Stephen Chambers
Elizabeth Varki
CS 620

Homework 3

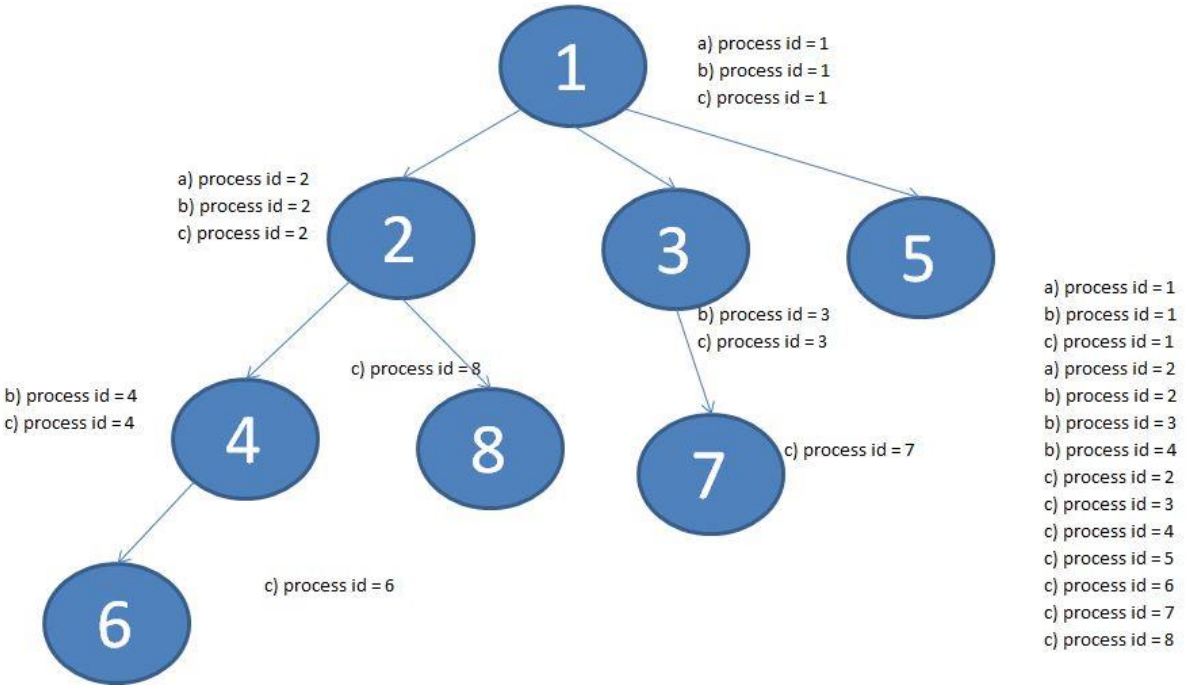
1. (5)

```
#include <stdio.h>
#include <unistd.h>

int main(){
  /* fork a child process */
  fork();
  printf(‘‘a) process id = %d", getpid());
  /* fork another child process */
  fork();
  printf(‘‘b)process id = %d", getpid());
  /* and fork another */
  fork();
  printf(‘‘c)process id = %d’’, getpid());
  return 0;
}
```

Including the initial parent process, how many processes are created by the above program? Explain and draw the process tree diagram. Write the output next to the diagram so that the connection between the output and the process is clear.

At the first fork the parent creates a child process, meaning there are two processes running. At the second fork the two processes running all create child processes, meaning there are four processes running. Finally, at the third fork all four processes running create child processes, meaning that there are eight processes running in total at the end of the code. The children are going to race when executing “printf” and creating other children because no parents wait for their children to complete. Therefore, there is no “correct” output. The parent is process 1 and the children process IDs are incrementing by one.



