4 (system V, release 4 – source Stallings)



Notes on the UNIX state diagram:

The running state is divided into 2 states, to distinguish when the process is running in the user mode and in the kernel mode. Note that when running in the kernel mode, the process is in fact running OS code. In class, the view was taken that when an interrupt occurs, the OS runs outside the processes. The UNIX state diagram reflects the view that the OS runs in the context of processes. The advantage of this approach is to reduce process switching. When the kernel code has completed and no process switching is required (e.g. the system call was satisfied immediately), then control is simply returned to the user code. Thus the OS is seen as a collection of subprograms run in kernel mode when some interrupt or system trap occurs.

The zombie state corresponds to the terminated state (zombie processes were discussed in class).

Asleep in memory corresponds to the waiting state.

Note that when insufficient memory is available to run a process when it is created, it is placed in swap space (on the hard drive).

Note that when dealing with hardware interrupts in kernel mode, the interrupt is serviced and control returned back to the kernel code running at the time of the interrupt

The Preempted state and ready to run in memory state are essentially the same state (as indicated by the dotted line). Processes in either state are placed in the same scheduling queue (the ready queue). When the kernel is ready to return control to the user program, it may decide to preempt the current process in favor of another that is ready and has a higher priority. In this case, the current process is placed in the preempted state.

2. Consider Figure 3 that shows and example of how scheduling queues can be defined for the 5-state diagram from Question 1.

   a) First label each of the transitions (with no label) with the action/event that triggers the action.

   b) Then indicate which transitions and queues are the responsibility of each of the following OS subsystems (that is, determine the scope of each subsystem).

      - CPU Scheduler (short term scheduler)
      - File Subsystem (includes a I/O scheduler and device drivers for dealing with the I/O on all devices including the disk): This OS component is responsible for servicing file system requests such as opening files, reading from and writing to files, and closing files; and all input/output with the computer hardware.
      - IPC subsystem: Responsible for all interprocess communications functions in the OS (including dealing with signals.

   c) Modify Figure 3 to accommodate the new 7-state diagram from Question 1. Include now a medium term scheduler as part of the OS subsystems to complete the Figure.

   d) For each queue, identify the state or states the processes in the queue will (can) have.

