Toby Chappell

CPSC 380

Homework #2

Part II – Process Management

1. Describe the actions taken by a kernel to context-switch between processes.

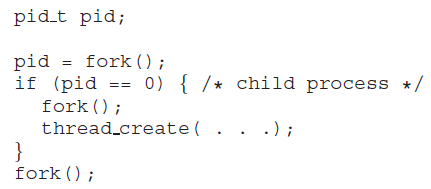
(3 pts)

The system must save the current context of the process running on the CPU (known as a state save). Next, the system needs to load the saved context of a different process (known as a state restore).

1. When a process creates a new process using the *fork( )* operation, which of the following state is shared between the parent process and the child process? (3 pts)
   1. Stack
   2. Heap
   3. Shared memory segments
2. Which of the following components of program state are shared across threads in a multithreaded process? (4 pts)
   1. Register values
   2. Heap memory
   3. Global variables
   4. Stack memory
3. Is it possible to have concurrency but not parallelism? Explain. (4 pts)

Yes, it is possible to have concurrency but not parallelism. The reason for this is because concurrency can be done on a single core but parallelism requires multiple cores. So, if a system only has a single core, it can have concurrency but not parallelism.

1. Consider the following code segment: (2 pts)



* 1. How many unique processes are created?

There are six unique process created (assuming the parent process is also considered).

* 1. How many unique threads are created?

Two unique threads are created.

1. What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other? (6 pts)

For one, user-level threads are managed without kernel support and kernel-level threads are managed by the operating system. Additionally, the existence of user-level threads is unknown to the kernel while kernel-level threads are known to the kernel. Kernel-level threads are better than user-level threads when running in a multiprocessor environment. The reason for this is because kernel-level threads will take advantage of the multiple processors available and run simultaneously (user-level threads will only use a single processor even if multiple processors are available). User-level threads are better than kernel-level threads when the kernel is time shared. The reason for this is because, in a time-shared environment, context switching takes place much more often. Since, context switching user-level threads has little to no overhead (unlike context switching kernel-level threads which has high overhead), user-level threads would perform better.

1. Race conditions are possible in many computer systems. Consider a banking system with two methods: deposit(amount) and withdraw(amount). These two methods are passed the amount that is to be deposited or withdrawn from a bank account. Assume that a husband and wife share a bank account and that concurrently the husband calls the withdraw() method and the wife calls deposit(). Describe how a race condition is possible. (6 pts)

A race condition is possible since both the wife and husband have access to the “Current Balance” variable. If they were to access that variable simultaneously, they can inadvertently change the variable in display a value that is incorrect (assuming that the access/update is not properly synchronized). For instance, assume that the Current Balance is $100. The. husband and wife then both access the Current Balance. Next, the husband withdraws $50 changing the Current Balance to $50 while the wife deposits $50 changing the Current Balance to $150. The Current Balance is then updated to display either $50 or $150, when the correct balance should be $100.

1. Explain why spinlocks are not appropriate for single-processor systems yet are used in multiprocessor systems. (4 pts)

Spinlocks should not be used for single-processor systems since the condition to break a process out of a spinlock can be obtained only by executing another process. If the process does not permit other programs to run on the processor, there is no way for the process to break out of the spinlock. However, in a multiprocessor system, other processes can execute in parallel on other processors and therefore are able to release the first process from the spinlock.

1. A multithreaded web server wishes to keep track of the number of requests it services (known as **hits**.) Consider the following two strategies to prevent a race condition on the variable hits. The first strategy is to use a basic mutex lock when updating hits: (3 pts)

*int hits;*

*mutex lock hit lock;*

*hit lock.acquire();*

*hits++;*

*hit lock.release();*

A second strategy is to use an atomic integer:

*atomic t hits;*

*atomic inc(&hits);*

Explain which of these two strategies is more efficient.

The second strategy is more efficient. The reason for this is because the atomic integer provides an atomic update of the hits variable and therefore makes sure that there is no race condition on hits. This involves no kernel intervention unlike the first strategy (which requires a system call and can possibly put a process to sleep if the lock is unavailable).

1. Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism — a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available: (3 pts)
   1. The lock is to be held for a short duration.

Spinlock

* 1. The lock is to be held for a long duration.

Mutex lock

* 1. The thread may be put to sleep while holding the lock.

Mutex lock

1. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs? (4 pts)

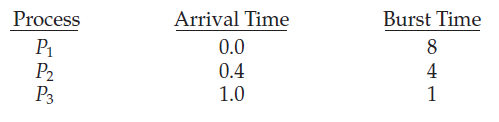
I/O-bound programs generally do not use their entire CPU quantum whereas CPU-bound programs do. As such, the scheduler can make better use of the computer’s resources by giving a higher priority to I/O-bound programs and therefore them to execute ahead of CPU-bound programs.

1. Explain the difference between preemptive and non-preemptive scheduling. (4 pts)

Preemptive Scheduling: There is a choice in terms of scheduling (for instance when an interrupt occurs or at completion of I/O). Preemptive scheduling can result in race conditions when data is shared among multiple processes.

Non-Preemptive Scheduling: There is no choice in terms of scheduling (a new process must be selected for execution). Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU.

1. Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use non-preemptive scheduling, and base all decisions on the information you have at the time the decision must be made. (4 pts)



* 1. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

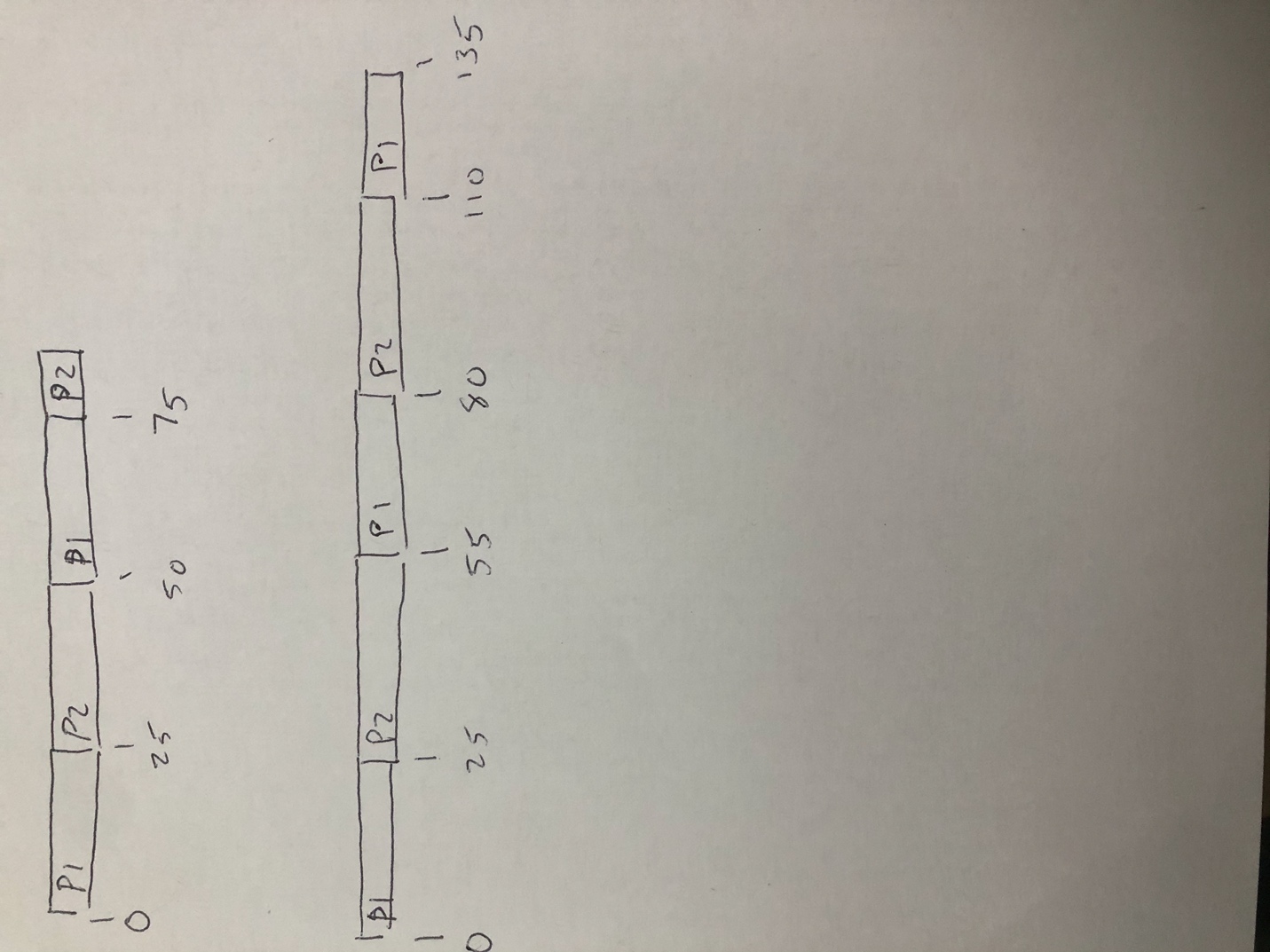
Average Turnaround Time = Sum(Waiting Time + Burst Time)/(Number of Processes) = ((0+8)+(7.6+4)+(11+1))/3 = 10.53