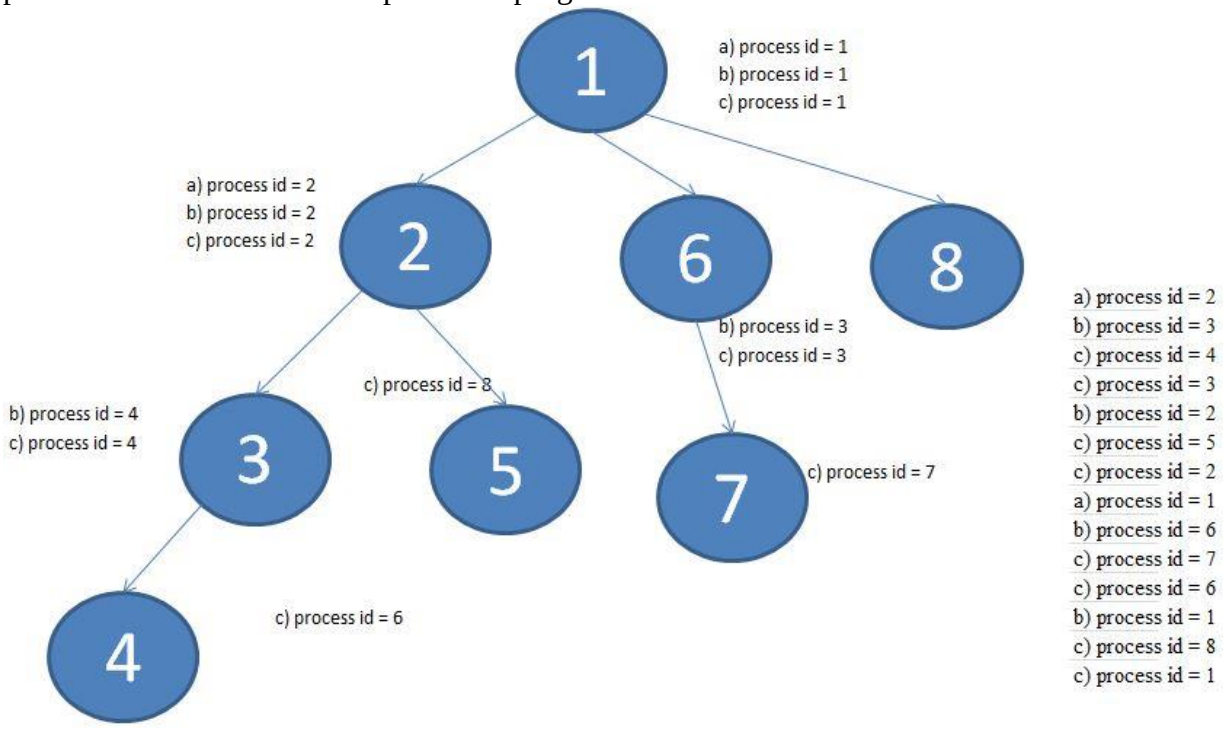
2. (5) Add the wait() system call to the code segment:

```
#include <stdio.h>
#include <unistd.h>

int main(){
  /* fork a child process */
  fork();
  wait(&status);
  printf(‘‘a) process id = %d’’, getpid());
  /* fork another child process */
  fork();
  wait(&status);
  printf(‘‘b)process id = %d’’, getpid());
  /* and fork another */
  fork();
  wait (&status);
  printf(‘‘c)process id = %d’’, getpid());
  return 0;
}
```

How does the inclusion of wait() impact the code? Do the number of processes change? Does the output change? Explain your answers.

When the wait command is added, the number of processes does not change, but the output does change. This is because the parent has to wait for its children to complete before it can execute more instructions. Therefore, there is a “correct” solution to the order of the processes in the process tree as well as the output of the program.



3. (25) Calculate the mean response time (R) for the following scheduling policies using the workload given in Table 1.

(a) SRTN (Shortest Remaining Time Next)
(b) LIFO preemptive
(c) RR with quantum (Qt) = 2 time units.
(d) MLF non-preemptive
(e) MLF preemptive

Table 1: Workload submitted to CPU		
Process	**Arrival Time**	**CPU Burst**
p1	**0**	**2**
p2	**1**	**4**
p3	**3**	**4**
p4	**6**	**1**
p5	**8**	**3**

Assume that context switch time is 0. For preemptive policies, assume that if a job arrives at the same time that another job finishes its quantum, then the arriving job gets into the CPU queue ahead of the job that just finished its quantum. For MLF, There are 3 Queues: qt1=1; qt2=2; qt3=3.

SRTN: MEAN RT = 4.4

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*RT*
P1	E	E														2
P2		W	E	E	E	E										5
P3				W	W	W	W	E	E	E	E					8
P4							E									1
P5									W	W	W	E	E	E		6

LIFO Premptive: MEAN RT = 7

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*RT*
P1	E	W	W	W	W	W	W	W	W	W	W	W	W	W	E	14
P2		E	E	W	W	W	W	W	W	W	W	W	E	E		12
P3				E	E	E	W	E								5
P4							E									1
P5									E	E	E					3

Round Robin: MEAN RT = 5.2

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*RT*
P1	E	E														2
P2		W	E	E	W	W	E	E								7
P3				W	E	E	W	W	W	E	E					8
P4							W	W	E							3
P5									W	W	W	E	E	E		6

MLF Non-Premptive: MEAN RT = 6.2

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*RT*
P1	E1	W2	E2													3
P2		E1	W2	W2	E2	E2	W3	W3	W3	W3	W3	W3	E3			12
P3				E1	W2	W2	W2	E2	E2	W3	W3	W3	W3	E3		11
P4							E1									1
P5									W1	E1	E2	E2				4