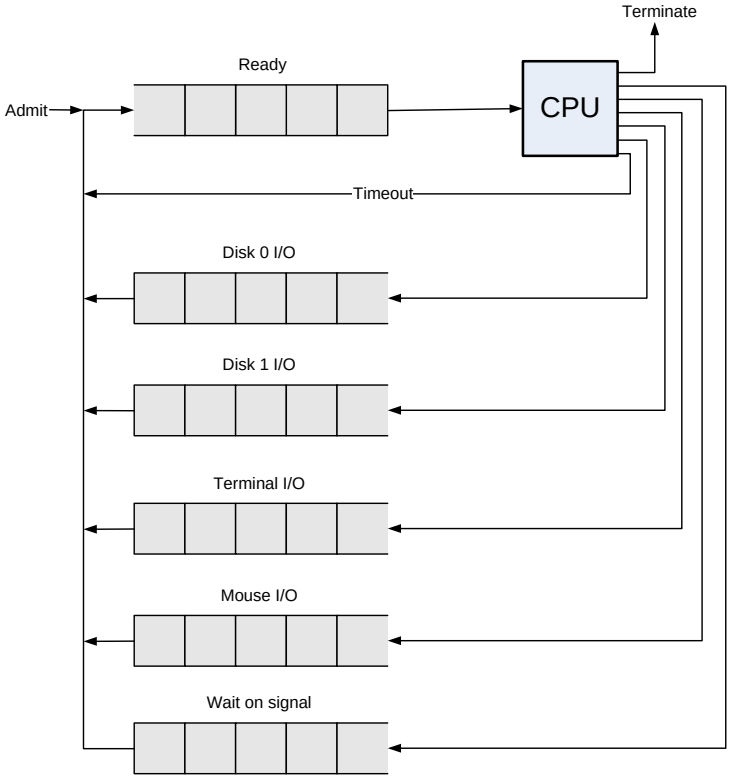
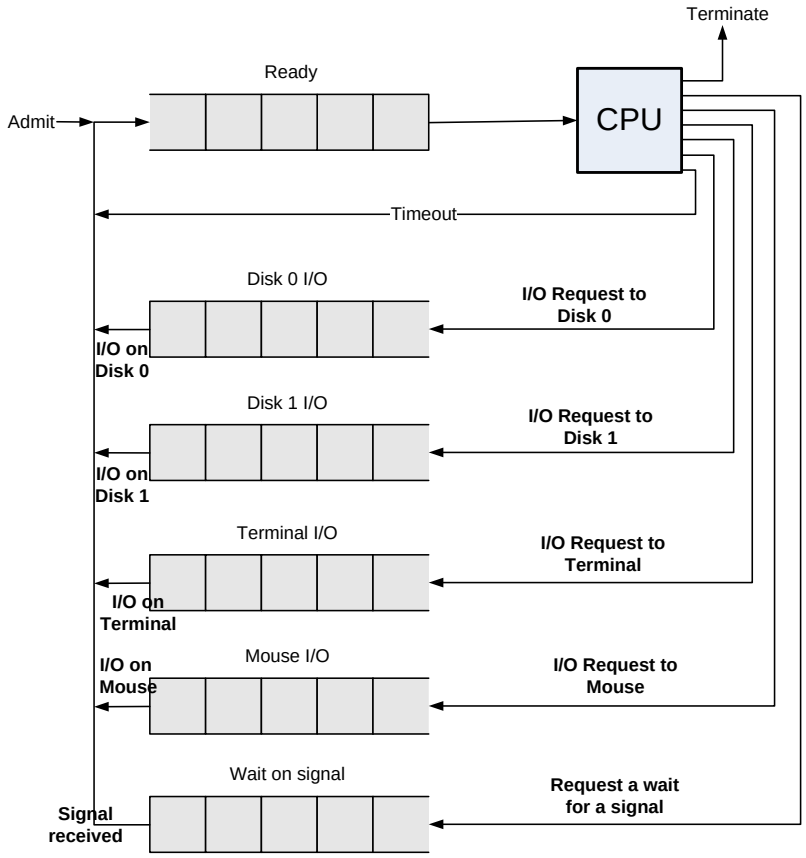
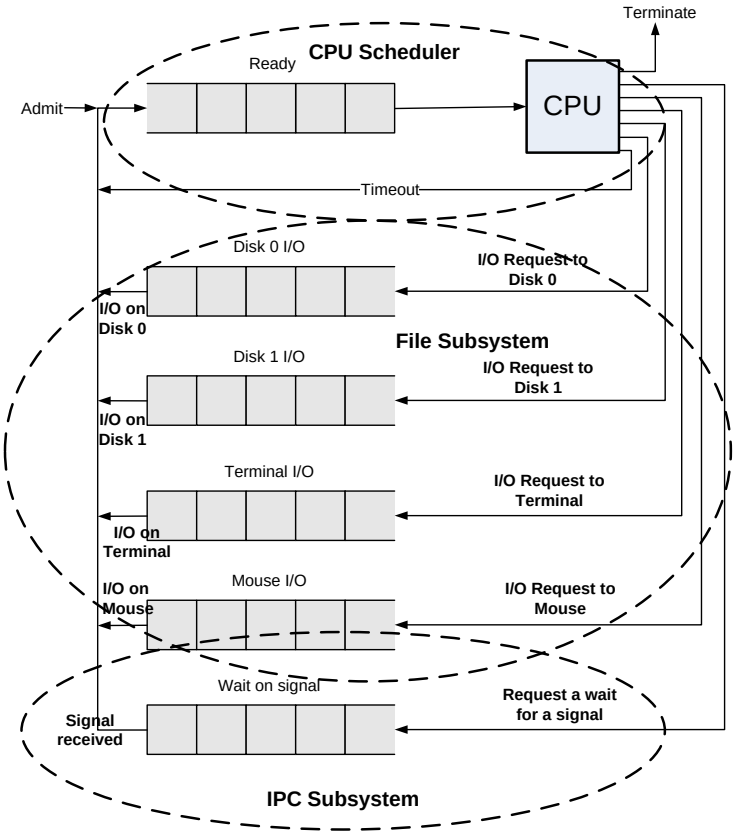