* 1. What is the average turnaround time for these processes with the SJF scheduling algorithm?

Average Turnaround Time = ((0+8)+(7+1)+(8.6+4))/3 = 9.53

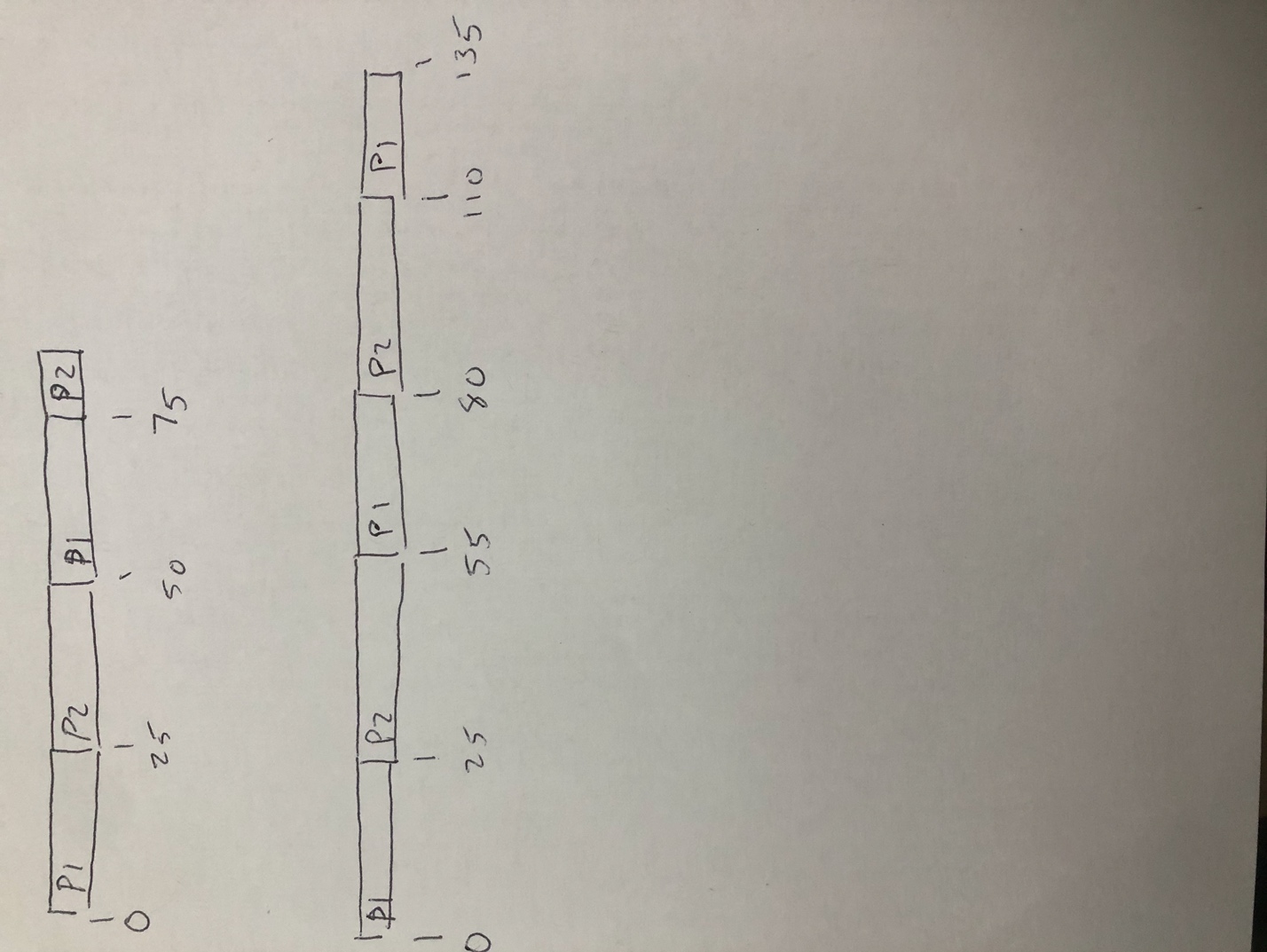
1. Consider two processes, *P*1 and *P*2, where *p*1 = 50, *t*1 = 25, *p*2 = 75, and *t*2 = 30. (8 pts)
   1. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart?