MLF Premptive: MEAN RT = 6.2

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*RT*
P1	E1	W2	E2													3
P2		E1	W2	W2	E2	E2	W3	W3	W3	W3	W3	W3	E3			12
P3				E1	W2	W2	W2	E2	W2	E2	W3	W3	W3	E3		11
P4							E1									1
P5									E1	W2	E2	E2				4

4. (6) Consider a multiprocessor system and a multi-threaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios:
(a) The number of kernel threads allocated to the program is less than the number of processors.
(b) The number of kernel threads allocated to the program is equal to the number of processors.
(c) The number of kernel threads allocated to the program is greater than the number processors but less than the number of user-level threads.
a) Since there are less kernel threads than processers, some of the processers will be idle. Having this number of kernel threads does not utilize the systems resources most efficiently.

b) In this scenario, there are no idle processors. This allows the systems resources to be maximized, assuming that no threads were blocked for I/O or any other reason, assuming the number of user threads is greater than the number of processers in the system.

c) This is a very similar example to part b, but with one small caveat. Since the amount of kernel threads is greater than the number of processors, another kernel thread can be switched to if the kernel thread running is blocked.

5. (2) If a child dies before its parent calls wait(), the child becomes a zombie. What is a zombie (in terms of process data structures)?
The OS erases the address space when the child dies, as is customary when any process is completed. However, the OS retains the process table entry of the child.

6. (2) What is the reason behind making the child a zombie, instead of just erasing the child completely from memory?
The OS retains the process table entry of the child because it assumes the parent process wants to know what status the child exited with.

7. (9) Consider a system running 50 I/O-bound jobs and 5 CPU-bound jobs constantly. Assume that the I/O-bound jobs issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Assume that the CPU-bound jobs use up their entire quantum quota on the CPU. Also assume that the context switching overhead is 0.1 millisecond and that all processes are long-running jobs. What is the percentage of CPU time wasted in context switching when:

The processes are running on the CPU constantly. Therefore, the amount of I/O bound jobs and CPU bound jobs is arbitrary when discussing the percentage of CPU time wasted in context switching. In general, the percentage of wasted CPU time is going to decrease as the quantum time increases. This is because the low quantum time is wasting the fact that the CPU bound jobs do not have to repeatedly switch for I/O and waste CPU time context switching. The percentage of CPU time wasted is:

(1 – Total Time / (Total Time + Context Switching Time)) * 100

a) The time quantum is 1 milliseconds

The CPU is going to perform a context switch every one millisecond, regardless of the type of job. This is because the time quantum is equal to the amount of time I/O bound jobs can stay on the CPU before context switching. Therefore, the percentage of CPU time wasted is (1 - 1/1.1) * 100 or 9%.

b) The time quantum is 10 milliseconds

The I/O bound job will have to switch ten times before it is completed, or 10*1.1, same as always. However, the CPU job is only going to take 10.1 seconds because it can run on the CPU for 10 milliseconds without having to perform a context switch. Therefore, the percentage of CPU time wasted is (1 - (10+10 / 10*1.1 + 10.1)) * 100 or (1- 20/21.1) * 100 or 5.21%.

(c) The time quantum is 30 milliseconds

The I/O bound job will still have to switch ten times before it is completed. The CPU can now run for 30 milliseconds without having to perform a context switch. Therefore the percentage of CPU time wasted is (1- 10+30 / 10*1.1 + 30.1) * 100 or (1-40/41.1) * 100 or 2.68%.

8. (10) Which of the following scheduling algorithms could result in starvation? Give a brief explanation in each case.

As a disclaimer, a long job will take a long time regardless of which scheduling policy is used. However, some policies are fairer than other policies, and that is what the explanations will be based on. It is also assumed that all policies except for part d are non-preemptive.

(a) First come first served (FCFS) *no*

The processes in this algorithm are based upon arrival time. Therefore, it is not possible for a job to *never* gain the CPU. However, if the jobs near the head of the queue are long jobs, than the jobs in the middle and the tail end queue may have to wait a very long time, as they have to wait for those jobs to finish before executing. The absence of priorities makes them unable to be starved however.

(b) Shortest job first (SJF) *yes*
If a job is a long job, it could constantly lose the CPU to shorter jobs even though the policy is non-preemptive. As long as a job exists that is shorter than the time remaining on the long job, the long job will starve.

(c) MultiLevel Feedback (MLF) *yes*
This is an unfair policy towards long jobs because a job will constantly get bumped down priority queues until it is completed.

(d) Last come first served preemptive (LCFS) *yes*
A long job near the beginning of the queue will be starved because all shorter jobs after it will get priority, as the policy is preemptive. Additionally, jobs near the beginning of the queue will continue to be starved as long as more processes enter the ready queue.

(e) Round Robin (RR) *no*
Although a job could starve if it was long and there was a very short quantum time, this is a very fair policy. Each job gets a finite time on the CPU regardless of how long the job is.