Figure 3

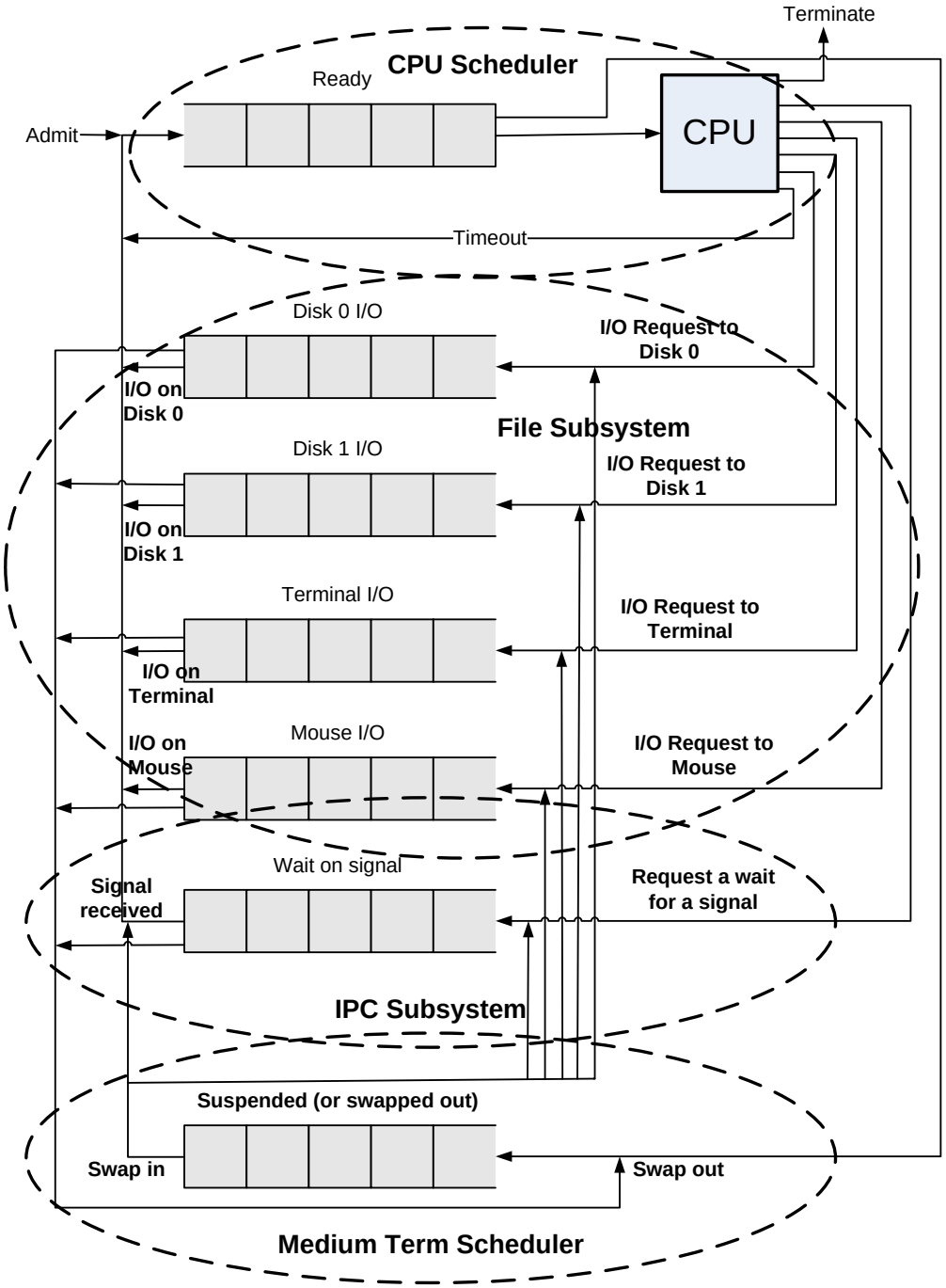
Part a)