These processes cannot be scheduled using rate-monotonic scheduling. P1 runs first (since it has higher priority) completing its burst at 25. Then, P2 runs till 50. However, at time 50, P1 is available so it preempts P2 and runs till 75. After P1 finishes its task, P2 is available to run, but there is a remained CPU burst of 5 which causes a miss.



* 1. Illustrate the scheduling of these two processes using earliest deadline-first (EDF) scheduling.

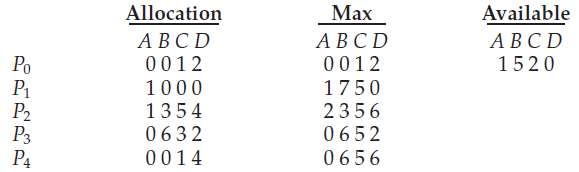
These processes can be scheduled using earliest deadline-first scheduling. P1 runs first (since it has higher priority) completing its burst at 25. Then, P2 runs till 50 and P1 becomes available. Then, the algorithm checks the next deadlines of P1 and P2. Since P2 has an earlier deadline (P1’s next deadline is 100 and P2’s next deadline is 75), P2 is allowed to complete its burst time so it runs till 55. P1 runs until 75 when P2 becomes available so the algorithm again checks which process has an earlier deadline. P1 has an earlier deadline (at time 100 vs P2 which has a deadline at 150) so it runs till it finishes its burst at 80. P2 then runs until 100 when P1 becomes available. Since both processes have the same next deadline (150), P2 finishes its burst at 110 and P1 runs until 135.



1. 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. Show that the system is deadlock-free. (4 pts)

If the system is deadlocked, each process is holding is holding a resource and waiting for another. However, since there are 4 resources and 3 processes, one of the processes is able to obtain a second resource. Since this process will not require more resources, it will return its resources when it is done, meaning the system is deadlock-free.

1. Consider the following snapshot of a system: (6 pts)



Answer the following questions using the banker’s algorithm:

* 1. What is the content of the matrix **Need**?

P0 = (0, 0, 0, 0)

P1 = (0, 7, 5, 0)

P2 = (1, 0, 0, 2)

P3 = (0, 0, 2, 0)

P4 = (0, 6, 4, 2)

* 1. Is the system in a safe state?

Yes, since Available is equal to (1, 5, 2, 0), P0 or P3 can run. Once P3 runs, it releases its resources allowing all other processes to run.

1. Is it possible to have a deadlock involving only one single-threaded process? Explain your answer. (2 pts)

No, it is not possible to have a deadlock involving only one single process. Since deadlock involves a circular “hold-and-wait” condition between two or more processes, “one” process cannot hold a resource and simultaneously be waiting for another resource that it is holding.