9. (4) Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O bound programs and yet not permanently starve CPU-bound programs?
This algorithm will favor I/O bound programs over CPU bound programs because I/O bound programs will generally always use less processing time than a CPU bound job in the recent past because it has to constantly block for I/O. However, while the I/O bound job is being blocked, the processor can focus on the CPU bound jobs, resulting in the CPU bound jobs not being permanently starved.

10. (3) How do user level threads get to execute when the operating system has no knowledge of a user thread?
A user level thread shares the parents processing time on the CPU, which is how they get to execute. Additionally, it is very common to map user level threads to kernel threads, the latter of which the operating system is well aware of. The operating system gives time on the CPU to kernel threads, which can then execute the user threads that are mapped to it.

11. (5) In class, I mentioned that LIFO preemptive performs closest to Shortest Job First. Could FIFO perform better than LIFO preemptive? If yes, give an example. If no, justify.
FIFO cannot perform *better* than LIFO preemptive. This is because LIFO preemptive jobs are more biased towards short jobs. On the contrary, FIFO is biased towards long jobs. Policies that are biased towards short jobs will always have a higher throughput than a counterpart policy that is biased towards long jobs. This is because by definition, a scheduling policy will have a higher throughput if more jobs are completed This means that LIFO preemptive can perform more jobs than FIFO given the same amount of time.

12. (5) Is it possible for SJF non-preemptive to perform better than SRTN (SJF preemptive)? If so, give an example. If not, justify.

It is not possible for SJF non-preemptive to perform *better* than SJF preemptive. While both scheduling policies are biased towards short jobs, SJF preemptive is even more biased toward short jobs than SJF non-preemptive. This is because short jobs get to interrupt the currently executing job if it is longer and execute immediately, while the short job would have to wait for the longer job to complete in SJF non-preemptive. This means that SJF preemptive can a higher amount of jobs completed in the same amount of time as SJF non-preemptive.

13. (9) Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate A; when it is running, its priority changes at a rate B. All processes are given a priority of 0 when they enter the ready queue. The parameters A and B can be set to give many different scheduling algorithms.

(a) What is the algorithm that results from B > A > 0? Justify your answer.

(b) What is the algorithm that results from A < B < 0? Justify your answer.

a) FIFO is the algorithm that satisfies these conditions. This is because B, or the priority rate when the process is running, increases faster than A, the priority rate when the process is not running. Additionally, all processes are given a priority of 0 when they enter the ready queue. This will cause whatever process that gets in the CPU first to have more priority than the processes waiting in the waiting queue *and* any new processes that are entering the ready queue. Therefore, the *first* one in the queue that starts running will be the *first* one out.

b) LIFO is the algorithm that satisfies these conditions. Priority B, or the priority rate when the process is running, decreases slower than A, the priority rate when the process is not running. When a new process comes into the ready queue, it automatically has the highest priority than any other process in the ready queue, and it decreases slower than the process that is running. Therefore, the **last** process to arrive in the queue will always have the highest priority and will execute **first**.

14. (5) Consider a multiprocessor CPU scheduling policy. There are 2 options: 1) a single common ready queue of jobs; when a CPU becomes free, the job at the head of the queue goes to this free CPU. 2) a ready queue for each CPU; the arriving job joins the shortest queue. In general, do you expect the common queue or the shortest queue policy to perform better. Justify.

In *general* terms, a ready queue for each CPU will not perform to the same level as if there were a single common ready queue of jobs. This is because there is a possibility that a CPU will sit idle if there is nothing in the ready queue. Consider the following example with 15 processes.

[ ] = Queue element
Option 1

CPU1 CPU2 CPU3

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

Option 2:

CPU1 [ ] [ ] [ ] [ ] [ ] CPU2 [ ] [ ] [ ] [ ] [ ] CPU3 [ ] [ ] [ ] [ ] [ ]

Under Option 2, if CPU2 finished all jobs in its ready queue, it would have to sit idle while waiting for another job, even though there could be other jobs waiting in CPU1 and CPU3. Under Option 1, this scenario would not occur, because CPU2 would have additional jobs that it would be able to execute.

15. (5) Consider a scheduling policy for a system with soft real time jobs. How does the CPU scheduler for a system with soft real time jobs fundamentally differ from a scheduler that does not have to deal with real time jobs?

The fundamental difference between these two CPU schedulers is that the CPU scheduler with soft real time jobs has to keep into account individual job priorities. Additionally, most real time jobs have a release time and a deadline, or a start and stop time, respectively. This means that if an event triggers that requires a soft real time job to be executed, it will interrupt the CPU in order to run. Furthermore, a soft real time job needs enough time on the CPU to complete its task, so it may stay on the CPU longer than other jobs. However, the release time and the deadline are not as crucial to a system with soft real time jobs as they are to a system with hard real time jobs, so they may not be followed strictly.