Terminate
Ready
Admit
CPU
Timeout
Disk 0 I/O
I/O Request to Disk 0
I/O on Disk 0
Disk 1 I/O
I/O Request to Disk 1
I/O on Disk 1
Terminal I/O
I/O Request to Terminal
I/O on Terminal
I/O on Mouse
Mouse I/O
I/O Request to Mouse
Wait on signal
Request a wait for a signal
Signal received

Part b)


Terminate
CPU Scheduler
Ready
Admit
CPU
Timeout
Disk 0 I/O
I/O Request to Disk 0
I/O on Disk 0
File Subsystem
Disk 1 I/O
I/O Request to Disk 1
I/O on Disk 1
Terminal I/O
I/O Request to Terminal
I/O on Terminal
I/O on Mouse
Mouse I/O
I/O Request to Mouse
Wait on signal
Request a wait for a signal
Signal received
IPC Subsystem

Part c)



Part d)

Queue	State
Ready	**Ready**
Disk 0 I/O	**Waiting**
Disk 1 I/O	**Waiting**
Terminal 0 I/O	**Waiting**
Mouse I/O	**Waiting**
Wait on signal	**Waiting**
Suspended	**Waiting/suspend or Ready/suspend**

Note that it is acceptable to have two queues instead of the single suspended queue – one queue for each of the states "waiting/suspend" and "ready/suspend".

3. Figure 4 is a simplified diagram showing how three processes occupy memory in a running system (assume that all processes are ready for execution). Note that part of the memory has been reserved by the operating system. In this problem, two components of the operating system are run: the short term scheduler (or dispatcher) that selects a process for running on the CPU, and the file system that handles I/O requests.

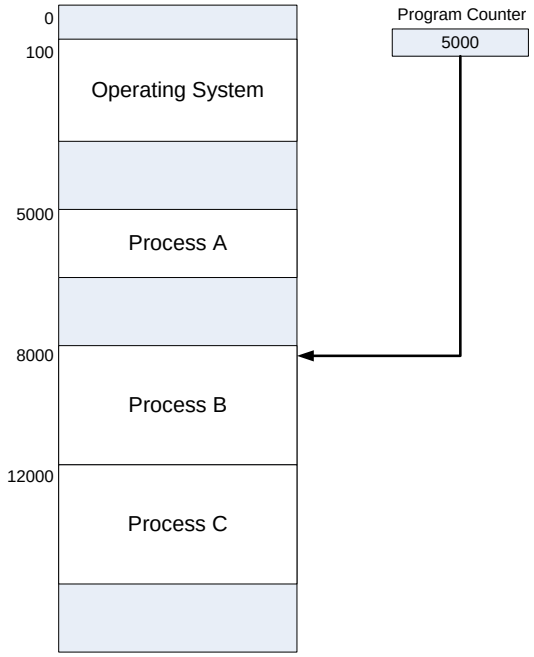


Figure 4

Table 1

Instruction Cycle	Address	Instruction Cycle	Address	Instruction Cycle	Address	Instruction Cycle	Address
1	5000	27	8002	53	12058	79	5022
2	5001	28	8003	54	12059	80	5100
3	5002	29	8004	55	12060	81	5101
4	5000	30	8005	56	12100	82	5102
5	5001	31	8006	57	12101	83	300
6	5003	32	400	58	12102	84	301
7	5004	33	401	59	300	85	302
8	5005	34	402	60	301	86	303
9	5006	35	300	61	302	87	12103
10	5010	36	301	62	303	88	12061
11	5011	37	302	63	5101	89	12062
12	5012	38	303	64	5102	90	12063
13	5010	39	12000	65	5103	91	12074
14	5011	40	12001	66	5014	92	12075
15	5012	41	12002	67	5015	93	12076
16	5010	42	12003	68	5015	94	12077
17	5011	43	12004	69	5020	95	12074
18	5012	44	12053	70	5021	96	12075
19	5013	45	12054	71	5022	97	12076
20	5100	46	12055	72	5100	99	12077
21	300	47	12056	73	5101	99	12080
22	301	48	12057	74	5102	100	12081
23	302	49	12100	75	5103	101	12082
24	303	50	12101	76	5022	102	12100
25	8000	51	12102	77	5020	103	12101
26	8001	52	12103	78	5021	104	12102

