Math 203
The List Processing Algorithm

An algorithm is a specific sequence of steps to follow in order to solve a problem. Given the same input data, two people following the algorithm should produce the exact same output. This means that the algorithm should specify how decisions are made at every single step, with no room for interpretation by the reader. Our list processing algorithm, which we described informally as "As each processor finishes a task, assign it the highest priority task that the order requirement digraph says is available", can be turned into a 'real' algorithm with a bit more care.

Recall that our basic assumptions are:

Each processor can do any of the tasks (no specialists).

Once a task is started, it is completed without stopping.

When a processor finishes, it will immediately start a task if one is available.

An order-requirement digraph tells us when a task is available (i.e., if it's prerequisite tasks have been completed).

A priority list tells us which jobs we want to do first. Essentially, the priority list breaks ties when one processor has more than one available task to choose from.

The algorithm begins with a given collection of processors, which we number from 1 on. If two processors are ready to begin a task, the lower-numbered processor <u>always</u> chooses a task first. (This ensures that we will all always choose the same processor to work on a particular task.) A task T is *available* if, according to the o.r. digraph, all of the tasks with arrows pointing to T have been completed.

We start by assigning the first processor the highest priority task that is available. We continue down the list of processors, assigning them the next highest priority available task until we either run out of processors or run out of available tasks. Any processor that is not initially assigned a task is designated as *idle*, and waits for a task to become available. The processors that are working on a task are not idle, and will continue to work on the task until the time the task was started, plus the time that the task requires, has elapsed. Any tasks that are available but unassigned are marked as available (in the priority list), and all tasks that have just been assigned to processors are marked as being worked upon. (E.g., underline available tasks and half-strike (\) working tasks?)

All of the action now occurs each time a task is finished. As a task or tasks finish (i.e., as time moves to the time that the task started plus the time required to complete the task), they are marked as completed (complete the strike through with a / ?), and the list of available tasks is updated (mark the task(s) T with the property that all tasks pointing *to* T are now completely struck through (X)). The processor(s) finishing the task(s) are now considered to be idle. If there are now any available tasks, we assign the highest priority available task to the lowest-numbered idle processor (and mark the task as being worked on), and we again repeat this until we either run out of processors or run out of available tasks. We then wait for the next worked upon task to finish, and repeat the process again.

We continue to use this procedure to assign tasks to processors, until all tasks have been assigned. As the tasks now finish, the processors are set to idle; when the last running task finishes, the project is finished.

We could modify this procedure to assign tasks more "equitably", e.g., by assigning the highest priority available task to the lowest numbered processor with the highest total amount of idle time to that point (i.e., tasks go to the "most idle" worker). But our main focus is really on determining how quickly the entire project can be completed, and this kind of modification will not, in our simplified model (no worker fatigue?), have any effect on the overall completion time....