Table 1 gives a picture of the system execution of 104 instructions cycles in the running system. The address of the instructions for each cycle is shown (note that to simplify the example each instruction occupies one single address). Assume that each instruction cycle takes 1 time unit. Note that timeouts and I/O requests occur at the following times:

a) After cycle 20: Process timeout (the process has exceeded allotted time with the CPU).
b) After cycle 31: Process requested I/O on hard disk 1
c) After cycle 58: Process timeout
d) After cycle 82: Process timeout

Complete Figure 5 to show for each process how its state changes during the system execution. For each process fill in the bar to show its state at different times using the following legend:

For the operating system, indicate the times that its code is running. (To make things simple, assume that the a process moves to the ready or wait states as soon as its execution is terminated, that is, when the OS starts executing; and to the running state when the OS has completed the execution of its code).

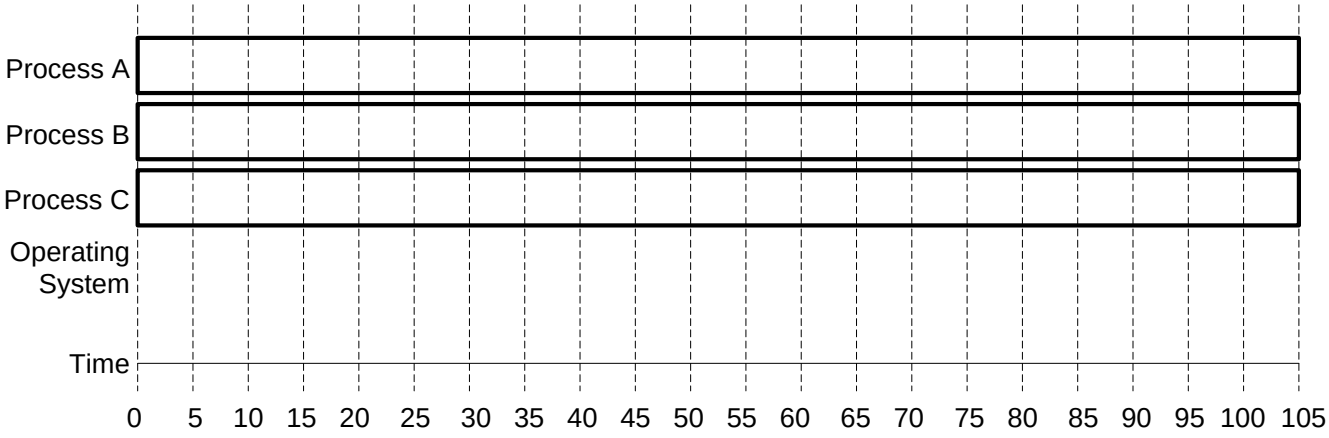
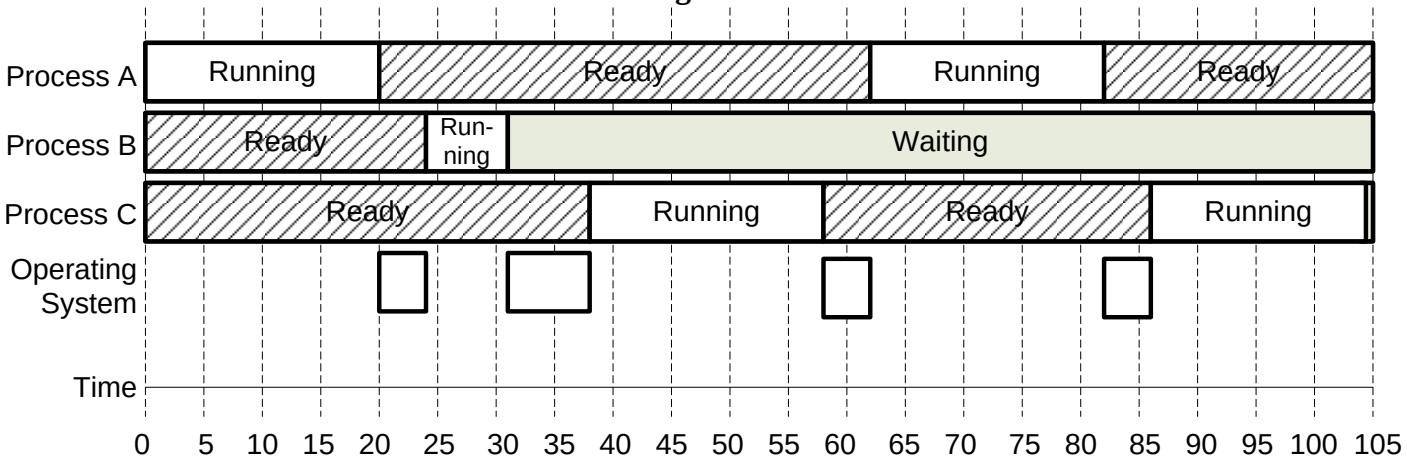


Figure 5



Determine the percentage of time that each process and the operating system are running on the CPU. Are these numbers reasonable?

Total running time is 104 time units (see Table 1).
Process a runs for 20 + 20 = 40 time units, thus 40/104 = 38% of the time
Process B runs for 7 time units, thus 7/104 = 6.7 % of the time
Process C run for 20 + 18 time units, thus 38/104 = 36% of the time (in fact this should be considered higher as the process has not completed its time allowed with the CPU).
Operating system runs for 4+7+4+4 = 19 time units, thus 18% of the time.

An operating system that runs for 18% of the time is not very well designed. One would hope to keep the OS share of the execution time under 10%. Note that in a real system, a process (and OS) would run many more instructions than shown in this example.

4. When a process creates a new process using the fork() operation, which of the following resources is shared between the parent process and the child process?
   - **a.** Stack
   - **b.** Heap
   - **c.** Shared memory

Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

5. What will LINE A output to the screen in the program below? Explain your answer.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value=5;

int main()
{
   pid_t pid;

   pid = fork();
   if(pid == 0) /* child process */
   {
      value = value + 15;
   }
   else if(pid > 0)  /* parent process */
   {
      wait(NULL);
      printf("PARENT: value = %d\n",value); /* LINE A */
      exit(0);
   }

}
```

The output is:

PARENT: value = 5

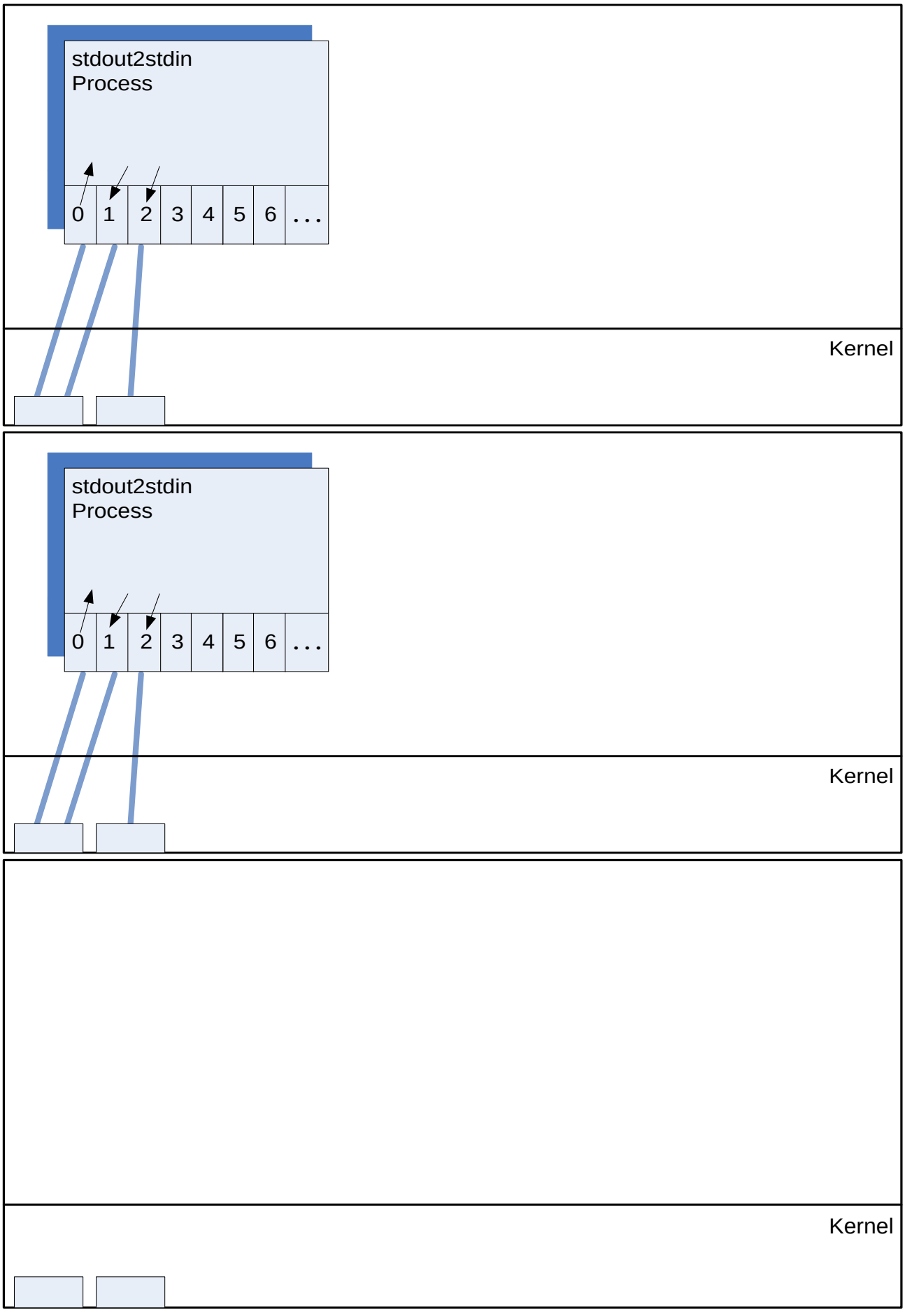
The execution of the child does not affect the value of the variable "value" in the parent process since it will have its own copy. From the point when fork is executed, two processes are running the above program each with their own copy of the program (i.e. their own copy of the variable "value"). Given that fork returns the PID of the child in the parent process, the "else" portion of the *if* statement is executed in the parent. In the child, fork retuns 0 and it is the "if" portion of the *if* statement that is executed, but the variable "value" is changed only in the child.

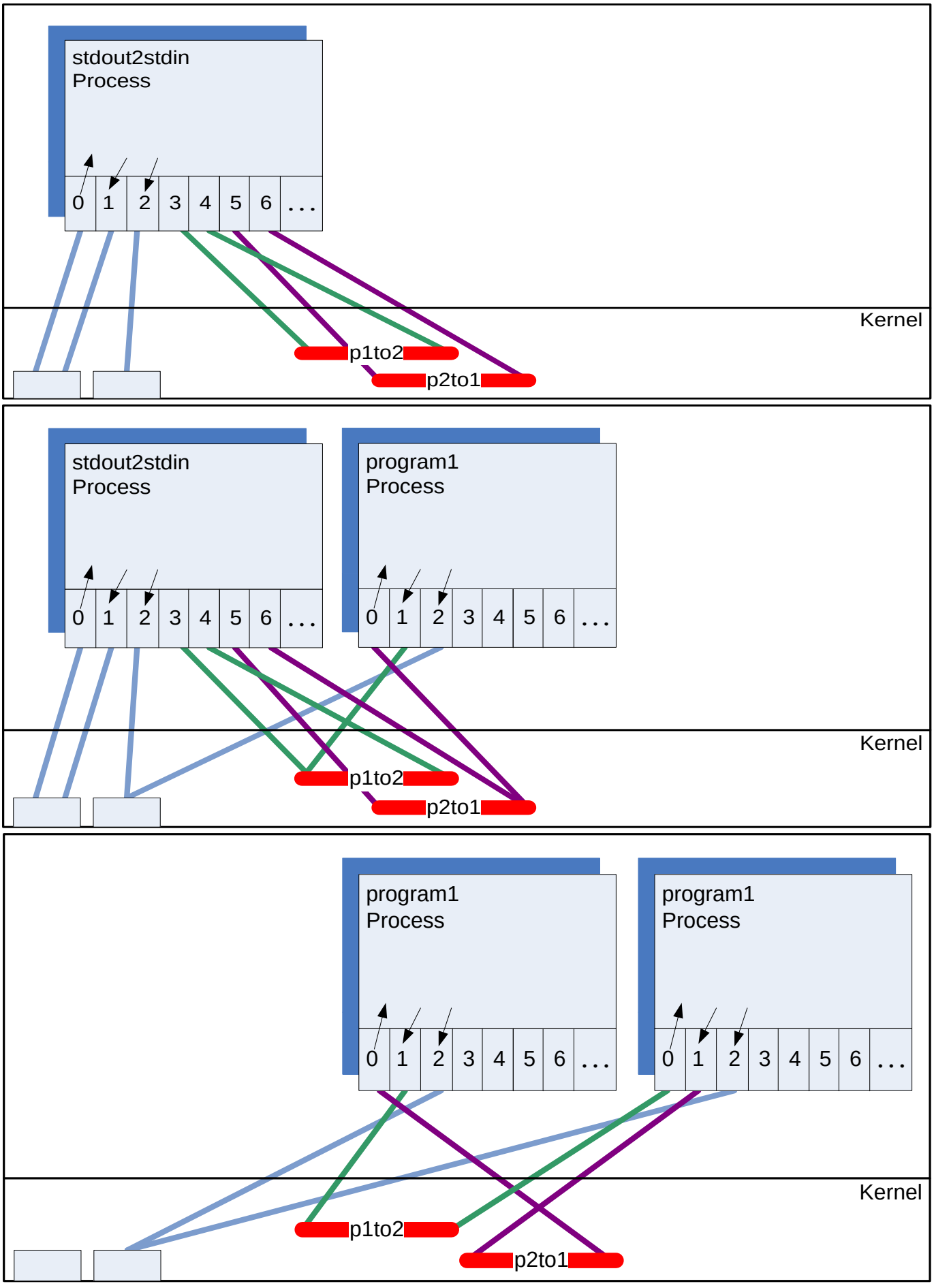
6. Examine the following C code for the program stdout2stdin.

```
int main(int argc, char *argv[])
{

   char *pgrm1;         /*pointer to first program */
   char *pgrm2;         /*pointer to second program */
   int p1to2fd[2];  /* pipe from prc1 to prc2 */
   int p2to1fd[2];  /* pipe from prc2 to prc1 */
   int pid;

   if(argc != 3)
   {
     printf("Usage: stdout2stdin <pgrm1> <pgrm2> \n");
     exit(1);
   }

   /* get programs */
   pgrm1 = argv[1];
   pgrm2 = argv[2];

   /* create the pipes */
   pipe(p1to2fd);
   pipe(p2to1fd);
            /* Complete first diagram when program is at this point */
   /* create process 1 */
   pid = fork();
   if(pid == 0)
   {
      dup2(p1to2fd[1], 1);
      dup2(p2to1fd[0], 0);
      close(p1to2fd[0]);
      close(p1to2fd[1]);
      close(p2to1fd[0]);
      close(p2to1fd[1]);
      execlp(pgrm1, pgrm1, NULL);
   }
            /* Complete second diagram when program is at this point */

   /* create process 2 */
   pid = fork();
   if(pid == 0)
   {
      dup2(p2to1fd[1], 1);
      dup2(p1to2fd[0], 0);
      close(p1to2fd[0]);
      close(p1to2fd[1]);
      close(p2to1fd[0]);
      close(p2to1fd[1]);
      execlp(pgrm2, pgrm2, NULL);
   }
}
            /* Complete third diagram when program has terminated */
```

Complete the diagrams on the next page to show what processes are created and how they are connected with pipes when the following command is executed.

```
stdout2stdin program1 program2
```


stdout2stdin
Process
0
1
2
3
4
5
6
...
Kernel
stdout2stdin
Process
0
1
2
3
4
5
6
...
Kernel
Kernel


stdout2stdin
Process
0
1
2
3
4
5
6
...
Kernel
p1to2
p2to1
stdout2stdin
Process
program1
Process
0
1
2
3
4
5
6
...
Kernel
p1to2
p2to1
program1
Process
program1
Process
0
1
2
3
4
5
6
...
Kernel
p1to2
p2to1